Année 1998

THÈSE

présentée en vue de
l'obtention du titre de

DOCTEUR

de

**L'ÉCOLE NATIONALE SUPÉRIEURE
DE L'AÉRONAUTIQUE ET DE L'ESPACE**

SPÉCIALITÉ : INFORMATIQUE

par

Simon de GIVRY

ALGORITHMES D'OPTIMISATION SOUS CONTRAINTES ÉTUDIÉS DANS UN CADRE TEMPS RÉEL

Soutenue le ... devant la Commission d'Examen :

MM.	**M. ADELANTADO**	**Co-directeur de thèse**
	Y. CASEAU	**Rapporteur**
	F. CHARPILLET	
	E. GREGOIRE	**Rapporteur**
	R. SARAZIN	
	G. VERFAILLIE	**Directeur de thèse**

Thèse préparée au sein du Département d'Études et de Recherches en Informatique
de l'ONERA CERT.

Table des matières

Liste des figures

Introduction générale

Les systèmes temps réels "intelligents", c'est à dire capables de prendre des décisions en temps contraint, constituent un nouveau défi. La nouveauté provient de la nature des tâches à ordonnancer sur le processeur : une prise de décision n'est pas vue comme un résultat complet, optimum, obtenu en temps borné; elle correspond plutôt à un résultat approché, de qualité croissante en fonction du temps alloué. Le système ne vise plus seulement à garantir les échéances temporelles d'un ensemble de tâches; il essaie aussi de maximiser la *qualité de service*.

Nous avons choisi le cadre des problèmes d'optimisation sous contraintes pour modéliser des problèmes de décision. C'est un nouveau formalisme, appelé *Problèmes de Satisfaction de Contraintes Valuées*, qui a été retenu pour représenter ces problèmes d'optimisation. Pour un problème donné, il est difficile de prédire l'évolution de la qualité, afin de l'ordonnancer au mieux vis à vis des autres tâches gérées par le système. Notre approche pragmatique consiste à encadrer progressivement l'optimum lors de la résolution d'un problème. Cet encadrement est une information exacte de l'état d'avancement d'une tâche. C'est un critère important pour répartir les efforts de calcul pour un ensemble de tâches d'optimisation.

Concrètement, pour des problèmes de minimisation, l'encadrement est obtenu par coopération de *méthodes incomplètes* pour produire des majorants et de *méthodes complètes* pour produire des minorants. Nos travaux portent sur la production de minorants. Nous décrivons plusieurs méthodes originales et génériques pour simplifier la structure d'un problème. Nous examinons aussi des algorithmes conçus spécifiquement pour la preuve de minorant. Ces techniques sont expérimentées sur des problèmes générés aléatoirement, ainsi que sur des problèmes réels.

L'objectif de cette thèse est donc double. D'abord, il s'agit de montrer l'intérêt, pour l'optimisation en temps limité, d'une approche délibérative fondée sur la connaissance de la qualité. Cette connaissance peut être obtenue à l'exécution par un encadrement progressif de l'optimum. Ensuite, il faut proposer des moyens de calculer cet encadrement. Le calcul de minorants à l'aide de simplifications génériques est un domaine ouvert. Notons que le terme "temps limité" est volontairement choisi plutôt que le terme "temps réel", plus fort. L'affaiblissement porte d'une part sur le déterminisme qui est remplacé par la notion de maximisation de la qualité de service. Et d'autre part, sur la nature des contraintes temporelles. Plutôt que d'être stricte, périodique, très courte, elle est globale, évolutive et dans une échelle de temps compatible avec les capacités des algorithmes d'optimisation actuels.

La thèse est séparée en deux parties. La première est appelée "Approche délibérative en optimisation sous contraintes". Les deux premiers chapitres donnent un état de l'art, enrichi d'expérimentations, d'une part sur les solutions délibératives pour des systèmes temps réel intelligents et d'autre part sur les techniques liées aux Problèmes de Satisfaction de Contraintes Valuées. Le chapitre suivant tente de composer ces deux domaines, en amenant une réflexion et des solutions pour l'optimisation sous contraintes en temps limité. La deuxième partie de la thèse développe un aspect technique soulevé lors de la première partie : le "calcul de minorants à partir de simplifications génériques". Cette partie comporte également trois chapitres qui décrivent plusieurs techniques de simplification.

Les six chapitres de ce rapport sont constitués de la manière suivante :

Chapitre 1 Les systèmes temps réel classiques sont d'abord présentés. Puis, nous expliquons les problèmes que pose l'introduction de fonctionnalités intelligentes dans ces systèmes. C'est principalement le temps de calcul non borné qui est en cause. Nous décrivons alors deux approches délibératives pour ordonnancer ces nouvelles tâches intelligentes. Ces approches délibèrent sur le compromis judicieux à déterminer pour répartir le temps disponible aux tâches, de façon à maximiser la qualité des résultats attendus. La première approche utilise le contrat temporel pour ajuster ses efforts. La seconde se veut plus souple et accepte d'être arrêtée à tout moment. Ces deux approches ont besoin de connaître à l'avance l'évolution de la qualité en fonction du domaine d'application ciblé.

Chapitre 2 Les Problèmes de Satisfaction de Contraintes Valuées (VCSP) offrent un cadre général pour la modélisation des nombreux problèmes d'optimisation discrète rencontrés en Intelligence Artificielle. Parmi eux, les problèmes additifs sont de loin les plus utilisés. Ce sont aussi les plus difficiles à résoudre. Nous montrons dans ce chapitre comment le passage de Problèmes de Satisfaction de Contraintes (CSP) à des problèmes d'optimisation est délicat. Nous décrivons les techniques développées pour les CSP et leurs évolutions pour traiter les VCSP. Il s'agit des méthodes de renforcement de cohérence, c'est à dire une sorte de pré-traitement ou raisonnement local pour faciliter la résolution proprement dite. Puis les méthodes de résolution, complètes (qui produisent l'optimum) et incomplètes (qui produisent des bons majorants plus rapidement que les méthodes complètes) sont décrites. Des schémas de coopération des deux méthodes, dont certains sont novateurs, ont aussi été abordés.

Chapitre 3 Ce chapitre apporte un jugement et des éléments de solution pour combiner des algorithmes d'optimisation sous contraintes dans des systèmes temps réel intelligents. La première partie aborde le cas d'une unique résolution en temps contraint. Elle montre l'utilité des méthodes incomplètes. Elle présente la notion de qualité en optimisation et donne plusieurs arguments contre la validité des profils de qualité. En effet, la nature même des problèmes combinatoires rend très délicat l'obtention de statistiques fiables. C'est pourquoi nous nous sommes orientés sur un contrôle dynamique de la qualité courante et des heuristiques simples pour déterminer son évolution. La deuxième partie prend le cas d'un ensemble de problèmes d'optimisation à résoudre en temps limité. Nous fixons des hypothèses sur le système temps réel et donnons des moyens de l'évaluer, à partir des notions d'utilité globale et de dominance d'un ordonnancement. Nous décrivons alors l'approche interruptible et délibérative retenue. Elle est composée d'algorithmes interruptibles réalisant un encadrement progressif de l'optimum et d'une série de critères heuristiques pour l'ordonnanceur. Cette approche est validée sur un ensemble de problèmes de la planification du satellite SPOT5.

Chapitre 4 Nous présentons un nouveau type de simplification, les simplifications par changement de la structure de valuation, combinées à une relaxation des contraintes par niveau d'importance. Nous les étudions d'abord de façon théorique. Nous décrivons des propriétés qui permettent de les classer suivant leur difficulté de résolution et de déduire des minorants. Nous étudions leur comportement par des expérimentations sur des problèmes aléatoires. Puis, nous proposons un enchaînement de ces simplifications pour obtenir un encadrement progressif de l'optimum. Cette technique est comparée à d'autres approches et validée sur des problèmes réels. Finalement, nous donnons plusieurs pistes pour améliorer l'enchaînement.

Chapitre 5 Ce chapitre commence par un aperçu des techniques de relaxation des contraintes. Ensuite nous en développons deux, en précisant les concepts utilisés, les algorithmes employés et leur efficacité sur des problèmes réels du CELAR ou de SPOT5. La première stratégie consiste à décomposer l'ensemble des variables en grappes, des paquets distincts de variables fortement connectées entre elles. Deux méthodes incomplètes sont décrites pour réaliser cette décomposition. La deuxième stratégie vise à réduire les domaines en supprimant des valeurs ayant un comportement similaire vis à vis des contraintes.

Chapitre 6 Les simplifications de l'objectif se traduisent par l'utilisation d'autres algorithmes que le classique *Depth First Branch and Bound*. La structure du problème n'est pas modifiée. Le nouvel objectif est de produire un minorant et non l'optimum. C'est le cas des méthodes de renforcement de cohérence étendues à l'optimisation. Nous étudions également une approche novatrice combinant *Programmation Linéaire en nombre 0/1* et relaxation arc-cohérente optimale. Le principe du *Branch and Bound* peut aussi être adapté à un cadre temps réel. De cette étude, nous proposons un algorithme original produisant efficacement des minorants.

Partie I

Approche délibérative pour l'optimisation sous contraintes

Chapitre 1

Systèmes temps réel intelligents

Nous présentons brièvement la problématique des systèmes temps réels. Puis nous examinons ses limites face à des tâches de raisonnement issues du domaine de l'Intelligence Artificielle (cf. section 1.1).

Nous décrivons alors les principales approches d'ordonnancement pour des systèmes temps réel intelligents. Nous présentons d'une part l'approche contractuelle (cf. section 1.2) et d'autre part l'approche interruptible (cf. section 1.3). Pour ces deux approches, nous décrivons les principes généraux (cf. sections 1.2.1 et 1.3.1) et nous présentons certains travaux majeurs.

Pour l'approche contractuelle, nous examinons plus en détail une de ses applications concrètes pour les systèmes experts temps réel (cf. section 1.2.2). De même, nous exposons plusieurs algorithmes pour la planification temps réel (cf. section 1.2.3), dont la problématique est assez proche de celle retenue pour la thèse, c'est à dire la résolution de problèmes d'optimisation sous contraintes, au sein d'un système temps réel.

Pour l'approche interruptible, nous décrivons une extension au modèle de tâche interruptible contenant une partie obligatoire et une partie interruptible (cf. section 1.3.2). Enfin, nous détaillons deux techniques d'ordonnancement ANYTIME, d'une part une compilation *off-line* d'algorithmes interruptibles (cf. section 1.3.3) et d'autre part, un contrôle dynamique de l'évolution de la qualité d'une tâche interruptible (cf. section 1.3.4).

1.1 Problématique des systèmes d'intelligence artificielle temps réel

1.1.1 Problèmes liés aux systèmes temps réel

Cette section a pour objectif de donner quelques notions très générales sur les systèmes temps réel. Ces systèmes sont souvent très complexes et durs à modéliser. Une étude des algorithmes pour le temps réel nécessite d'examiner des modèles simplifiés pour être capable de les analyser et de les manipuler correctement.

1.1.1.1 Système temps réel

Définition 1 ———————————————————————————
Système temps réel *[Gro88] :*

système informatique dont le fonctionnement est assujetti à l'évolution dynamique de l'état d'un environnement qui lui est connecté et dont il doit contrôler le comportement.

L'évolution dynamique de l'état de l'environnement est perçue par le système grâce à des capteurs qui lui fournissent un ensemble de mesures et une suite d'événements, dont la vitesse d'évolution dépend de la dynamique inhérente à l'environnement. Le fait d'être assujetti à l'évolution de l'environnement implique de répondre à des contraintes temporelles. La perception du temps pouvant prendre différentes formes, les **contraintes temporelles** sont de différents types [Dzi90] :

- si le temps est vu comme une suite d'événements, alors une contrainte peut être le "top" d'une horloge (par ex., l'événement 13h47 mais aussi l'événement "température supérieure à 83 degrés") ou bien une relation d'ordre entre plusieurs événements[1] ;
- si le temps est vu comme une durée, alors c'est une contrainte de temps de réponse, appelée échéance temporelle.

En fonction des capteurs, le système temps réel contrôle le comportement de l'environnement par l'exécution de tâches qui vont manipuler des actionneurs, capables de modifier l'environnement. Pour que ce contrôle soit correct, le système temps réel doit respecter plusieurs propriétés :

- la **réactivité** , capacité à réagir continûment aux événements de l'environnement ;
- la **ponctualité** , capacité à respecter et s'adapter aux échéances temporelles ;
- la **prédictibilité** , possibilité de prédire le comportement logique et temporel du système en fonction du comportement de l'environnement.

La réactivité implique que le système réagisse suffisamment rapidement pour que l'environnement soit correctement contrôlé. Le choix du niveau de contrôle attendu détermine si l'évolution de l'environnement est significative ou non et permet donc de filtrer certains événements jugés insignifiants.

D'un point de vue opérationnel, la notion d'échéance temporelle se définit par :

Définition 2 ---

Échéance temporelle *[XP93] :*
durée maximale entre le moment où un événement nécessite l'exécution d'une certaine tâche et le moment où l'exécution de cette tâche doit se terminer.

Un système temps réel est strict lorsque les échéances doivent obligatoirement être respectées. Si le non-respect de certaines échéances est envisageable, moyennant un coût de dépassement, alors il s'agit plutôt d'un système à temps contraint régi par des **échéances relatives** .

La prédictibilité permet de garantir un système temps réel comme étant conforme à son cahier des charges. C'est une propriété cruciale pour les systèmes temps réel, ainsi toute approche temps réel doit définir son niveau de déterminisme, quelles garanties elle offre, en particulier vis à vis des contraintes temporelles.

[1] Dans ce cas, la contrainte n'est pas spécifiquement temporelle mais correspond bien à un instant d'émission de l'événement. Cette généralisation du temps physique au temps logique est à la base des langages temps réel synchrones. Les instants discrets permettent de définir rigoureusement des concepts comme la simultanéité d'événements et la synchronisation.

1.1.1.2 Ordonnanceur

Le système assure le contrôle des tâches qui interagissent avec l'environnement, c'est la partie ordonnanceur du système.

Définition 3
Ordonnancement temps réel *[ABCC+93] :*
élaboration du planning d'exécution d'un ensemble fini de tâches soumises à des contraintes temporelles et à des contraintes de ressources (processeur, mémoire, ...).

Idéalement, les ordonnanceurs temps réel classiques visent les propriétés suivantes :

- respect des échéances strictes ;
- respect des relations de précédence entre tâches ;
- prévention des accès simultanés à des ressources partagées (mémoire commune, ports E/S) ;
- limitation des coûts de changement de contexte lors des préemptions[2] de tâches ;
- optimalité de l'ordonnancement (au sens où s'il existe une solution maximisant les objectifs de l'ordonnanceur, celui-ci la trouve).

Concrètement les objectifs de l'ordonnanceur peuvent être divers :

- trouver un ordonnancement qui respecte toutes les échéances strictes ;
- minimiser le nombre de processeurs nécessaire pour respecter toutes les échéances strictes (parallélisme, ordonnancement centralisé ou réparti ?) ;
- minimiser le retard des tâches à échéance relative.
- minimiser le nombre d'échéances ratées ;

1.1.1.3 Modèle de tâche

Plusieurs hypothèses simplificatrices sont nécessaires pour atteindre ces objectifs. Il s'agit essentiellement de bien modéliser les tâches. Plusieurs choix de modèle ont été traités dans la littérature [XP93] :

- périodique (tâche devant s'exécuter cycliquement) ou apériodique ;
- préemptible ou non ;
- indépendante ou en relation de précédence avec d'autres tâches ;
- indépendante ou avec partage de ressources communes ;
- à échéance stricte ou relative ;
- degré d'importance de la tâche.

[2] La préemption signifie que la tâche est interrompue en cours d'exécution, son état est sauvegardé puis restauré lorsque la tâche a de nouveau le processeur

En revanche, une hypothèse constamment retenue pour les algorithmes d'ordonnancement est la connaissance du **temps d'exécution au pire cas** de chaque tâche à ordonnancer [XP93].

Lorsque le modèle choisi devient proche de la réalité (multiprocesseur, exclusion mutuelle), le problème de l'ordonnancement est souvent NP-dur. Les stratégies d'ordonnancement sont alors sous-optimales, guidées par des heuristiques. La prédictibilité n'est alors garantie que de façon probabiliste.

1.1.1.4 Exemple d'ordonnanceur : *Earliest Deadline First*

Pour illustrer ces définitions, nous présentons brièvement l'algorithme d'ordonnancement *Earliest Deadline First* [LL73] pour machine monoprocesseur. Les hypothèses retenues sont des tâches périodiques[3], préemptibles, indépendantes, à échéance stricte sur requête (c-à-d que la tâche doit se terminer avant le début d'une nouvelle période). L'objectif de cet algorithme consiste à trouver un ordonnancement qui respecte toutes les échéances.

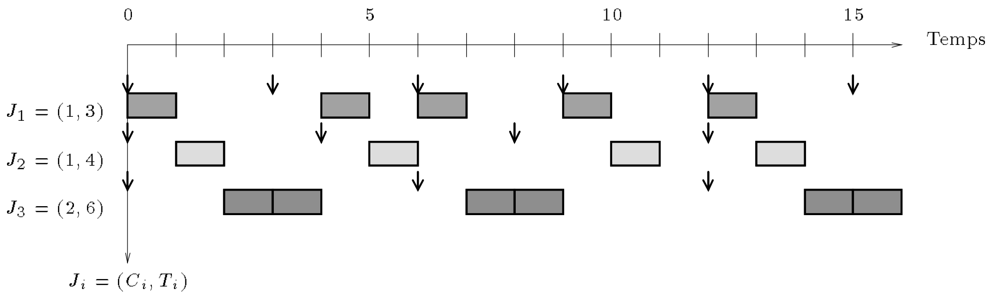


Fig. 1.1 - Exemple d'ordonnancement de tâches périodiques par priorité dynamique "Earliest Deadline First"

Dans l'exemple de la figure 1.1, un ordonnancement est donné pour trois tâches J_i avec des durées d'exécutions C_i et des périodes T_i (les flèches indiquent les échéances). Le temps est décomposé en quantum où une seule tâche peut s'exécuter à la fois. L'algorithme alloue chaque quantum à la tâche qui a sa date d'échéance temporelle la plus proche. La priorité relative des tâches évolue donc de façon dynamique. Nous dirons que l'ordre d'exécution des tâches et les instants de préemption sont déterminés par une **priorité dynamique** [4].

Cet algorithme est optimal pour les hypothèses retenues. Une condition nécessaire et suffisante permet de déterminer si l'objectif est atteignable ou pas, c-à-d si les échéances seront respectées :

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \quad , \quad n \text{ tâches à ordonnancer}$$

Cette condition permet de prédire le comportement temporel du système, donc concourt à définir la prédictibilité du système.

[3] La plupart des applications temps réel strictes gèrent essentiellement des tâches périodiques. Les tâches apériodiques sont relativement peu nombreuses et ont généralement des temps d'exécution très courts.

[4] Il existe d'autres politiques d'ordonnancement où la priorité est statique. Par exemple, le *Rate Monotonic* [LL73] fixe les priorités des tâches inversement proportionnelles à leur période.

Par ailleurs, cette condition n'est valable que si toutes les tâches et leurs paramètres sont connus à l'avance. Nous parlons dans ce cas d'**ordonnanceur statique** par opposition aux **ordonnanceurs dynamiques** qui acceptent de recevoir de nouvelles tâches à ordonnancer, durant l'exécution du système.

Dans le cas où l'ordonnancement est impossible, il faut alors soit redimensionner le système (amélioration des performances matérielles ou logicielles), soit réduire les fonctionnalités du système (simplifier ou retirer des tâches).

1.1.1.5 Bilan

En résumé, l'ordonnancement temps réel n'est aisément prédictible que dans des cas simples. L'accroissement de la complexité du modèle mathématique des tâches augmente vite la complexité des algorithmes d'ordonnancement et diminue leur niveau de prédictibilité[5]. D'après [XP93], seul l'ordonnanceur statique est capable d'offrir des garanties sur le respect de toutes les échéances strictes.

De plus, la sûreté de fonctionnent des systèmes oblige à prendre en compte d'autres informations, comme la redondance vis à vis des problèmes de tolérance aux fautes, qui rajoutent à la complexité de l'ordonnanceur.

Enfin, une des hypothèses simplificatrices est la connaissance des temps d'exécution des tâches. Nous allons voir que cette hypothèse est difficilement réaliste pour des systèmes temps réel intelligents.

1.1.2 Problèmes liés aux systèmes temps réel "intelligents"

De nombreuses applications propres à l'Intelligence Artificielle se tournent vers le domaine du temps réel pour prendre en compte de façon plus réaliste les contraintes temporelles. De même les systèmes temps réel se dirigent vers des applications de plus en plus complexes qui requièrent des comportements "intelligents" . Ces deux domaines de recherche convergent actuellement vers un nouveau sous-domaine, celui des **systèmes temps réel "intelligents"** ou systèmes temps réel IA [MHA+95]. Ces applications seront plus ou moins critiques vis à vis du respect des échéances strictes. En allant du plus critique au moins critique, nous pouvons citer comme application, un système d'aide à la décision tel le *Copilote électronique* [AF96] qui propose à un pilote des parades face à une ou plusieurs menace(s), ou bien le système de guidage d'un robot, ou encore un outil intelligent d'exploration du *World Wide Web*. L'apport de facultés de raisonnement soulève de nouvelles difficultés pour les systèmes temps réel.

1.1.2.1 Temps d'exécution variable ou incompatible

Premièrement, et principalement, le temps d'exécution d'une tâche de raisonnement est difficile à estimer. Il y a deux raisons à cela. D'une part, le temps d'exécution varie suivant les données en entrée du problème. C'est aussi le cas pour une tâche classique. D'autre part, le fait de raisonner implique le plus souvent de parcourir un espace de solutions potentielles jusqu'à trouver une solution réalisable et éventuellement optimale suivant certains critères. En fonction du niveau de connaissance du domaine à traiter, il est possible de se diriger plus ou moins vite vers la bonne solution. Ce niveau de connaissance est traduit algorithmiquement par

[5] Il est intéressant de noter que le problème de l'ordonnancement temps réel statique est lui-même un bon candidat pour les techniques d'optimisation qui vont être présentées par la suite.

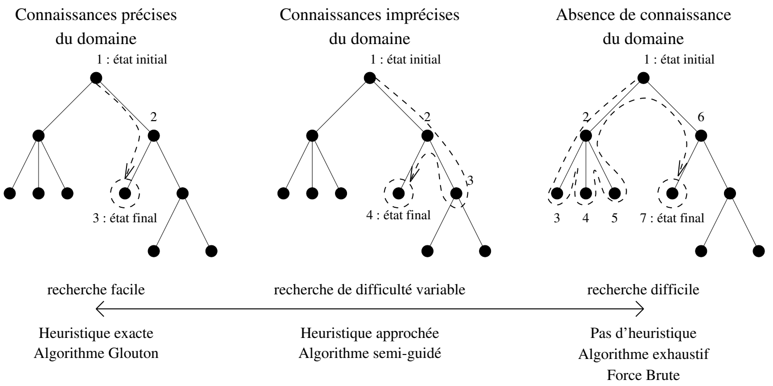


Fig. 1.2 - échelle de difficulté de la résolution par rapport aux connaissances du domaine

des heuristiques qui guident la recherche de solution. La figure 1.2 issue de [SP94] résume ce problème en montrant le lien entre la qualité des heuristiques et la difficulté de la recherche. Moins les heuristiques sont pertinentes, plus le risque de parcourir une grande partie de l'espace de recherche augmente, et donc plus la **variance sur le temps d'exécution** est forte.

De plus, la taille de l'espace des solutions potentielles évolue souvent de manière exponentielle en fonction des données du problème. Cela vient du fait que les problèmes de raisonnement appartiennent à la classe des **problèmes** NP . Cela signifie, que même pour un algorithme guidé par de bonnes heuristiques, le temps d'exécution au pire cas est exponentiel. L'application de l'hypothèse sur les temps de calcul bornés, à respecter par les systèmes temps réel classiques, devient inacceptable car elle obligerait à sur-dimensionner le système, de manière prohibitive.

1.1.2.2 Environnement fortement imprédictible

Deuxièmement, les systèmes temps réel IA sont confrontés au problème du fort indéterminisme de l'environnement qu'ils ont à contrôler. Cela signifie que les contraintes temporelles ne sont connues que dynamiquement. Le système doit alors s'**adapter** , en activant diverses tâches apériodiques de raisonnement. Cela implique un ordonnancement dynamique où il est plus difficile de garantir le respect des échéances temporelles [XP93].

Dans certains cas, l'indéterminisme est tel que la durée des échéances temporelles est difficilement prédictible. Par exemple, l'arrivée d'un événement imprévu peut raccourcir l'échéance d'une tâche de raisonnement et donc remettre en cause l'ordonnancement précédent.

1.1.2.3 Bilan

Pour résumer, les systèmes temps réel IA doivent faire face à de nouvelles contraintes. Le terme “bounded rationality” décrit la nécessité de prendre en compte les contraintes de vitesse d'exécution, de taille mémoire. Le terme “bounded reactivity” décrit la nécessité de prendre en compte les limites physiques des senseurs et des actionneurs. Il s'agit donc de trouver de bons compromis pour s'adapter à ces nouvelles contraintes. Ces compromis sont à décider en fonction d'objectifs principaux, exhibés dans [MHA+95], pour les systèmes temps réel intelligents :

- prendre en compte les événements synchrones ou asynchrones de manière prédictible, avec des garanties sur les temps de réponse ;

- dégrader ses performances de manière progressive, en fonction de la charge du système (modification des objectifs, des méthodes en cours d'exécution) mais aussi des pannes (prendre en compte les tâches de recouvrement de panne) ;
- concentrer les ressources sur les événements les plus critiques ;
- s'accommoder de données incertaines ou incomplètes ;
- rester opérationnel durant une longue période de temps (autonomie du système) ;
- s'interfacer avec l'environnement extérieur à l'aide de senseurs et d'actionneurs.

Nous examinons dans la section suivante les moyens pris pour respecter ces objectifs.

1.1.3 Éléments de solution pour une approche délibérative

1.1.3.1 Les principales approches

Il s'agit de forcer les tâches de raisonnement à respecter leurs échéances temporelles. Pour cela, il faut soit réduire le temps d'exécution des tâches et sa variance, soit rendre les tâches incrémentales et interruptibles. C'est possible s'il y a la propriété qu'**une tâche de raisonnement puisse fournir une solution dégradée en un temps plus court et que cette solution soit acceptable vis à vis du système** . Dans cette optique, nous distinguons principalement deux grandes approches :

- approche **réactive** qui suppose l'existence d'actions réflexes, enregistrées dans une table de correspondance *situation* $\Rightarrow$ *actions* (ou un arbre de décision) obtenue par compilation d'un ensemble de connaissances sous forme de règles ; voir par exemple, le système KHEOPS [GP89, DGP94], l'algorithme RETE [For82, GFN89] qui effectue une étape supplémentaire d'unification des données avec la partie condition des règles ; l'avantage principal de cette approche est bien sûr sa rapidité d'exécution ; l'inconvénient majeur est que le système voit son adaptabilité limitée à moins d'avoir une table de correspondance gigantesque ; l'approche devient réellement intéressante lorsqu'elle intervient en complément de techniques de raisonnement plus évoluées, utiles si le temps alloué à la tâche est grand [DSBL93] ;
- approche **délibérative** qui délibère sur le bon compromis entre un temps Δt accordé à l'exécution d'une tâche de raisonnement et le gain d'utiliser son résultat partiel après avoir attendu ce temps Δt ; nous choisissons de séparer la présentation de cette approche en deux sous parties :
  - approche **contractuelle** (cf. section 1.2) qui utilise des techniques d'approximation et de choix des méthodes de résolution pour s'adapter aux échéances temporelles, cf. sections 1.2.1 et 1.2.2, ou qui borne le mécanisme de recherche en fonction du temps disponible, cf. section 1.2.3 ;
  - approche **interruptible** (cf. section 1.3) qui suppose l'existence d'algorithmes produisant des résultats intermédiaires de façon continue de manière à pouvoir interrompre l'algorithme n'importe quand et récupérer un résultat cohérent.

Nous choisissons dans cette thèse l'approche délibérative comme élément de solution à la problématique des systèmes temps réel intelligents.

A part le cas trivial d'ordonnancement qui correspond à l'exécution d'une seule tâche de raisonnement jusqu'à la fin d'une échéance stricte, lorsqu'il y a plusieurs tâches ou si l'échéance est relative, l'ordonnancement est un problème d'optimisation délicat. Lorsqu'il s'agit d'ordonnancer plusieurs tâches en concurrence pour l'utilisation des ressources matérielles (CPU,mémoire,I/O) avec l'objectif de respecter une échéance stricte globale, l'ordonnancement doit décider du temps optimal alloué à chacune des tâches. Lorsqu'il s'agit d'ordonnancer une tâche soumise à une échéance relative, l'ordonnancement doit déterminer le moment optimal d'arrêt de la tâche, instant où le gain de continuer à exécuter la tâche devient inférieur au coût de dépassement de l'échéance.

1.1.3.2 Notion de qualité

L'optimisation ne peut se faire que suivant des critères bien définis. Pour cela, il faut définir la notion de qualité d'un résultat partiel fourni par une tâche de raisonnement. La qualité est une valeur normée dans l'intervalle $[0, 1]$. La valeur 1 correspond à un résultat exact obtenu par une exécution complète de la tâche. Il y a principalement trois grands axes pour mesurer la qualité [Zil96] :

- **exactitude** : distance par rapport au résultat exact, ou à une estimation du résultat exact ; par exemple, en optimisation, c'est la différence entre la valuation du résultat partiel et la valuation optimale ; en programmation fonctionnelle, pour un type construit, c'est le nombre de champs entièrement réduits [CB95] ;
- **spécificité** : degré de détail du résultat partiel ; par exemple, pour un système expert effectuant un raisonnement progressif [Mou93], c'est la granularité des informations (données et règles) considérées par le niveau courant de raisonnement du système expert ;
- **certitude** : degré de certitude qu'un résultat partiel soit exact ; par exemple, pour la preuve de théorème, [HK95] calcule la probabilité qu'une formule booléenne soit satisfiable en fonction du rapport entre l'espace de recherche parcouru et l'espace total.

Mesurer la qualité est une tâche difficile car elle doit permettre d'évaluer l'utilité d'un résultat partiel pour le système. Il faut donc exhiber les bons critères de qualité qui sont a priori différents pour chaque algorithme et pour chaque utilisation qu'il en est faite. Pour mettre en évidence ces problèmes, nous reprenons l'étude réalisée par Horvitz [Hor90] sur des algorithmes de tri , puis reprise dans [ABdG95a].

1.1.3.3 Exemple de critères de qualité : les algorithmes de tri

Horvitz étudie différents critères de qualité pour évaluer une liste de N éléments partiellement triés dans l'ordre croissant. Ces critères reposent sur l'exactitude du résultat partiel par rapport au résultat final, obtenu à l'instant $t_{complet}$; ils s'appellent :

- le désordre moyen représente la distance moyenne entre la position courante d'un élément et sa position finale ; il est noté Q_{DM} tel que :

  $$Q_{DM}(t) = 1 - \frac{\sum_{i=1}^{N} |p_{x_i}(t) - p_{x_i}(t_{complet})|}{N^2}$$

  avec x_i, le $i$ème élément qui se trouve à la position $p_{x_i}(t)$ à l'instant t ;
- le pourcentage de la liste triée par le bas, correspondant au nombre d'éléments contigus en partant du début de la liste qui occupent leur place définitive.

Les algorithmes de tri [CLR94] retenus pour les expérimentations sont le tri rapide (*Quicksort*), le tri de Shell (*Shellsort*) et le tri par tas (*Heapsort*, légèrement modifié pour trier par ordre croissant). Pour avoir une représentation graphique du travail réalisé par ces algorithmes, la liste initiale est composée d'une permutation aléatoire de N éléments de valeurs distinctes allant de 1 à N [6]. Les figures 1.3 1.4 et 1.5 représentent la valeur des éléments (en ordonnée) en fonction de leur position (en abscisse) dans la liste courante, à l'instant $t = 0.06$ sec pour $N = 10^4$ éléments. À la fin de l'exécution de chaque tri, la représentation graphique de la liste complètement triée est une diagonale.

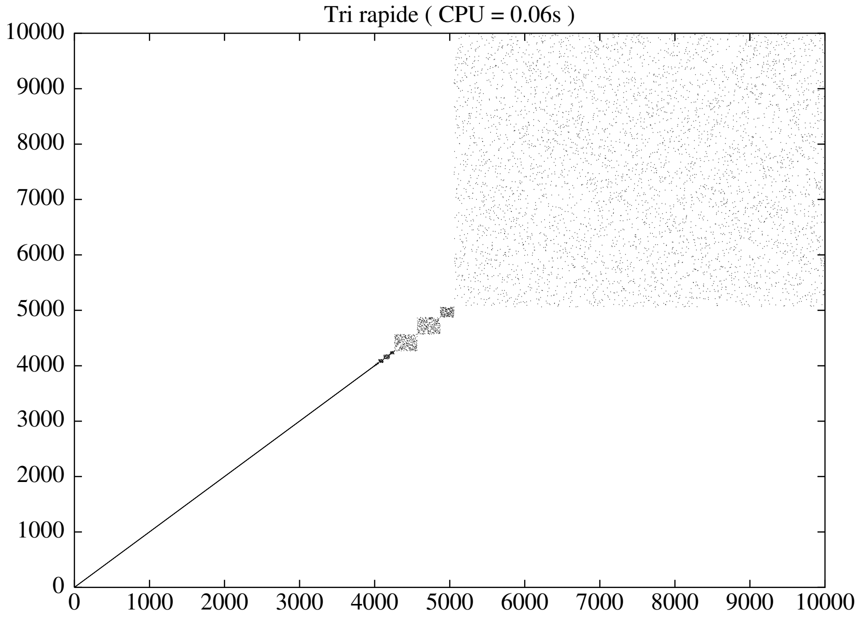


Fig. 1.3 - Tri rapide sur 10^4 éléments, après 0.06 sec de CPU

L'évolution de la qualité, suivant les deux critères retenus, en fonction du temps de calcul, est représentée par les courbes 1.6 et 1.7, appelées profils de qualité. Pour avoir des durées significatives sur ces deux figures, les tris sont effectués sur 10^5 éléments, permutés aléatoirement, au lieu de 10^4. Pour une permutation aléatoire, le tri de Shell, bien que se terminant presque deux fois plus tard que le tri rapide, a au cours du temps le meilleur profil de qualité pour la réduction du désordre moyen. En revanche, c'est le tri rapide qui a le meilleur profil pour le pourcentage d'éléments complètement triés.

Dans le pire cas, seule la complexité du tri par tas est en $O(N \log N)$, celle du tri rapide et du tri de Shell sont en $O(N^2)$ [Wei92]. Par exemple, pour une liste initialement triée dans l'ordre inverse (décroissant) de 10^4 éléments, le tri rapide est environ vingt fois plus lent que les deux autres tris. Ainsi, nous voyons qu'il est nécessaire d'analyser les profils de qualités en fonction du type des données en entrée. Si ces données sont quelconques, le tri par tas semble le choix le plus sûr, d'un point de vue robustesse face aux entrées (faible variation par rapport à la configuration des données, évolution progressive suivant les deux critères de qualité).

[6] Cette condition sur la valeur des éléments peut être utilisée pour accélérer le tri (algorithme de complexité linéaire). L'objectif étant d'analyser des algorithmes généraux qui ne font aucune hypothèse sur les éléments à trier, cette condition n'est pas utilisée pour accélérer le tri.

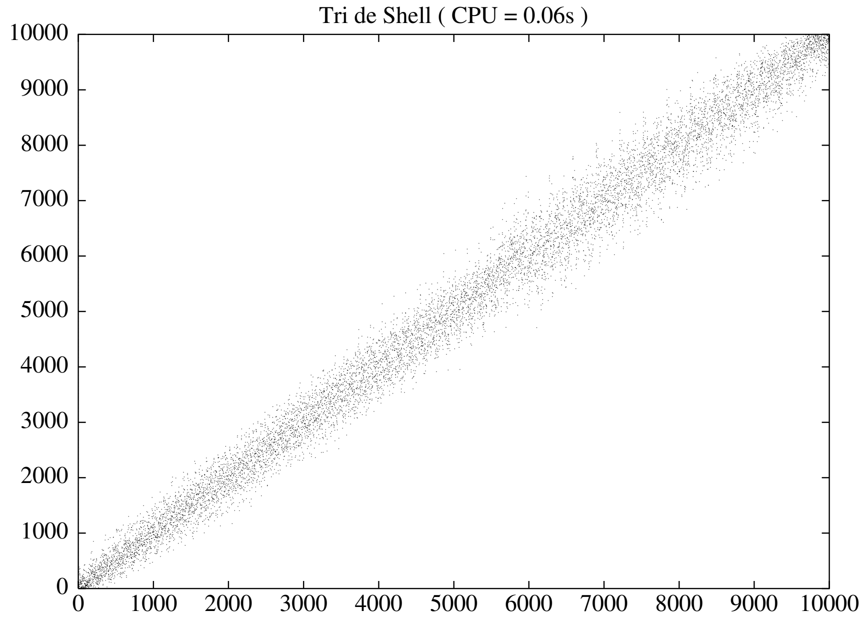


Fig. 1.4 - Tri de Shell sur 10^4 éléments, après 0.06 sec de CPU

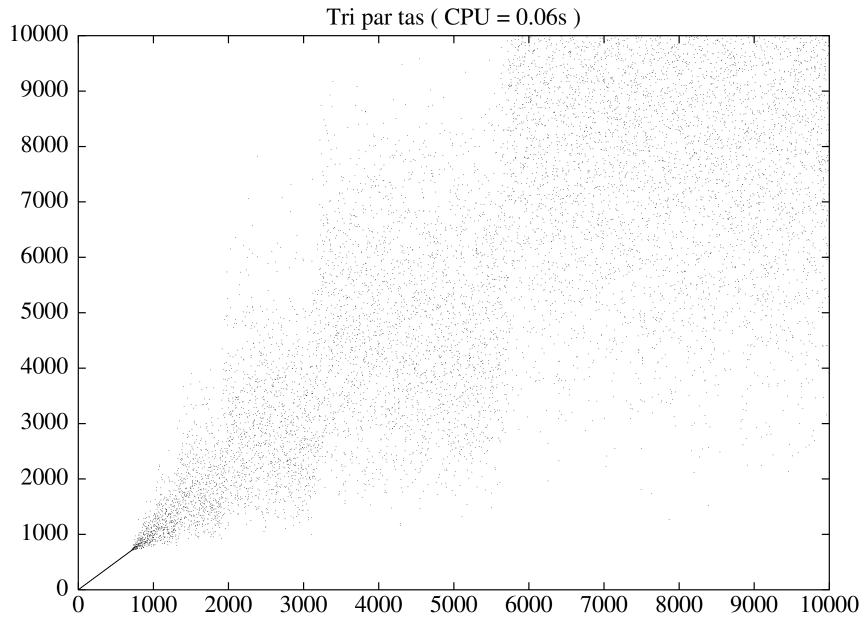


Fig. 1.5 - Tri par tas sur 10^4 éléments, après 0.06 sec de CPU

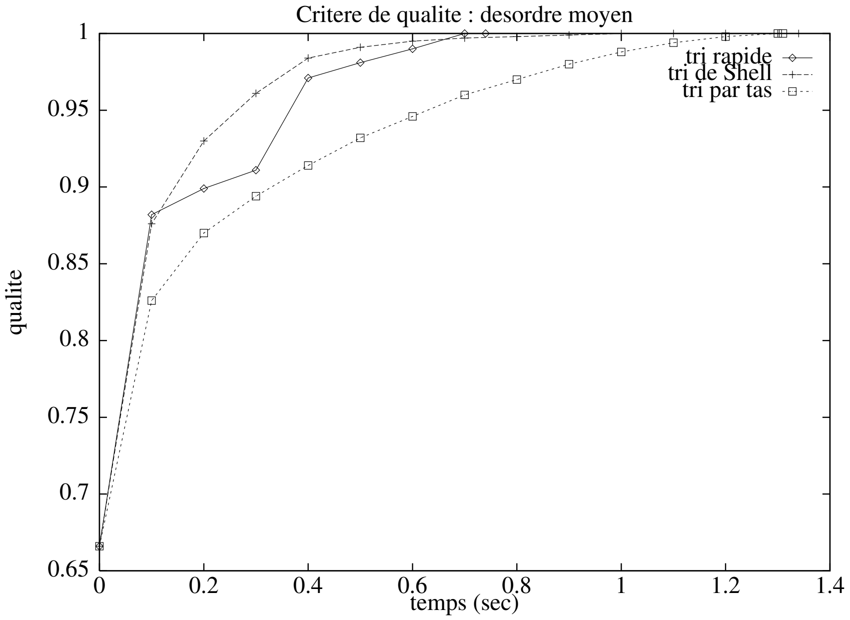


Fig. 1.6 - Profil de qualité (désordre moyen) pour 10^5 éléments avec une permutation initiale aléatoire

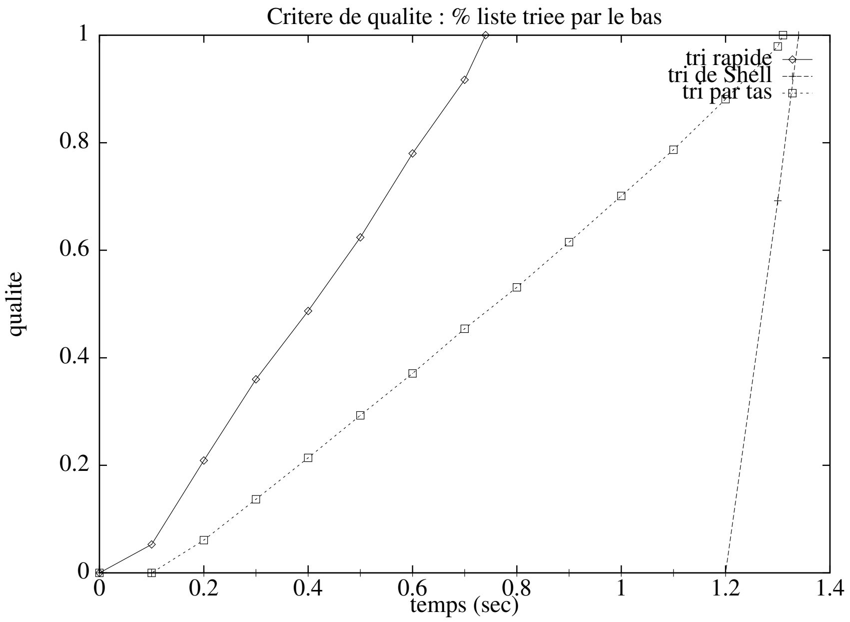


Fig. 1.7 - Profil de qualité (% de la liste trié par le bas) pour 10^5 éléments avec une permutation initiale aléatoire

1.1.3.4 Notion d'utilité

Un profil de qualité est une mesure objective des performances d'un algorithme suivant un certain critère et en fonction d'un certain type de donnée en entrée. La qualité est liée à une notion plus subjective d'**utilité intrinsèque** . En effet, l'utilité intrinsèque d'un résultat partiel représente le bénéfice pour le système d'exploiter ce résultat, hors contraintes temporelles. Par exemple, dans le cas du tri, si l'objectif est d'abord d'avoir une liste approximativement triée, le tri de Shell est l'algorithme qui a la plus forte utilité intrinsèque. En revanche, si nous sommes aussi intéressé par la connaissance des premiers éléments de la liste, alors nous effectuons une combinaison des deux critères de qualité, et l'algorithme du tri rapide peut devenir le meilleur choix[7]. Ainsi l'utilité intrinsèque $U_I(t)$ d'un résultat produit après un temps t de calcul s'écrit $U_I(t) = f(Q_c(t))$ avec f, une fonction combinant diverses qualités Q_c de critère c.

La prise en compte des contraintes temporelles vient atténuer l'utilité intrinsèque pour aboutir à la notion la plus générale d'utilité[8]. Par exemple, si le système est soumis à une échéance relative à $t = 0$, impliquant un **coût temporel** $CT = g(t)$, alors l'utilité, notée $U(t)$, d'un résultat partiel vaut à l'instant t :

$$U(t) = U_I(t) - CT(t) = f(Q_c(t)) - g(t)$$

1.1.3.5 Bilan

Dans les sections suivantes, nous présentons deux approches pour ordonnancer des tâches de raisonnement. L'approche contractuelle utilise la connaissance du temps d'exécution avant échéance, le contrat, pour sélectionner la stratégie de raisonnement utilisant au mieux ce temps. Ce sont donc les tâches qui s'adaptent au contrat. L'approche interruptible considère que la tâche produit continûment des résultats partiels de qualité croissante. Elle ordonnance alors les tâches en fonction d'une prévision sur les profils de qualité. Nous verrons que la frontière entre les approches contractuelle et interruptible est parfois floue.

Pour prendre en compte l'indéterminisme des temps d'exécution des tâches et de l'environnement extérieur, l'ordonnanceur doit être adaptatif. Les politiques d'ordonnancement présentées sont toutes dynamiques. De plus le modèle de tâche prend en compte plus ou moins de caractéristiques : connaissances sur l'évolution de la qualité, périodique ou apériodique, indépendante ou en relation de précédence avec d'autre tâches, à échéances temporelles strictes ou relatives ...

Une première conclusion sur les solutions apportées aux systèmes temps réel intelligents est que le sur-dimensionnement du système dû aux temps d'exécution élevés et à leurs grandes variances est évité grâce au compromis qualité/temps d'exécution et que le problème de la prédictibilité temporelle est reporté sur le problème de la garantie d'obtention "dans les temps" d'un résultat partiel de qualité acceptable. La notion subjective de solution satisfaisante ou acceptable permet de définir un **seuil d'acceptabilité** de la qualité. Généralement, l'ordonnancement sera "classique" pour atteindre ce seuil et ensuite optimal ou sous optimal pour maximiser l'utilité globale du système.

[7] Ce choix dépend des pondérations appliquées aux critères de qualité et indiquant leurs importances respectives. Pris comme exemple dans la section 1.3.3, nous examinerons plus en détail une combinaison d'algorithmes de tri donnant une meilleure utilité intrinsèque qu'un seul algorithme de tri.

[8] Venant de la théorie de la décision [vNM47, Fis81, Hor90], l'utilité définit un ordre total entre plusieurs éventualités de choix d'action. Dans notre cas, il y a deux éventualités : soit exploiter le résultat partiel disponible, soit poursuivre le traitement pour obtenir un résultat de meilleure qualité.

1.2 Politiques d'ordonnancement de tâches contractuelles

Dans l'approche contractuelle, les résultats partiels peuvent n'être fournis qu'à la fin du contrat. C'est le cas pour la politique d'ordonnancement *Design-to-time* (cf. section 1.2.1) et dans une certaine mesure la politique adoptée pour le *Raisonnement Progressif* (cf. section 1.2.2). En revanche, les algorithmes de planification temps réel décrits dans la section 1.2.3 fournissent des résultats intermédiaires et déterminent la profondeur de la recherche en fonction du contrat.

1.2.1 *Design-to-time*

1.2.1.1 Approximations et méthodes multiples

Pour résoudre un problème dans un temps limité, l'approche *Design-to-time* [Gar96, GL93a, GL94] suppose connu un ensemble de méthodes et d'approximations réalisant toutes un compromis { qualité du résultat / durée d'exécution au pire cas } différent. Cette connaissance des compromis doit être suffisamment précise (majorant des valeurs exactes) pour que l'ordonnancement soit prédictible. Le fait d'avoir un ensemble de méthodes pour résoudre un problème permet de choisir, à l'exécution, parmi plusieurs algorithmes, celui qui est le plus apte à résoudre une instance particulière du problème, en fonction de son compromis {qualité/durée}.

Les approximations interviennent sur les trois axes mesurant la qualité : sur l'exactitude, en ignorant des caractéristiques du problème qui sont redondantes ou peu probables ; sur la spécificité, en supprimant des données jugées peu importantes, ou en regroupant plusieurs données en une seule ; sur la certitude, en simplifiant les fonctions qui caractérisent le problème.

[SP94] présentent une vue synthétique des techniques générales pour réduire le temps d'exécution, au moins de façon pratique, d'un problème de planification[9] :

- élagage : supprimer des portions de l'espace de recherche pour lesquelles il est possible de déterminer qu'elles ne contiennent pas d'état but ; cela peut être fait de manière statique, en partitionnant le problème en plusieurs sous-problèmes (approche *Diviser pour régner*) ; ou bien de manière dynamique, en utilisant les connaissances obtenues à l'exécution pour effectuer des coupes dans l'espace de recherche ;

- ordre : ordonner la recherche par des heuristiques pour examiner en premier les zones les plus prometteuses ; par exemple, la stratégie de recherche du meilleur d'abord dans l'algorithme A^* [AS94] ;

- approximation : approximer les données par des regroupements ou réductions ; au niveau fonctionnel, faire des simplifications sur la modélisation du problème ;

- portée : limiter la profondeur de son raisonnement (en temps et en espace) avant de décider du choix du prochain opérateur ; par exemple, l'algorithme *Iterative Deepening* [Kor85] effectue un raisonnement de portée croissante.

Parmi ces techniques, seul l'élagage permet de réduire le temps d'exécution dans le pire des cas tout en préservant l'optimalité du raisonnement, c'est à dire la possibilité d'atteindre un état but.

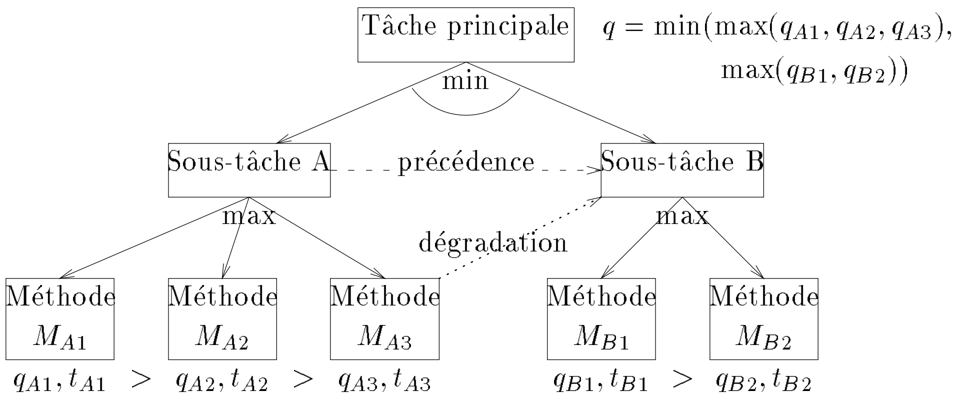


Fig. 1.8 - Description d'un groupement de tâches dans le modèle TAEMS

1.2.1.2 Le modèle *TAEMS*

Le terme *Design-to-time* signifie construire dans le temps imparti, à l'exécution, une solution de qualité maximale en utilisant toutes les ressources disponibles. Un modèle de tâche, appelé *TAEMS* [DL93a, DL93b] (Analyse de Tâche, Modélisation de l'Environnement et Simulation) permet de décomposer chaque tâche principale en une hiérarchie de sous tâches (graphe acyclique dirigé), éventuellement en relation de dépendance entre elles. Un exemple est donné figure 1.8 où deux sous-tâches A et B doivent être exécutées successivement. Les feuilles de la hiérarchie sont les différentes méthodes exécutables avec pour chacune l'information {qualité,durée} . Des opérateurs min,max (ainsi que $\sum$,moyenne,etc. dans le cas général) , inscrits en dessous des noeuds du graphe sur la figure, indiquent comment la qualité est synthétisée pour obtenir la qualité globale q de la tâche principale. Dans l'exemple, la sous-tâche A peut être réalisée par trois méthodes M_{A1}, M_{A2}, M_{A3} dont l'algorithme et/ou les approximations diffèrent. Des relations supplémentaires inter-tâches permettent de décrire des modifications sur la qualité suivant les méthodes choisies. Par exemple, l'utilisation d'une approximation très forte M_{A3} réduit la qualité de la solution de la sous-tâche B car celle-ci utilise en entrée le résultat de la sous-tâche A (flèche annotée "dégradation").

L'ordonnancement de ce modèle de tâche consiste à sélectionner tout d'abord les méthodes qui procurent les solutions de qualité maximale. Puis, si la durée complète d'exécution des tâches dépasse l'échéance stricte, l'heuristique choisit, parmi les méthodes des sous-tâches, celle qui réduit le moins possible la qualité par rapport au temps gagné.

Théorème 1

L'ordonnancement est optimal lorsque le graphe est un arbre, qu'il n'y a que des opérateurs min *et* max *pour combiner la qualité, que les relations de précédence sont acycliques et que chaque relation influant sur la qualité ("dégradation" ou "amélioration") soit associée à une relation de précédence de façon cohérente [GL93b].*

[9] La planification consiste à trouver une séquence d'opérateur permettant d'aller d'un état initial à un état but [MH95].

1.2.1.3 Extensions au modèle

Il est possible de prendre en compte dans ce modèle, des méthodes de résolution incrémentales en discrétisant leur profil de qualité [GL95a] ; c'est donc une passerelle pour utiliser des tâches interruptibles (cf. section 1.3.1).

Pour améliorer le comportement du système face à un environnement incertain, un contrôle dynamique des tâches est ajouté. Il permet d'augmenter le niveau d'approximation d'une tâche si l'échéance devient trop courte. De même, les résultats intermédiaires obtenus en interne par une tâche jugée trop lente, sont réutilisés pour démarrer une approximation plus forte de cette tâche. Enfin, des méthodes d'urgence , permettant d'atteindre rapidement une solution acceptable, sont appliquées dans le cas où l'ordonnanceur détermine que les échéances ne peuvent pas être respectées, ou bien dans le cas où le contrôle détecte qu'une tâche ne respecte pas sa durée d'exécution.

Le point faible de l'approche *Design-to-time* est l'obligation de connaître précisément les durées d'exécution au pire cas des différentes méthodes de résolution. Une extension au modèle *TAEMS* utilisant une connaissance incertaine des compromis {qualité/durée} offerts par les tâches, sous forme d'une distribution de probabilité des durées et des qualités, est étudiée dans [GL95b, Gar96].

1.2.2 Raisonnement progressif

1.2.2.1 Le modèle *GREAT*

Le modèle de raisonnement progressif *GREAT* (Guarantee REAsoning Time) [MZ95, Mou93, MCH92] s'adresse aux systèmes à base de connaissance dans un contexte temps réel. Il vise à corriger deux défauts de l'approche *Design-to-time*. D'une part le temps de calcul déjà effectué par une tâche est perdu lorsque celle-ci voit son échéance raccourcie par un nouvel événement et qu'elle est alors approximée à un niveau d'approximation plus fort[10]. D'autre part si l'échéance est prolongée, la tâche n'en profite pas pour améliorer sa qualité.

La solution proposée consiste à décomposer le raisonnement en plusieurs niveaux correspondant à des niveaux d'approximation décroissants pour obtenir, après le calcul de chaque niveau, des résultats intermédiaires utilisables. L'amélioration par rapport à l'approche *Design-to-time* est que les niveaux sont créés dynamiquement en fonction du problème et surtout sont directement reliés entre eux de telle façon que le travail d'un niveau i constitue le point de départ du niveau suivant $i+1$. Ainsi la solution s'affine progressivement suivant le temps d'exécution disponible[11].

La base de connaissance du modèle *GREAT* contient des données $D = \{\{d_1, d_2, \ldots, d_n\}^*\}$ sous forme d'objets formés par une liste d'*attributs*. À chaque attribut d_i correspond un triplet {*valeur, granularité, degré de certitude*}. La valeur contient l'information proprement dite. La *granularité* indique le degré de précision ou d'approximation de la valeur de l'attribut. C'est une relation d'ordre partielle car les données de même granularité ne sont pas forcément comparables. La granularité permet de définir de manière sémantique les différents niveaux de raisonnement. Elle est donc établie par expertise. Enfin le degré de certitude permet de donner un niveau de

[10] Ce propos est à nuancer puisque [GL93a] considère que des résultats intermédiaires peuvent être échangés entre les méthodes, cf. section 1.2.1

[11] Cette solution est appelée *depth-first deepening* car elle n'exécute qu'une fois le raisonnement à un niveau donné ; à l'inverse *depth-first iterative-deepening* recommence à chaque fois le raisonnement au premier niveau jusqu'à une profondeur donnée [CB94a].

confiance dans la valeur de l'attribut.

En plus des données, il y a une *base de règles* $K = \{r_1, r_2, \ldots, r_m\}$ telle que

$$r_i : \text{ si } f(d_i, \ldots, d_k) \text{ alors } g(d_j, \ldots, d_l)$$

, avec la partie condition $f : D^{k-i+1} \mapsto \{vrai, faux\}$ et la partie action $g : D^{l-j+1} \mapsto D^p$. La granularité d'une règle est définie comme étant la granularité maximale de la partie condition. Une propriété importante des règles est de n'avoir en résultat de sa partie action que des attributs de granularité supérieure ou égale à la granularité de la règle. “Cette caractéristique assure une expansion du raisonnement qui consiste à utiliser des données de plus en plus fines” [Mou93].

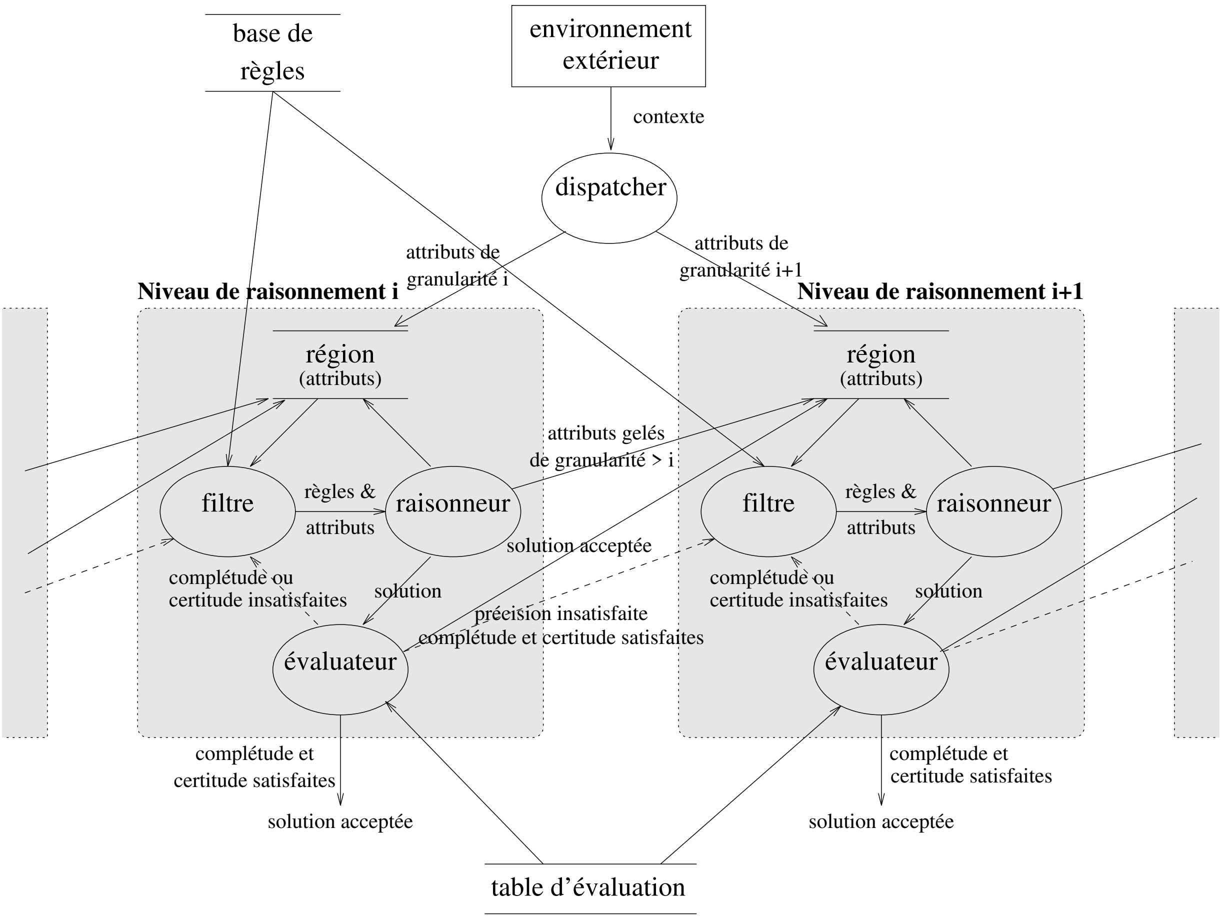


Fig. 1.9 - Modèle *GREAT*

La figure 1.9 [Mou93] décrit le fonctionnement du modèle *GREAT*. En fonction de l'événement déclenchant la tâche, un certain *contexte* est sélectionné, c'est à dire que le système se focalise sur un sous-ensemble des objets de D. à partir de ce contexte, les différentes granularités des attributs des objets retenus vont déterminer les *régions* utilisées par les niveaux de raisonnement (rôle du *dispatcher*). Chaque niveau consiste à n'utiliser que des attributs de même granularité ainsi que uniquement des règles dont la partie condition porte sur des attributs de granularité inférieure ou égale à la granularité associée au niveau (rôle du *filtre*). Au sein d'un même niveau, les données sont aussi filtrées suivant leur importance (par exemple, leur degré de certitude).

La qualité d'une solution est définie suivant trois critères, proches de ceux présentés dans la section 1.1.3 :

- *complétude* : liste d'attributs que doit contenir la solution ;
- *précision* : granularité minimale à atteindre pour chacun de ces attributs ;
- *certitude* : degré de certitude minimal à atteindre pour chacun de ces attributs.

La difficulté n'est pas de calculer ces critères mais plutôt de les normer, ce qui revient ici à connaître la liste des attributs composant la solution, et surtout les seuils d'acceptabilité en précision et en certitude pour chaque niveau de raisonnement. Cette connaissance est établie par une phase d'apprentissage. Elle est sauvegardée dans une *table d'évaluation* pour chaque contexte et chaque niveau de raisonnement. **L'originalité de l'approche *GREAT* est d'utiliser ces critères de qualité pour conduire la progression du raisonnement : la qualité instantanée contrôle la quantité d'information à traiter, c'est à dire le degré d'approximation.**

Le raisonnement, un moteur d'inférence, active toutes les règles sélectionnables. Les attributs en partie action des règles sont alors ajoutés, modifiés ou supprimés (rôle du *raisonneur*). Si un attribut prend une valeur de granularité supérieure au niveau courant, il est alors gelé, c'est à dire que seul le niveau de même granularité pourra s'en servir. De façon cyclique, l'évaluateur teste les trois critères de qualité de la solution. Si les qualités sur la complétude et la certitude sont insuffisantes, un nouveau cycle de raisonnement est lancé, éventuellement en rajoutant des données moins importantes (rôle du *filtre*). En revanche si ces qualités sont satisfaisantes, la solution est acceptée. Elle fait partie des résultats intermédiaires qui concourent à l'évolution progressive de la qualité fournie par le système[12]. Si la précision de la solution est insatisfaisante, l'évaluateur déclenche le niveau de raisonnement suivant.

1.2.2.2 Le modèle *Multi-GREAT*

Des algorithmes d'ordonnancement pour le modèle *Multi-GREAT*, composé de plusieurs tâches *GREAT* à exécuter sur un seul processeur, sont étudiés dans [Mou93]. Nous présentons dans le paragraphe suivant un des algorithmes retenu dans le cadre du projet européen *REAKT* [LCH92]. C'est un ordonnanceur dynamique à priorités dynamiques ; à chaque tâche est associée une échéance, ainsi qu'une valeur indiquant l'importance relative de la tâche par rapport aux autres tâches.

Les durées d'exécution des tâches sont difficiles à estimer. Néanmoins nous supposons connu une estimation de la durée de chaque niveau de raisonnement. Pendant l'exécution du système, l'ordonnancement courant, une séquence S ordonnée de tâches dont la date de début et de fin sont connues, est mis à jour par l'arrivée d'une nouvelle tâche E dont l'échéance, l'importance et la durée de ses niveaux sont connues. L'ordonnanceur privilégie la réactivité en effectuant un ordonnancement en profondeur d'abord de la tâche la plus urgente. Concrètement, l'ordonnancement *Earliest Deadline First* (cf. section 1.1.1) est employé pour ordonnancer les

[12] Il est donc possible de générer des profils de qualité lors d'une phase d'apprentissage. Ces profils sont alors utilisés suivant les techniques interruptibles décrites dans la section 1.3. Mais dans cette section, nous nous intéressons seulement aux politiques d'ordonnancement contractuelles qui utilisent au moins une estimation sur la durée des tâches, ici sur la durée des niveaux de raisonnement.

tâches par ordre croissant de leurs échéances. L'algorithme tente d'insérer tous les niveaux de la nouvelle tâche E dans S suivant la priorité de son échéance. Les dates de début et fin des tâches après le point d'insertion sont décalées en fonction de la durée complète de E. Si toutes les échéances sont respectées, alors cet ordonnancement est optimal. Sinon, les tâches de plus petite importance sont approximées[13], c'est à dire que l'ordonnanceur supprime leurs niveaux les plus profonds jusqu'à leur premier niveau exclu. Cela permet de garantir l'exécution du premier niveau de chaque tâche. Si ces approximations ne suffisent pas, alors les tâches les moins prioritaires sont supprimées, une exception est levée pour chacune d'elles.

[Mou93] propose aussi des solutions pour les problèmes de cohérence de la mémoire globale, partagée par les différentes tâches de raisonnement. Ces solutions consistent à utiliser des mémoires locales et gérer les conflits de mise à jour mémoire locale $\leftrightarrow$ mémoire globale.

Parmi les récentes extensions au modèle *Multi-GREAT*, un algorithme d'ordonnancement acceptant aussi bien des tâches suivant le modèle de raisonnement progressif, que des tâches suivant le modèle de méthodes multiples *TAEMS* (cf. section 1.2.1) est proposé dans [DMG97]. De plus, il y a une prise en compte de la notion d'utilité, appliquée à chaque niveau de raisonnement, qui permet de faire un choix sur les approximations mieux guidé que simplement le degré d'importance des tâches. Cette idée est approfondie dans [MZ97].

1.2.3 Planification temps réel

L'intitulé "Planification temps réel" (en anglais, *Realtime Search*) regroupe plusieurs travaux sur la façon d'explorer un espace de recherche en temps contraint[14]. Généralement, l'espace de recherche est exploré en vue de résoudre un problème de planification, d'où le titre de cette section. L'objectif est d'atteindre un état but avant une date t fixée, en étant parti d'un certain état initial et en appliquant des opérateurs de transition pour passer d'un état à l'autre. La contrainte temporelle est unique et correspond à une échéance stricte donnée comme contrat à l'algorithme d'exploration. L'ordonnanceur doit répartir le temps CPU entre la phase d'exploration de l'espace de recherche et la phase d'exécution du meilleur (sous-)plan trouvé par l'algorithme d'exploration. Nous présentons deux approches suivant que l'environnement est considéré comme dynamique ou statique.

1.2.3.1 Environnement dynamique : planification et exécution entrelacées

Un environnement dynamique signifie que la phase d'exploration travaille sur des données incomplètes. La profondeur maximale de la recherche n'est pas connue à l'avance. L'espace de recherche est vu comme un graphe, c'est à dire qu'il peut y avoir plusieurs chemins qui passent par le même noeud (possibilité de bouclage). L'approche consiste à réaliser une succession d'étapes de recherche à horizon limité et d'exécution du plan intermédiaire trouvé par l'étape de planification précédente. Cette approche est assez réaliste car elle considère qu'une recherche complète est impossible par manque d'information (par exemple, capteur à porté limitée) et que l'environnement (donc l'espace de recherche) évolue au cours du temps. La figure 1.10 [SH97] décrit cet entrelacement planification & exécution.
Pour la plupart des algorithmes évoqués ci-dessous, la stratégie de recherche est de type *meilleur en premier* (*Best-First Search*). Ces algorithmes diffèrent principalement par la façon d'effectuer

[13] En cas d'importance égale, la tâche dont l'échéance est la plus lointaine est approximée en premier. Cela permet de lui réallouer du temps si les tâches précédentes durent moins longtemps que prévu.

[14] Les travaux présentés dans cette section font appel à des algorithmes d'exploration d'un espace de recherche. Le lecteur peut se référer à la section 2.3.1 pour des explications sur ce type d'algorithme.

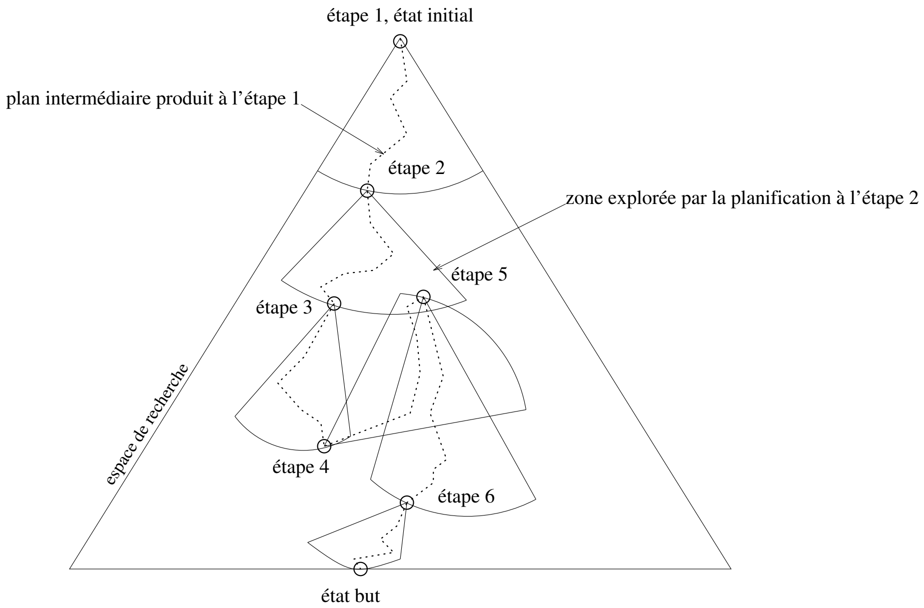


Fig. 1.10 - Exemple d'entrelacement d'étapes de planification et d'exécution d'un plan partiel

le choix d'un noeud pendant parmi les meilleurs et suivant le critère d'arrêt de l'étape d'exploration (fixe ou adaptatif).

L'algorithme *Real-Time-A** (RTA^*) [Kor90] effectue des recherches à profondeur fixe pour chaque noeud fils d'un noeud courant. La meilleure solution partielle (en terme de distance à l'optimum) trouvée à la frontière du sous-espace de recherche d'un des noeuds fils permet de sélectionner ce noeud fils comme nouveau noeud courant[15]. Le choix de la profondeur de recherche contrôle le compromis temps de résolution/qualité du plan.

L'algorithme *DYnamic Near Optimal Response-time Algorithm II* (*DYNORAII*) [SH93a, SH93b] cherche à améliorer la délimitation de l'horizon de recherche. Il s'appuie sur le critère d'arrêt $C_p \geq \alpha C_e$ avec C_p, le coût d'une planification (nombre de noeuds explorés) et C_e, le coût d'exécution estimé pour atteindre un état but. Le paramètre α permet de régler l'optimalité du plan, en fonction du contrat temporel. Pour $\alpha \to \infty$, l'algorithme se comporte comme l'algorithme optimal A^* [Pea84] ; pour $\alpha \to 0$, l'algorithme se comporte comme un algorithme glouton de descente en gradient. L'algorithme *Self-Adjusting Real-Time Search* (*SARTS*) [SH97, HS95] propose d'ajuster ce paramètre α d'un cycle de planification & exécution à l'autre. Au cycle i, $\alpha(i)$ est calculé en fonction du temps restant $T^r(i)$ et d'une estimation du temps nécessaire pour exécuter les prochains cycles $T_e^r(i)$: $\alpha(i) \leq \frac{T^r(i)}{T_e^r(i)} - 1$. De plus, ce calcul de $\alpha(i)$ permet de déterminer à l'avance s'il risque d'y avoir une violation de l'échéance. C'est le cas lorsque le calcul donne un $\alpha(i) \leq 0$.

L'algorithme *Decision Theoric A** (DTA^*) [RW91], proche de RTA^*, démontre l'intérêt d'évaluer au mieux l'utilité d'explorer chaque noeud pendant (générés par une recherche *meilleur en pre-*

[15] L'algorithme RTA^* n'exécute qu'un seul déplacement à la fois, suivant la stratégie d'agir au plus tard. Une fois un noeud choisi, l'algorithme recommence l'exploration en supposant que la possibilité de voir plus loin ne le dirigera pas forcément vers le même second noeud.

mier), lors de la phase de planification. Il utilise aussi la stratégie du *Best-First Search* mais en plus, départage les noeuds premiers ex-aequo. Soit β_1 un noeud jugé juste inférieur aux meilleurs noeuds. Il s'agit de sélectionner, parmi les meilleurs noeuds, celui qui a la plus petite probabilité d'excéder l'estimation de β_1 après être étendu. L'algorithme choisira donc de préférence un noeud pendant se trouvant à une plus grande profondeur, dont la distance à un état but est la plus petite.

L'algorithme *k-best* [Pem95] combine une connaissance probabiliste de l'évolution d'une heuristique évaluant le coût d'aller d'un noeud à un état but avec une connaissance exacte des coûts pour aller de la racine aux noeuds pendants. Le critère d'arrêt est la profondeur maximale d'exploration. Les k meilleurs noeuds pendants sont examinés, lors de l'arrêt de l'exploration, pour décider quel fils du noeud courant sélectionner. L'algorithme choisit le fils qui a la plus petite estimation des k meilleurs noeuds pendants, en fonction du minimum estimé des chemins allant de la racine à un état but.

Le succès des méthodes que nous venons de présenter dépend fortement de la précision de l'heuristique utilisée pour évaluer la distance à un état but. Nous verrons que dans le cadre général des *Problèmes de Satisfaction de Contraintes Valuées*, cette heuristique, appelée minorant de la valuation d'une affectation partielle, est relativement mal informée. C'est pourquoi une approche utilisant la stratégie du *Best-First Search* est délicate à mener, pour cause d'explosion mémoire de la liste des noeuds pendants. De même, une prévision sur l'évolution de l'heuristique risque d'être très incertaine, surtout en début de recherche. De plus, il y a toujours le risque que l'exploration à profondeur limitée se fasse piéger par des difficultés placées à une profondeur supérieure[16]. Enfin, l'exploration à profondeur limitée ne garantie pas l'optimalité.

1.2.3.2 Environnement statique : planification puis exécution

L'approche statique considère qu'elle a une connaissance complète du problème à résoudre. L'espace de recherche est plutôt représenté par un arbre. L'objectif est alors uniquement de contrôler au mieux l'algorithme de recherche en fonction du temps imparti de façon à proposer une solution la plus proche possible de la solution optimale, avec des garanties sur la distance à l'optimum de la solution proposée.

Il existe des algorithmes d'approximation en temps non polynomial [Pea84] qui effectuent une recherche systématique, comme l'algorithme du *Branch and Bound* [Tsa93] et qui sont guidés par une heuristique pour couper des parties de l'espace de recherche. Ces algorithmes garantissent la distance maximale à l'optimum de la solution rendue. La difficulté est de bien régler l'heuristique pour trouver dans le temps imparti la meilleure solution possible.

L'algorithme *Time-Constrained Approximate Guided Depth-First Search* (*TCGD*) [WC97] utilise l'heuristique $lb_i \geq \frac{ub_{best}}{1+\epsilon}$, avec lb_i, un minorant du noeud courant, ub_{best}, la meilleure solution trouvée et ϵ, le degré d'approximation. Il applique une recherche en profondeur d'abord avec comme particularité d'examiner (calcul des minorants) tous les fils du noeud courant avant d'en choisir un pour continuer sa recherche. L'algorithme *TCGD* exécute une succession de recherches avec des degrés d'approximation ϵ décroissants. Cette décroissance peut être linéaire

[16] L'affrontement entre joueurs humains et ordinateurs aux jeux d'échecs est un bon exemple. La profondeur de recherche d'un joueur reste, pour l'heure, supérieure à celle d'un ordinateur. Le joueur peut tenter de placer des pièges au-delà de l'exploration faite par l'ordinateur. Pourtant, la récente victoire de l'ordinateur *Deep Blue* d'IBM sur le champion du monde Kasparov montre qu'avec une puissance de calcul conséquente, la recherche à profondeur limitée peut être performante.

ou géométrique. Elle peut aussi être adaptative, dans le sens où les paramètres contrôlant le choix de ϵ varient en cours d'exécution suivant le comportement observé.

L'algorithme *Real-Time Search* (*RTS*) [CW91] compare et combine deux types d'heuristiques ; d'une part l'ϵ-approximation et d'autre part la θ-approximation, qui correspond à couper l'espace de recherche lorsque le minorant d'un noeud dépasse un certain seuil, noté θ. Un modèle mathématique décrit l'évolution du temps de résolution par rapport au degré d'approximation. Il permet de sélectionner le degré d'approximation en fonction du contrat temporel.

Enfin, [Chu94] examine des solutions fondées sur l'apprentissage par *algorithmes génétiques* pour adapter dynamiquement l'évolution des paramètres d'approximation.

Le principe des algorithmes d'approximation de la recherche s'applique bien au cadre des *Problèmes de Satisfaction de Contraintes Valuées*. Ils seront examinés plus en détail dans le chapitre 6. Cependant le modèle mathématique décrivant l'évolution du temps de résolution par rapport au degré d'approximation devient imprécis dans les cas où [CW91] :

- il y a des contraintes dures qu'il faut absolument respecter ;
- l'évaluation d'un minorant d'un noeud est très grossière, c'est à dire qu'elle différencie mal les noeuds.

Or les *Problèmes de Satisfaction de Contraintes Valuées* contiennent précisément ces deux cas.

1.3 Politiques d'ordonnancement de tâches interruptibles

L'approche interruptible considère que les tâches sont capables de résoudre un problème de manière incrémentale, par raffinements successifs, de façon à proposer un résultat de qualité croissante au cours du temps. Initialement apparu dans [DB88], le terme ANYTIME désigna un algorithme satisfaisant le modèle de tâche de l'approche interruptible. Par la suite, le mot ANYTIME a pris un sens plus général, englobant la notion d'algorithme interruptible et les politiques d'ordonnancement pour ce type d'algorithme.

Nous présentons dans la section 1.3.1 le modèle de tâche d'un algorithme interruptible. Nous examinons dans la section 1.3.2 un autre modèle de tâche plus réaliste, ainsi qu'un ordonnanceur adapté à ce modèle. Ensuite, nous décrivons deux politiques d'ordonnancement ANYTIME pour un modèle de tâche interruptible. La première est un ordonnancement pré-calculé ; la deuxième assure un contrôle dynamique. Plus précisément, c'est à partir des profils de qualité des algorithmes interruptibles que se construisent les politiques d'ordonnancement ANYTIME. Cette connaissance permet de composer les profils de qualité des différentes tâches à ordonnancer de manière à déterminer l'ordonnancement maximisant l'utilité globale du système. Cette phase de compilation d'algorithmes interruptibles se fait avant l'exécution du système, nous l'étudions dans la section 1.3.3. Elle indique comment répartir le temps de calcul aux différentes tâches. Néanmoins ce n'est pas une indication infaillible au sens où les profils ne donnent que des informations probabilistes et où l'environnement extérieur est souvent imprévisible. Il est nécessaire d'ajouter à cette compilation un contrôle dynamique de l'évolution de la qualité des tâches, nous verrons cela dans la section 1.3.4.

1.3.1 Algorithme interruptible

L'approche interruptible considère un modèle de tâche idéal au sens où la tâche est capable de rendre un résultat de qualité non nulle à tout instant et que ce résultat soit de la meilleure qualité possible vis à vis du temps imparti.

1.3.1.1 Propriétés algorithmiques

Pour que les politiques d'ordonnancement ANYTIME soient applicables, il est nécessaire de préciser davantage ce modèle de tâche. Ainsi plusieurs propriétés algorithmiques doivent être vérifiées pour satisfaire au modèle ANYTIME [Zil96] :

- être capable de mesurer de façon précise la qualité d'une solution ; dans le cas où cette mesure correspond à une distance entre un résultat approché et un résultat exact, la qualité n'est mesurable que si ce résultat exact est calculable en un temps raisonnable ;
- pouvoir effectuer cette mesure rapidement, en un temps constant ; c'est la principale difficulté qui se pose pour adapter un algorithme d'optimisation au modèle ANYTIME ;
- avoir une croissance monotone de la qualité des solutions intermédiaires en fonction de la qualité des entrées et du temps de calcul alloué ; lorsque la qualité est mesurable, le système peut toujours garantir la monotonicité en ne retournant que la dernière solution de qualité maximale plutôt que la dernière solution calculée ;
- avoir une qualité qui évolue de manière cohérente avec la qualité des entrées et le temps de calcul alloué, c'est à dire que la variance sur la qualité en sortie de l'algorithme est suffisamment restreinte pour permettre des prévisions sûres de la qualité à l'exécution ;
- avoir un profil de qualité qui croît d'avantage au début de l'exécution de l'algorithme qu'à la fin ;
- être interrogeable, c'est à dire pouvoir fournir à tout moment une solution ;
- être préemptible, c'est à dire pouvoir être suspendu et relancé sans trop d'*overhead* du système.

1.3.1.2 Profils de qualité

Contrairement à un algorithme classique qui ne fournirait qu'un unique résultat optimal à la fin de son temps d'exécution, un algorithme interruptible doit être capable d'élaborer des solutions intermédiaires de qualité croissante au cours du temps. Le profil de qualité de cet algorithme décrit l'évolution de la qualité du résultat intermédiaire en fonction du temps de calcul alloué à l'algorithme.

Sur la figure 1.11, à la date t_1 inférieure au temps de calcul complet t_{class} de l'algorithme classique, la solution rendue par l'algorithme interruptible est de qualité $q_1^{\text{algo-it}}$, inférieure à la qualité optimale q_{max} mais supérieure à la qualité nulle $q_1^{\text{algo-class}}$ de l'algorithme classique qui n'a encore rien produit.

Un profil de qualité décrit le comportement d'un algorithme pour un ensemble de problèmes similaires. La connaissance des profils de qualité varie suivant qu'elle est analytique ou discrète, suivant qu'elle est probabiliste ou non, suivant qu'elle prend en compte ou non la qualité en entrée de l'algorithme ... Les principales représentations d'un profil de qualité sont :

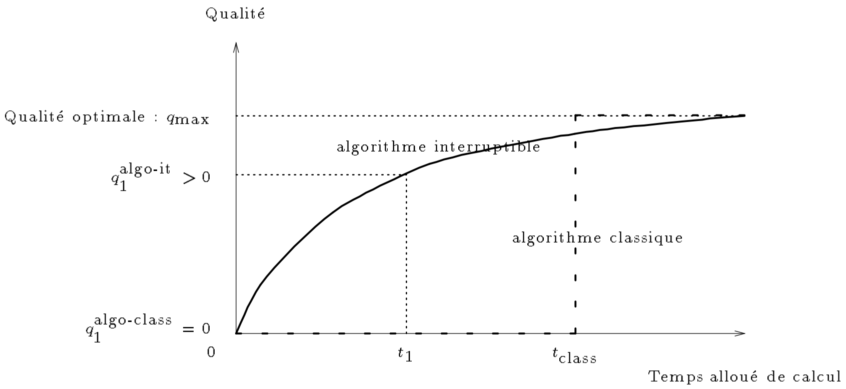


Fig. 1.11 - Profil de qualité d'un algorithme interruptible comparé à un algorithme classique non interruptible

- analytique : $t \mapsto \alpha(1 - \exp^{-\lambda t}), \lambda > 0$; un ordonnancement optimal, utilisant des profils analytiques, est décrit dans [Bod91] ;
- discrète :
  - profil standard, ne dépend que du temps alloué, peut être codé de différentes façons :
    * déterministe : $t \mapsto q$, avec q, valeur moyenne ; c'est la représentation la plus simple d'un profil discrétisé ; c'est celle utilisée pour les profils de qualité des algorithmes de tri des figures 1.6 et 1.7 ;
    * par intervalle : $t \mapsto q_{lb}, q_{ub}$, avec q_{lb} comme minorant de la qualité et q_{ub} comme majorant ;
    * probabiliste : $t \mapsto Proba(q)$, avec $Proba(q)$, une distribution de probabilité sur l'intervalle $[0, 1]$ d'avoir une certaine qualité q ;
  - profil conditionnel probabiliste : $t, q_{in} \mapsto Proba(q_{out})$; cette représentation permet de prendre en compte la qualité des données en entrée ; c'est cette représentation qui a été utilisée pour la compilation d'algorithmes interruptibles dans [Zil93] (cf. section 1.3.3) ;
  - profil dynamique probabiliste : $\Delta t, q_t \mapsto Proba(q_{t+\Delta t})$; cette représentation est préférée lorsque la variance sur la qualité en sortie de l'algorithme à t donné est grande ; elle permet de mieux suivre d'éventuelles variations de performance de l'algorithme ; son utilisation suppose de contrôler la qualité q_t à l'exécution (cf. section 1.3.4).

La connaissance d'un profil de performance nécessite soit une expertise précise sur le comportement d'un algorithme face à un certain type d'entrée, soit une phase d'apprentissage à partir d'un jeu de test. Cette phase risque d'être assez lourde pour les algorithmes de recherche arborescente [SH93a]. Une fois connu le profil discret, il est possible de le modéliser par une fonction analytique [DB88].

1.3.1.3 Utilisation des profils de qualité

Les profils de qualité permettent de calculer l'utilité intrinsèque d'un résultat partiel vis à vis de son utilisation dans le système, en fonctionnement nominal. La prise en compte de la charge du système, de la pression temporelle aboutit au calcul de l'utilité du résultat partiel pour le système. L'objectif des ordonnanceurs suppose de maximiser cette utilité (cf. section 1.1.3).

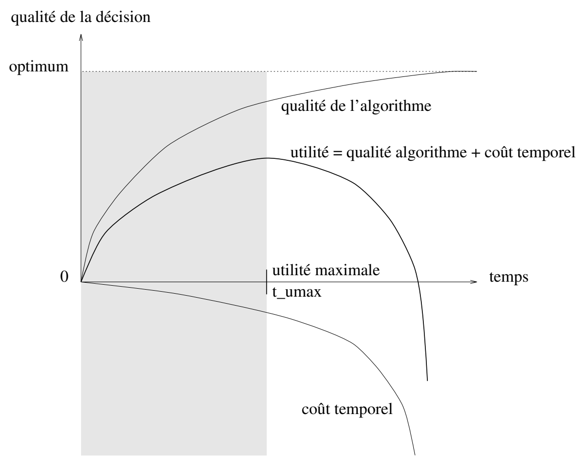


Fig. 1.12 - Maximisation de l'utilité

Prenons l'exemple de la figure 1.12 qui représente le profil de qualité d'un algorithme, dans un système soumis à une échéance relative dont le coût temporel commence au temps zéro. L'utilité est simplement définie par la somme de la qualité et du coût temporel. L'ordonnanceur détermine le temps maximal d'exécution de l'algorithme, t_{umax}, lorsque l'utilité est maximale, ce qui revient ici à l'instant où l'évolution de l'utilité devient négative. La zone grisée indique les instants où l'algorithme peut être interrompu.

Ainsi le contrôle fondé sur une connaissance de l'utilité des résultats permet de raisonner plus finement que directement sur les profils de qualité. Cette approche est apparue initialement pour le calcul flexible proposé dans [Hor90]. Nous verrons dans les sections 1.3.3 et 1.3.4 sur les ordonnanceurs ANYTIME comment les profils de qualité sont mis en oeuvre.

1.3.2 Politiques d'ordonnancements de tâches imprécises

1.3.2.1 Modèle de tâche imprécise

Avant d'aborder les politique d'ordonnancement ANYTIME proprement dites, nous présentons quelques travaux réalisés autour du modèle de tâche imprécise. En effet, ce modèle offre une vision plus réaliste que celui de l'approche ANYTIME, en décomposant l'exécution d'une tâche en une partie obligatoire et une partie optionnelle (cf. figure 1.13) [LLS+91]. La partie optionnelle peut être en partie exécutée d'où la production de résultats partiels *imprécis*. À l'inverse, une tâche interruptible suppose être capable de produire un résultat de qualité non nulle à tout instant. Par exemple, la figure 1.7 montre que l'algorithme du tri par tas a une qualité pour le critère du *% de la liste trié par le bas* qui est nulle durant environ les dix premières centi-secondes. Cela correspond au temps mis pour initialiser la structure de tas des 10^5 éléments à trier. C'est une partie obligatoire de l'algorithme de tri par tas pour pouvoir ensuite récupérer les éléments dans l'ordre croissant. Il est intéressant de remarquer que cette partie obligatoire n'existe pas pour ce même algorithme si nous considérons l'évolution de sa qualité suivant le critère du *désordre moyen* (cf. figure 1.6)[17]. Notons que le modèle de tâche imprécise permet aussi de représenter les tâches classiques (parties optionnelles nulles) et les tâches interruptibles (parties obligatoires nulles).

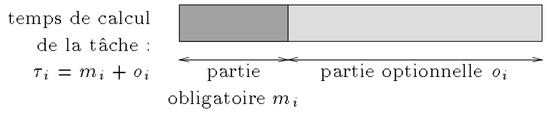


Fig. 1.13 - Modèle de tâche imprécise

Les principaux objectifs pour l'ordonnancement de tâches imprécises sont :

- ordonnancement exact pour les parties obligatoires (l'ordonnanceur réserve du temps pour exécuter ces parties) ;
- minimisation d'une certaine erreur E pour les parties optionnelles, suivant le temps restant ; l'erreur E est définie de plusieurs façons [Shi92] :
  - minimisation de l'erreur totale : $E = \sum_{i=1}^{N} \omega_i \times (o_i - \delta_i)$, avec N, le nombre de tâches à ordonnancer, $\delta_i \in [0, o_i]$, le temps effectivement alloué à la partie optionnelle o_i, et ω_i, le poids d'une tâche qui mesure son importance relative ;
  - minimisation de l'erreur normalisée maximale : $E = \max_{i=1}^{N} \frac{(o_i - \delta_i)}{o_i}$; l'ordonnancement essaie de répartir équitablement le temps restant à l'ensemble des parties optionnelles ;
  - minimisation du nombre de parties optionnelles non exécutées : $E = \sum_{i=1}^{N} \delta'_i$, avec $\delta'_i \in \{0, 1\}$, c'est une contrainte 0/1 au sens où la partie optionnelle est soit entièrement exécutée, soit entièrement ignorée ;

1.3.2.2 Ordonnanceur *NORA*

De nombreux algorithmes d'ordonnancement sont définis suivant que les tâches sont connues ou non avant l'exécution du système, suivant qu'elles sont retardées ou non[18], suivant qu'elle sont périodiques ou apériodiques [Shi92]. Les choix effectués pour simplifier le modèle de tâche permettent d'obtenir des ordonnanceurs de complexité temporelle polynomiale et éventuellement optimaux. Par exemple, l'algorithme *No-Off-line tasks and on-line tasks Ready upon Arrival* (*NORA*) [SL92] ordonne des tâches de poids identiques qui ne sont connues que lors de l'exécution du système et minimise l'erreur totale. Sa complexité en temps est en $O(n \log n)$, avec n, une étape élémentaire de l'algorithme. Cet algorithme est glouton car il essaie d'ordonnancer le plus de parties optionnelles possible. Ainsi l'erreur totale est bien minimale. En revanche cet algorithme est sous-optimal au sens où il peut ne pas trouver d'ordonnancement exact pour les parties obligatoires alors qu'une autre politique d'ordonnancement en aurait trouvé un. Cela peut arriver lorsque *NORA* exécute en premier une partie optionnelle ayant une échéance plus proche, et que l'exécution d'une partie obligatoire ayant une échéance plus lointaine est repoussée. Il est à noter que le coût de l'ordonnancement n'est pas pris en compte.

[17] La construction du tas a pour effet de réduire le désordre moyen.
[18] Une tâche est dite retardée si elle ne doit pas s'exécuter avant une date donnée.

Théorème 2

[SL92]

Il n'existe pas d'ordonnanceur qui, quelle que soit la configuration de tâches imprécises connues à l'exécution, garantisse à la fois que s'il existe un ordonnancement exécutant toutes les parties obligatoires, l'ordonnanceur le trouve et de plus minimise l'erreur totale[19].

Un modèle de tâche plus complet, avec prise en compte de contraintes sur l'utilisation de ressources et sur la communication entre tâches, est défini pour un environnement d'expérimentation d'algorithmes d'ordonnancement nommé *Prototyping Environment for Real Time Systems* (*PERTS*) [LLD+96]. Une intégration du modèle de tâche imprécise est en cours pour le système d'exploitation temps-réel *Mach* [HFL96].

1.3.2.3 Bilan

Les politiques d'ordonnancement de tâches imprécises qui minimisent l'erreur totale font l'hypothèse que les tâches produisent toutes des résultats intermédiaires dont la qualité croît de la même façon pour toutes les tâches. Ainsi minimiser l'erreur totale revient à maximiser la somme des qualités des tâches. Cette hypothèse relativement grossière n'est pas faite par les ordonnanceurs ANYTIME puisqu'ils exploitent le profil de qualité des tâches.

Nous avons vu dans la section 1.1.3 que la prédictibilité temporelle des systèmes temps réel classiques est remplacée par la garantie d'obtention de seuils d'acceptabilité pour les systèmes temps réel intelligents. Obtenir un seuil d'acceptabilité signifie pour une tâche de l'exécuter jusqu'à ce qu'elle produise un résultat de qualité supérieure à ce seuil. Ce temps d'exécution indique la durée de la partie obligatoire. Il est nécessaire d'avoir une évaluation d'une borne supérieure de ce temps pour obtenir un système temps réel intelligent prédictible. Cette borne supérieure doit être la plus petite possible pour éviter un sur-dimensionnement du système.

Nous verrons que dans le cas général des problèmes d'optimisation sous contraintes, aucune borne supérieure réaliste ne peut être trouvée. Dans ce cas, le système assure uniquement une dégradation progressive de ses performances en fonction de sa charge.

1.3.3 Politique d'ordonnancement ANYTIME : compilation d'algorithmes interruptibles

1.3.3.1 Principes généraux

La compilation répond au problème de la répartition du temps de calcul pour un ensemble de tâches se partageant un unique processeur, ayant des relations de dépendance entre-elles et soumises à une échéance stricte globale. Il s'agit de transformer un ensemble de profils de qualité d'un groupe de tâches interruptibles en un unique profil de qualité correspondant à une seule tâche englobante vis à vis de l'ordonnanceur. Cette tâche englobante est appelée tâche contractuelle ANYTIME car c'est une tâche contractuelle qui est capable d'adapter son raisonnement quel que soit le contrat temporel imparti. Dans notre cas, à chaque contrat, est associé une allocation temporelle pour chaque tâche interruptible du groupe et une qualité en sortie de la tâche englobante qui est fonction de ces allocations.

[19] En revanche, si l'objectif est uniquement de produire un ordonnancement exact pour les parties obligatoires à chaque fois que cela est possible, alors nous conjecturons que la politique d'ordonnancement *Earliest Deadline First* (cf. section 1.1.1) adaptée aux tâches connues à l'exécution est optimale.

La figure 1.14 décrit un groupe de 5 tâches interruptibles. Les flèches indiquent les dépendances de données entre les tâches. Ce graphe de dépendance admet deux entrées de qualité $q1_{in}$ et $q2_{in}$; il a une seule sortie de qualité $q3_{out}$. L'objectif de la compilation est, pour chaque temps t donné, de trouver une allocation optimale des temps t_i des 5 tâches de manière à avoir la qualité $q3_{out}$ maximale.

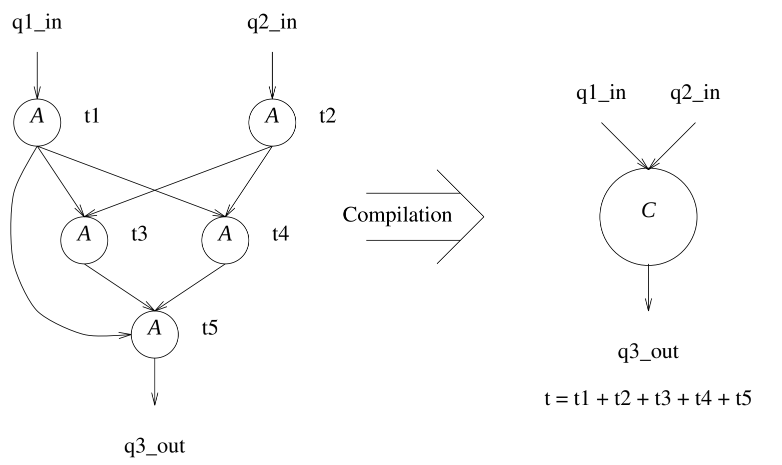


Fig. 1.14 - Compilation d'algorithmes interruptibles

Plusieurs techniques de compilation d'algorithmes interruptibles sont décrites dans [Zil93]. Elles effectuent soit un parcours exhaustif des combinaisons d'allocation temporelle par une méthode systématique de recherche arborescente, soit un parcours inexact par une méthode de recherche locale (le voisinage est défini par des échanges d'unité de temps entre deux noeuds). Dans le cas général, la compilation est un problème NP-*dur*. Pour réduire la complexité de la compilation, Zilberstein définit une compilation locale comme étant une allocation optimale ne considérant à la fois qu'un seul noeud et ses prédécesseurs. Le calcul est effectué des feuilles, les noeuds d'entrée, en remontant jusqu'à la racine, l'unique noeud de sortie.

Théorème 3
[Zil93]
La compilation locale est globalement optimale et prend un temps pseudo-polynomial $O(n \times k \times T^b)$, avec T, le nombre d'unité de temps considérées, b, le facteur de branchement d'un noeud du graphe, k, le nombre de noeuds, n, le nombre d'entrée du graphe, si :

- *le graphe de dépendance forme un arbre*
- *il y a croissance monotone de la qualité en sortie en fonction de la qualité en entrée :*

$$\forall t, \quad q_{in} \leq q'_{in} \Rightarrow q_{out}(t, q_{in}) \leq q_{out}(t, q'_{in})$$

1.3.3.2 Exemple : compilation d'algorithmes de tri

Pour illustrer cet ordonnancement par compilation, nous présentons un exemple reprenant les algorithmes de tri. Soit deux algorithmes de tri interruptibles mis en séquence : le tri de Shell puis le tri rapide (voir figure 1.15). Nous cherchons d'abord à réduire rapidement le désordre

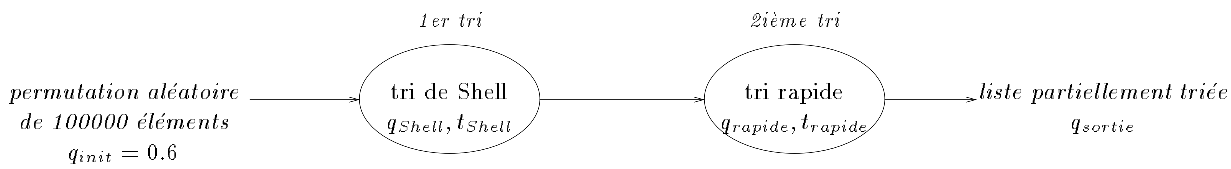


Fig. 1.15 - Graphe de dépendance de deux algorithmes de tri

moyen puis à obtenir les premiers éléments de la liste complètement triés. Nous choisissons de pondérer ces deux critères pour obtenir une utilité intrinsèque d'une liste partiellement triée égale à $0.9 \times q_{(\text{désordre moyen})} + 0.1 \times q_{(\text{\% liste triée})}$.

Une application, réalisée au CERT par Jean-Michel Forhan [AF96], implémente l'ensemble des algorithmes de compilation proposés dans [Zil93] pour des graphes de dépendance de tâches interruptibles ayant des profils de qualité conditionnels non probabilistes (la qualité en sortie est une valeur moyenne). Les tests sont effectués sur une permutation aléatoire de 10^5 éléments. La figure 1.16 représente le profil de qualité conditionnel du tri rapide en fonction d'une liste préalablement partiellement triée par le tri de Shell de qualité $q_{Shell} = \{0.6, 0.78, 0.83, 0.86, 0.88, 0.89, 0.9\}$. Il est intéressant de noter que la propriété de monotonicité sur l'évolution de la qualité n'est pas respectée par l'algorithme de tri rapide. En effet, le tri rapide est plus lent à trier une liste dont le désordre moyen est plus faible.

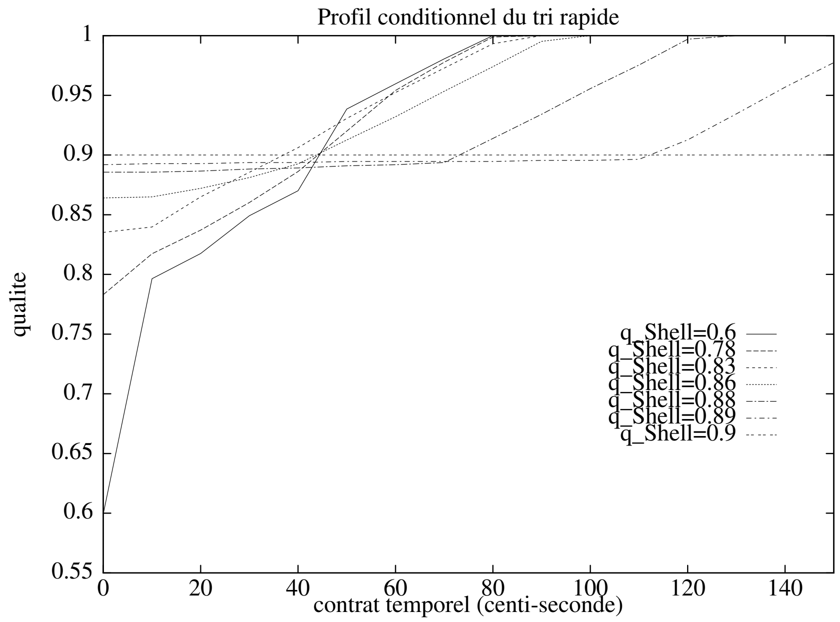


Fig. 1.16 - Profil de qualité conditionnel du tri rapide

Le résultat de la compilation globale par une méthode systématique produit un nouveau profil de qualité (appelé *compilation* dans la figure 1.17) correspondant à une unique tâche contractuelle ANYTIME, et produit une répartition du temps de calcul entre les deux algorithmes de tri (allocations temporelles de la figure 1.18). La figure 1.17 indique aussi les deux profils de qualité des deux algorithmes de tri lorsqu'ils sont exécutés isolément. Suivant le contrat temporel alloué globalement au tri, un des deux algorithmes domine l'autre en terme de qualité obtenue. Les irrégularités observées dans la figure 1.18 proviennent de l'interpolation continue de points d'un profil discrétisé qui est implémentée dans l'application.

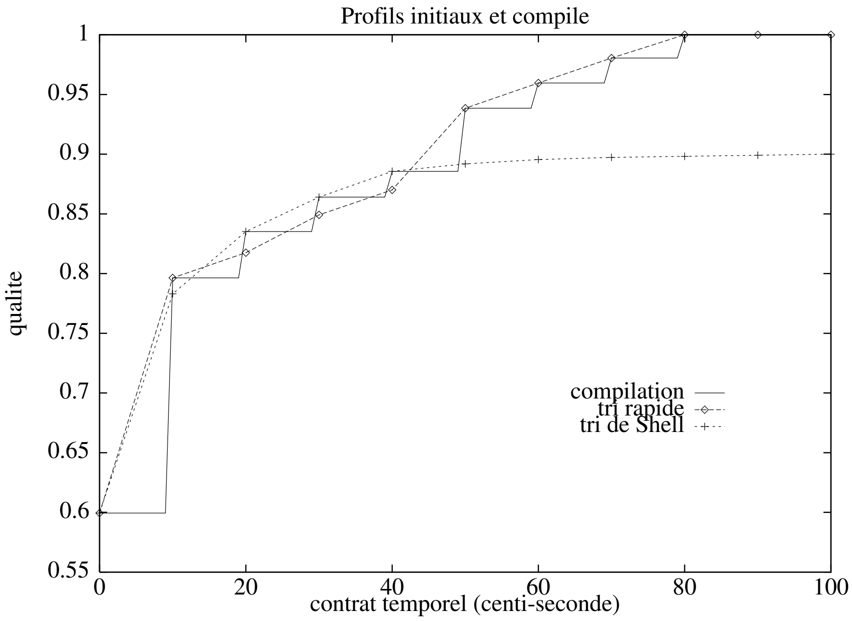


Fig. 1.17 - Comparaison des profils de qualité des algorithmes de tri exécutés séparément et du profil obtenu par compilation

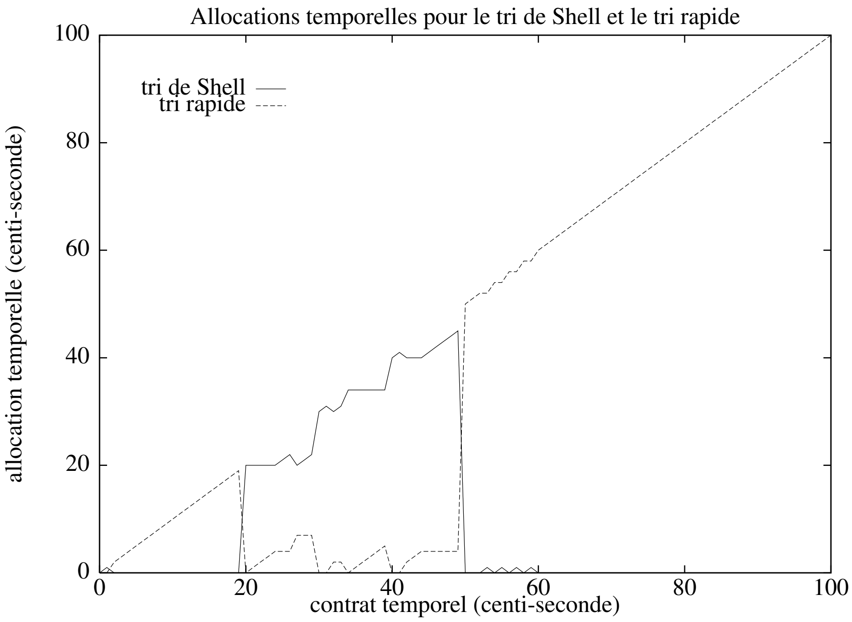


Fig. 1.18 - Répartition des allocations temporelles aux deux algorithmes de tri

1.3.3.3 Passage d'une tâche contractuelle à une tâche interruptible

Nous avons vu que la compilation transforme un ensemble de tâches interruptibles en une unique tâche contractuelle. Or, un des problèmes des systèmes temps réel intelligents est de faire face à des modifications imprévues de l'environnement extérieur pouvant raccourcir ou allonger les échéances temporelles. Seule une tâche interruptible peut faire face à ces modifications. Un théorème important indique comment transformer une tâche contractuelle en une tâche interruptible avec un minimum de perte de temps pour la génération des résultats partiels :

Théorème 4 ———
Réduction *[Zil93]*
Il est possible de transformer un algorithme contractuel A en un algorithme interruptible B tel que, quelles que soit les entrées, la propriété suivante soit respectée :

$$\forall t, \qquad q_B(4t) \geq q_A(t)$$

La construction de l'algorithme B se fait en exécutant successivement l'algorithme A avec un contrat τ, puis 2τ, ... $2^i\tau$, ... Le profil de qualité de B est une fonction en escalier. Pour cette stratégie de construction, le coefficient multiplicateur 2 est optimal (il implique le plus petit facteur possible dans l'expression précédente, soit le facteur 4).

1.3.3.4 Bilan

Pour conclure, l'approche par compilation est une solution intéressante au problème de la composition d'algorithmes interruptibles. Le profil obtenu par compilation permet d'ordonnancer rapidement les tâches lors de l'exécution du système (ordonnancement pré-calculé). Son efficacité repose sur une connaissance précise et fiable des profils de qualité, ainsi que sur la propriété de monotonicité de la qualité en sortie par rapport aux entrées (cf. théorème 3), nécessaire pour compiler dans des temps raisonnables des graphes de dépendance conséquents.

De plus, la notion de composition d'algorithmes permet de créer des librairies d'algorithmes interruptibles élémentaires servant de briques de base pour construire des algorithmes interruptibles plus complexes.

1.3.4 Politique d'ordonnancement ANYTIME : contrôle dynamique de la qualité

1.3.4.1 Principes généraux

L'ordonnancement d'une tâche interruptible peut s'effectuer principalement de deux façons. Premièrement, lors de l'arrivée d'une tâche, l'ordonnanceur lui alloue un temps d'exécution optimal en se fondant sur son profil de qualité et sur sa courbe d'utilité (recherche de t_{umax} dans la figure 1.12). Cette approche correspond à un ordonnancement pré-calculé qui consiste à déterminer un contrat temporelle pour la tâche, à l'instar de la compilation d'algorithmes interruptibles. L'approche est viable tant qu'il n'y a pas d'incertitude sur l'évolution de la qualité et qu'il n'y a pas de changement dans l'environnement extérieur (entraînant une modification de la courbe d'utilité).

En revanche, dans les cas où la première approche n'est pas viable, une seconde approche consiste à contrôler dynamiquement la tâche.

Définition 4
Contrôle dynamique *[HZ96] :*
le contrôle dynamique de l'évolution d'une tâche interruptible implique d'évaluer la qualité du résultat courant, de refaire une prévision sur l'évolution de l'utilité, et de décider s'il est judicieux de continuer l'exécution de la tâche.

[Hor90, BD94, Zil93] proposent un contrôle cyclique qui évalue régulièrement le gain de continuer l'exécution de la tâche plutôt que d'utiliser directement le résultat courant. À chaque délibération, l'utilité du résultat courant au temps t est comparée à l'utilité estimée du résultat obtenu après un temps Δt supplémentaire de raisonnement. Le critère d'arrêt de la tâche est atteint lorsque l'utilité estimée est inférieure à l'utilité courante. Ce contrôle permet de prendre en compte l'évolution de l'environnement extérieur de façon optimale sous certaines conditions :

Théorème 5
Optimalité du contrôle dynamique *[Zil93] :*
soit le calcul de l'utilité d'un résultat égale à la différence de l'utilité intrinsèque de ce résultat et du coût temporel. Alors le contrôle dynamique utilisant le critère d'arrêt fondé sur ce calcul de l'utilité est optimal quand $\Delta t \to 0$, quand l'utilité intrinsèque évolue suivant une fonction monotone croissante concave bas[20] *et quand le coût temporel évolue suivant une fonction monotone croissante concave haut.*

Néanmoins ce contrôle n'est optimal que pour un certain type de profil de qualité fixé. Si c'est un profil conditionnel probabiliste qui est choisi, il ne tient pas compte de l'évolution passée de la qualité, à l'instant t de la délibération. L'examen de cette évolution aurait permis de détecter une variation de l'évolution de la qualité par rapport à la prévision du profil conditionnel. Délibérer sur un connaissance complète de l'évolution passée n'est pas réaliste. Le profil dynamique probabiliste $\Delta t, q_t \mapsto Proba(q_{t+\Delta t})$ (cf. section 1.3.1) considère une connaissance minimale de l'évolution passée, il est uniquement informé par la qualité courante.

Par la suite, nous décrivons un contrôle dynamique, proposé dans [HZ96], qui utilise les profils dynamiques probabilistes et qui vise à résoudre uniquement le problème de l'incertitude sur l'évolution de la qualité[21]. Pour simplifier les calculs, [HZ96] utilisent une discrétisation du temps et de la qualité. La précision de cette discrétisation détermine le compromis précision des calculs versus temps d'ordonnancement. Le temps évolue entre $t_0, \ldots, t_n$, avec t_0, le temps de début d'exécution de la tâche et t_n, son temps maximal d'exécution. La qualité est représentée par des niveaux allant de q_0 à q_l, avec q_{start}, la qualité à l'instant t_0. La distribution de probabilité donnée par le profil dynamique probabiliste est discrétisée par un tableau à trois dimensions contenant la probabilité $Pr(q_j|q_i, \Delta t)$ d'avoir la qualité q_j après un temps supplémentaire Δt et en ayant précédemment observé la qualité courante q_i. L'ordonnanceur n'a qu'une seule tâche à contrôler qui est soumise à une échéance relative. La fonction utilité $U(q_i, t_k)$ ne dépend que du temps t_k et de la qualité q_i obtenue à cette date par la tâche.

[20] Une fonction f est dite concave haut (resp. bas) sur un intervalle I si elle est continue, différentiable par morceau et $\forall x, y \in I, x < y \Rightarrow f'(x) \leq f'(y)$ (resp. $f'(x) \geq f'(y)$).

[21] Une extension au contrôle pour gérer aussi le problème de l'évolution incertaine de l'environnement impliquerait une présentation plus chargée, où apparaîtrait la probabilité d'être dans un certain état S_t de l'environnement à une date t donnée.

Le contrôleur doit déterminer à l'instant t_k, avec une solution courante de qualité q_i, combien de temps supplémentaire Δt accorder à la tâche et décider si la tâche est définitivement arrêtée après ce temps Δt (décision notée $m = stop$) ou s'il y aura à nouveau un contrôle (décision notée $m = monitor$). Pour cela il recherche les paramètres $(\Delta t, m)$ qui maximisent la fonction $V(q_i, t_k)$, gain estimé d'attendre la date $t_k + \Delta t$, définie de manière récursive :

$$V(q_i, t_k) = \max_{\Delta t, m} \left[\begin{array}{ll} \sum_{j=0}^{l} Pr(q_j|q_i, \Delta t)\, U(q_j, t_k + \Delta t) & \text{si } m = stop \\ \sum_{j=0}^{l} Pr(q_j|q_i, \Delta t)\, V(q_j, t_k + \Delta t) - C & \text{si } m = monitor \end{array} \right. \tag{1.1}$$

L'originalité de ce contrôleur est de prendre en compte le **temps consommé C pour effectuer le contrôle dynamique** et de minimiser le nombre de fois où le contrôle est re-effectué. La maximisation de V permet d'avoir un contrôleur optimal, au sens où il maximise l'utilité des résultats fournis par un algorithme interruptible dont l'évolution de la qualité est totalement prédite par le profil dynamique probabiliste. Il est possible d'employer des techniques de programmation dynamique [CLR94] pour rechercher les paramètres $(\Delta t, m)$ optimaux. Cela permet de pré-calculer, à l'instar de la compilation d'algorithmes interruptibles, une table de décision contenant, pour chaque durée d'exécution t_i et pour chaque qualité q_i, un choix optimal des paramètres $(\Delta t, m)$.

1.3.4.2 Prise en compte de l'incertitude sur la qualité courante

La description précédente suppose que le contrôleur peut évaluer précisément la qualité courante en un temps constant inférieur à C. Il existe des cas où la qualité courante ne peut pas être mesurée de façon exacte. Par exemple, en optimisation, lorsque le problème consiste à trouver une solution de coût minimum, la qualité est habituellement définie par le rapport *coût de la solution courante* sur *coût d'une solution optimale*. À l'exécution, l'optimum est inconnu ; donc la qualité n'est pas calculable. Néanmoins, il est possible d'estimer la qualité avec un certain degré de probabilité. Cette estimation se fonde sur le calcul d'un critère, noté f_r, caractérisant la solution courante, ainsi que sur le temps t_k déjà consommé par l'algorithme. Dans le cas des problèmes d'optimisation, f_r peut être calculé à partir d'une connaissance d'un minorant de l'optimum :

$$f_r = \frac{\textit{coût de la solution courante}}{\textit{minorant du coût d'une solution optimale}}$$

Nous verrons dans la deuxième partie de la thèse comment obtenir à l'exécution un minorant de l'optimum d'un problème d'optimisation.

Le critère f_r est lui aussi discrétisé par des niveaux allant de f_0 à f_p. [HZ96] définit la probabilité $Pr(q_i|f_r, t_k)$ que la solution courante ait la qualité q_i en ayant observé le critère f_r, après un temps d'exécution t_k. De même, la probabilité $Pr(f_r|q_i, t_k)$ dénote la probabilité d'observer à l'exécution le critère f_r si la solution courante a effectivement la qualité q_i au temps t_k. Ces deux probabilités sont apprises lors d'une phase d'expérimentation du système sur des jeux de test, à l'instar de l'apprentissage du profil dynamique probabiliste. Le contrôleur est similaire à celui décrit précédemment, à l'exception de la formule définissant $V(f_r, t_k)$ qui prend maintenant en compte le critère f_r :

$$V(f_r, t_k) = \max_{\Delta t, m} \left[\begin{array}{ll} \sum_{j=0}^{l} Pr(q_j|f_r, t_k, \Delta t)\, U(q_j, t_k + \Delta t) & \text{si } m = stop \\ \sum_{s=0}^{p} Pr(f_s|f_r, t_k, \Delta t)\, V(f_s, t_k + \Delta t) - C & \text{si } m = monitor \end{array} \right.$$

avec, (1.2)

$$\begin{array}{rcl} Pr(q_j|f_r, t_k, \Delta t) & = & \sum_{i=0}^{l} Pr(q_i|f_r, t_k)\, Pr(q_j|q_i, \Delta t) \\ Pr(f_s|f_r, t_k, \Delta t) & = & \sum_{i=0}^{l} Pr(q_i|f_r, t_k) \sum_{j=0}^{l} Pr(q_j|q_i, \Delta t)\, Pr(f_s|q_j, t_k + \Delta t) \end{array}$$

1.3.4.3 Bilan

Le contrôle dynamique que nous venons de décrire a l'intérêt d'être rapide puisque les paramètres $(\Delta t, m)$ peuvent être pré-calculés dans une table de décision. Cette table permet de départager les problèmes qui ne nécessitent pas de contrôle dynamique (tâche dont l'évolution de la qualité est prédictible), des problèmes qui en ont besoin. [HZ96] appliquent leur contrôle sur des problèmes du voyageur de commerce [AL97]. Ils observent que les instants de contrôle ($m = monitor$ dans la table) sont plus fréquents à l'approche de la fin prévue de la tâche.

Le calcul de V permet d'exhiber le bon compromis entre le temps passé à calculer le critère d'observation f_r (ce temps est inclus dans C) et le gain obtenu par l'application d'un contrôle ayant une meilleure estimation de la qualité courante.

Enfin, le succès d'une telle politique d'ordonnancement dépend de la précision des informations probabilistes recueillies lors d'une phase d'apprentissage. C'est d'autant plus vrai pour le contrôleur qui traite de l'estimation de la qualité courante, car les probabilités à connaître sont plus nombreuses.

Nous verrons dans les sections suivantes que le calcul de f_r, pour un problème d'optimisation sous contraintes, est difficile. L'objectif de cette thèse est de proposer des techniques pour réaliser ce calcul, c'est à dire pour estimer en cours d'exécution la qualité d'une solution.

Conclusion

Lorsque le contrat temporel est connu et fixe, un algorithme contractuel est préférable à un algorithme interruptible car il ne perds pas de temps à élaborer des résultats intermédiaires. En revanche, si l'environnement extérieur change, alors le travail commencé par la tâche contractuelle risque d'être inutile (contrat raccourci) ou bien insuffisant (contrat étendu). La politique du raisonnement progressif améliore cette situation, en jalonnant le raisonnement de résultats intermédiaires. L'approche contractuelle reste assez stricte lorsqu'il s'agit d'ordonnancer plusieurs tâches. Elle impose de connaître précisément les durées d'exécution des différentes approximations et méthodes de raisonnement, ainsi que les qualités produites par ces méthodes.

Au contraire, l'approche interruptible se veut plus souple. Premièrement, elle propose des algorithmes qui s'adaptent mieux aux variations de l'environnement extérieur car ils produisent des résultats partiels de façon quasi-continue. Deuxièmement, elle s'appuie sur une prédiction probabiliste de l'évolution de la qualité d'une tâche interruptible en fonction du temps : les profils de qualité. Une utilisation statique de ces profils consiste à générer par compilation un ordonnancement quasi-optimal d'un ensemble de tâches interruptibles. Cela permet en fait de produire un système contractuel plus précis, faisant une meilleure utilisation du temps disponible, que ceux des approches contractuelles classiques. De plus, l'utilisation des profils de qualité dans un contrôle dynamique devrait produire des systèmes plus robustes. Néanmoins, l'approche interruptible fait des hypothèses assez fortes sur les propriétés algorithmiques requises. D'abord, un algorithme interruptible doit pouvoir produire un résultat même partiel à tout moment. C'est rarement le cas dans les tous premiers instants. Le modèle de tâche imprécise semble plus réaliste : une partie obligatoire est réservée pour produire un premier résultat de qualité acceptable. Ensuite, les profils de qualité sont sensés être acquis par une phase d'apprentissage qui produise des statistiques fiables du domaine cible. Nous verrons au chapitre 3 que c'est loin d'être acquis dans le domaine de l'optimisation combinatoire. Enfin, le contrôle dynamique suppose d'être capable de mesurer la qualité courante d'un résultat. C'est également une difficulté importante en optimisation combinatoire.

Les approches délibératives que nous avons présentées n'ont pas encore atteint un niveau industriel tel que celui exigé pour des systèmes temps réel embarqués complexes (déterministes, prédictibilité logique et temporelle). Il reste de nombreux problèmes à résoudre :

- avoir un modèle de tâche plus réaliste : prise en compte des contraintes sur l'utilisation des ressources physiques (mémoire et entrés/sorties en section critique) ; prise en compte d'un degré de priorité pour chaque tâche ;
- prendre en compte l'évolution dynamique de la situation qui nécessite de définir des algorithmes effectuant un raisonnement incrémental capable de réutiliser les résultats précédents lorsque la situation est modifiée ;
- prendre en compte l'incertitude sur les données des problèmes à traiter ; difficultés de la partie perception qui extrait les données ;
- avoir une prédictibilité logique et temporelle, c-à-d pouvoir garantir les temps de réponse pour prédire le respect ou non d'une échéance temporelle et aussi pouvoir évaluer précisément la durée des parties obligatoires (obtention d'un premier résultat acceptable).

Dans la suite de cette thèse, nous étudierons comment des algorithmes d'optimisation combinatoire sous contraintes peuvent s'intégrer à l'approche interruptible. L'étude de ces problèmes d'optimisation porte sur un formalisme général qui ne contient que très peu d'informations sur le domaine. C'est pourquoi nous ne nous occuperons pas de la partie apprentissage et viserons uniquement à exhiber des moyens de calculer la qualité d'une solution partielle, ce qui représente déjà en soi un problème délicat.

Chapitre 2

Problèmes de Satisfaction de Contraintes Valuées

Les *Problèmes de Satisfaction de Contraintes Valuées*, notés VCSP, sont définis dans un formalisme développé au *CERT* [Sch94], et décrivent de façon générale des problèmes d'optimisation sous contraintes.

Le formalisme des VCSP est une extension à l'optimisation d'un autre formalisme plus ancien, celui des *Problèmes de Satisfaction de Contraintes*, notés CSP. Nous examinons dans ce chapitre le formalisme des VCSP, et des algorithmes capables de résoudre ces problèmes. À chaque fois, nous établissons le lien existant avec le cadre des CSP.

La section 2.1 décrit le formalisme des CSP puis celui des VCSP, ainsi que le vocabulaire et la terminologie associés. La section 2.2 présente des méthodes de renforcement de cohérence qui sont la clé du succès des algorithmes de résolution des CSP, et montre les difficultés pour étendre ces algorithmes dans le cadre des VCSP. La section 2.3 aborde les algorithmes de résolution proprement dits en les séparant en deux catégories, d'une part celle qui offre des garanties sur l'obtention d'une solution optimale si le temps n'est pas limité, et d'autre part celle qui n'offre pas ces garanties, mais En revanche produit de meilleurs solutions en temps limité. Enfin, nous montrons dans la section 2.4 les principales difficultés propres à l'optimisation, à partir de résultats expérimentaux sur des problèmes générés aléatoirement.

2.1 Formalisme VCSP

Ce formalisme est une extension du champ d'application des CSP pour prendre en compte des notions telles que la préférence, l'imprécision ou l'incertitude. Cette extension se veut un cadre général qui peut être instancié en diverses sous-classes. La généricité a pour avantage de n'avoir à développer qu'une fois les techniques de résolution, indépendamment des sous-classes.

2.1.1 Formalisme général

2.1.1.1 Formalisme CSP

Nous abordons brièvement le langage et les principales notions associées aux CSP, initialement introduits dans [Mon74]. Le lecteur trouvera des présentations plus détaillées, notamment dans [Mac92, Tsa93].

Principales définitions

Définition 5
Problème de satisfaction de contraintes *[SFV97] :*
un problème de satisfaction de contraintes (en anglais, Constraint Satisfaction Problem *(CSP))* $P = (V, D, C)$ *est défini par :*

- *une séquence* $V = (x_1, \ldots x_n)$ *de* n *variables ;*
- *une séquence* $D = (d_1, \ldots, d_n)$ *de* n *domaines finis pour les variables de* V *; à chaque variable* x_i *est associée le domaine* $D(x_i) = d_i$ *;*
- *une séquence* $C = (c_1, \ldots, c_e)$ *de* e *contraintes ; chaque contrainte* c_i *est définie par un couple* (v_i, r_i) *:*
  - v_i *est une séquence de variables* $(x_{i_1}, \ldots, x_{i_{n_i}}) \subseteq V$ *sur lesquelles s'applique* c_i *; la longueur* n_i *de la séquence* v_i *est appelée arité de* c_i *;*
  - r_i *est une relation définie par un sous-ensemble du produit cartésien* $d_{i_1} \times \ldots \times d_{i_{n_i}}$ *des domaines associés aux variables de* v_i *; elle représente les n-uplets de valeurs autorisés pour ces variables.*

Remarque 1 Un CSP **binaire** est un CSP dont toutes les contraintes sont d'arité égale à 2. Dans le cas contraire, nous parlons de CSP **naire** . Lorsque le CSP est binaire (resp. naire), le couple (V, C) peut être vu comme un **graphe** (resp. **hyper-graphe**) non orienté où les sommets sont les variables et les arêtes (resp. hyper-arêtes) sont les contraintes. $\diamond$

Remarque 2 Le **degré** d'une variable correspond au nombre de contraintes qui pèsent sur cette variable. La **dureté** d'une contrainte $c_i = (v_i, r_i)$ (en anglais, *constraint tightness*), notée t, est définie par le rapport entre le nombre de n-uplets interdits $(|d_{i_1} \times \ldots \times d_{i_{n_i}}| - |r_i|)$ sur le nombre de n-uplets possibles $|d_{i_1} \times \ldots \times d_{i_{n_i}}|$. La **connectivité** d'un graphe de contraintes binaires (en anglais, *graph connectivity*), notée c, est définie par le rapport entre le nombre de contraintes $|C|$ sur le nombre maximum de contraintes $\frac{|V| * (|V|-1)}{2}$, correspondant à un graphe complet. $\diamond$

Définition 6
Instanciation *:*

- *une* **instanciation** A *d'un sous-ensemble* $Y = \{x_{y_1}, \ldots, x_{y_{n_y}}\}$ *des variables de* V *est une application qui affecte à chaque variable* x_{y_i}*, une valeur* $A(x_{y_i}) = val_{y_i} \in d_{y_i}$ *; une instanciation peut aussi s'écrire en extension par un ensemble d'affectations de variable* $\{x_{y_1} \mapsto val_{y_1}, \ldots, x_{y_{n_y}} \mapsto val_{y_{n_y}}\}$ *;*
- *la projection d'une instanciation sur un sous-ensemble* X *de variables se note* $A(X)$ *; c'est une restriction de* A *aux variables de* $Y \cap X$ *;*
- *une* **instanciation complète** *est une affectation de l'ensemble des variables ; dans le cas contraire, nous parlons d'***instanciation partielle** .

Définition 7 ———
Satisfaction :

- *une instanciation A de Y satisfait une contrainte $c_i = (v_i, r_i)$, telle que $v_i \subseteq Y$, si $A(v_i)$ appartient à la relation r_i ; sinon la contrainte est dite insatisfaite ou violée par A ;*
- *une instanciation A de Y est dite* **cohérente** *si elle satisfait toutes les contraintes c_i telles que $v_i \subseteq Y$; sinon elle est dite* **incohérente** *;*
- *une* **solution** *S de $P = (V, D, C)$ est une instanciation complète qui satisfait toutes les contraintes de C ;*
- *un* CSP *qui a au moins une solution est dit* **cohérent** *ou* **satisfiable** *; il est dit* **incohérent** *ou* **insatisfiable** *sinon.*

Résolution d'un CSP

Il existe plusieurs requêtes classiques pour un problème CSP :

1. le problème est-il satisfiable ?
2. trouver une solution au problème ;
3. combien de solutions possède le problème ?
4. trouver toutes les solutions du problème.

La requête 1 de satisfiabilité définit un problème de décision NP-complet. Nous choisissons de parler de **résolution** d'un CSP comme étant le fait de donner une réponse aux deux premières requêtes : "trouver une solution au problème , ou prouver qu'il n'y en a pas".

Langage de représentation

Le modèle CSP est très souple sur la description des valeurs et des contraintes. Les valeurs que peuvent prendre les variables sont de type quelconque, numérique ou symbolique. Pour une contrainte $c_i = (v_i, r_i)$, la relation r_i peut être codée par l'ensemble des n-uplets permis ; elle est dite "définie en extension". Mais il est aussi possible de coder la relation r_i par une fonction booléenne prenant en entrée les valeurs affectées aux variables v_i de la contrainte c_i ; la relation est dite "définie en intension" (par exemple, une fonction analytique comme l'égalité des valeurs en entrée).

La figure 2.1 représente le graphe d'un CSP binaire, *un problème d'habillage d'un robot féminin* [FW92]. Les noms des variables sont en gras, avec leurs domaines associés écrits en dessous. Le problème comporte trois contraintes définies en extension par une liste de couples de valeurs permis, indiqués en italique.

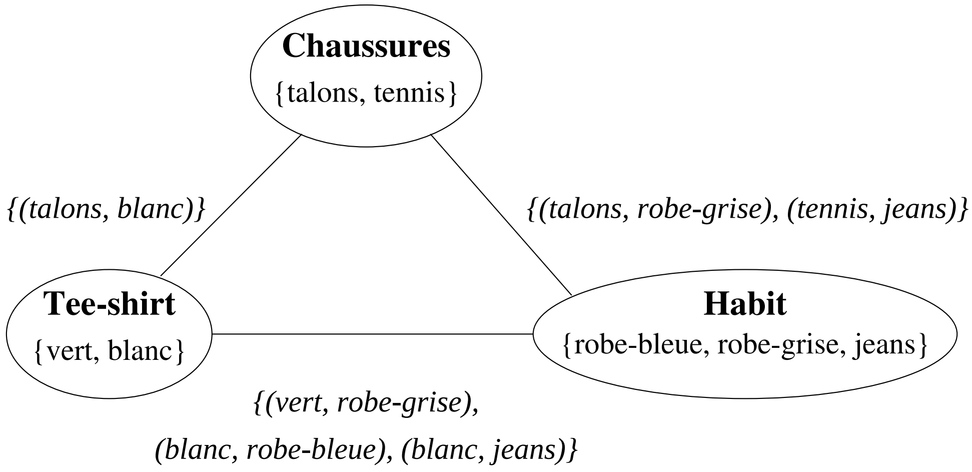


Fig. 2.1 - Exemple de CSP : un problème d'habillage de robot

Vers l'optimisation

Lorsque nous transformons un problème combinatoire du monde réel dans le modèle CSP, deux types de contraintes sont à distinguer :

- **contraintes dures** : elles représentent souvent des propriétés physiques du problème et parfois des préférences qu'il faut absolument respecter ;
- **contraintes molles** : elles correspondent à des contraintes de préférence qu'il est préférable de satisfaire mais ce n'est pas obligatoire.

Une approche simpliste consiste à ajouter les contraintes molles directement dans le CSP. Si le CSP est satisfiable, nous obtenons bien une solution optimale ; sinon, il y a absence de solution. Une autre approche consiste à ignorer le moins possible de contraintes molles pour retrouver une instanciation complète satisfiable.

Dans l'exemple du robot, il n'y a pas de solution satisfaisant toutes les contraintes ; En revanche, si la contrainte {*(talons, blanc)*} entre les variables *Chaussure* et *Tee-shirt* est retirée, alors la solution {*Chaussure* $\mapsto$ *tennis, Habit* $\mapsto$ *jeans, Tee-shirt* $\mapsto$ *blanc*} est satisfaisante.

Les VCSP sont décrits dans un formalisme général permettant d'exprimer et de manipuler des contraintes dures et des contraintes molles.

2.1.1.2 Formalisme VCSP

Le formalisme VCSP [Sch94, SFV95, SFV97] consiste à ajouter à un CSP une structure et une fonction de valuation.

Principales définitions

Définition 8
Structure de valuation *:*
la structure de valuation S est un quintuplet $(E, \succ, \top, \bot, \otimes)$ tel que :

- *E est un ensemble de valuation totalement ordonné par $\succ$, comprenant un élément maximum $\top$ et un élément minimum $\perp$;*
- *E est muni d'une loi de composition interne commutative et associative notée $\otimes$, appelée opérateur d'agrégation, qui vérifie*[1] *:*
  - *monotonie : $\forall a, b, c \in E,\ a \succeq c \Rightarrow (a \otimes b) \succeq (c \otimes b)$;*
  - *élément absorbant : $\forall a \in E,\ a \otimes \top = \top$;*
  - *élément neutre : $\forall a \in E,\ a \otimes \perp = a$.*

Définition 9
Fonction de valuation *:*
la fonction de valuation φ associe à chaque contrainte c_i une valuation $\varphi(c_i) \in E$ qui dénote l'importance de la contrainte.

Définition 10
Problème de satisfaction de contraintes valuées *:*
un problème de satisfaction de contraintes valuées (en anglais, Valued Constraint Satisfaction Problem *(*VCSP*)) est défini par un quintuplet $P = (V, D, C, S, \varphi)$, V étant l'ensemble des variables, D l'ensemble des domaines, C l'ensemble des contraintes, S la structure de valuation et φ la fonction de valuation.*

Définition 11
Valuation *:*

- *la* **valuation d'une instanciation complète** *A, notée $\varphi(A, P)$, est le résultat de l'agrégation des valuations des contraintes insatisfaites par A dans P, notées $C_{unsat}(A, P) \subset C$:*

$$\varphi(A, P) = \bigotimes_{c \in C_{unsat}(A,P)} \varphi(c)$$

- *la* **valuation globale d'une instanciation partielle** *A', notée également $\varphi(A', P)$, est la valuation minimum de toutes les extensions complètes de A'.*

Remarque 3 Le calcul de la valuation d'une instanciation complète est polynomial (en supposant que les contraintes puissent être testées en temps polynomial) ; alors que le calcul de la valuation globale d'une instanciation partielle est exponentiel. $\diamond$

Définition 12
Optimum *:*

- *une* **solution** *de $P = (V, D, C, S, \varphi)$ est une instanciation complète de valuation $\neq \top$;*

[1] S est une structure de demi-groupe ; l'existence d'un opposé transformerait S en une structure de groupe et permettrait l'expression de contraintes dont la violation est souhaitée.

- *une* **solution optimale** *de $P = (V, D, C, S, \varphi)$ est une instanciation complète de valuation $\neq \top$ optimale (c.-à-d. qui soit de valuation minimale) ;*
- *$\varphi(P)$ dénote la valuation d'une solution optimale ; il est appelé* **optimum** *de P.*

Résolution d'un VCSP

La requête habituelle pour un problème VCSP consiste à produire une solution optimale. Nous choisissons de parler de **résolution** d'un VCSP comme étant le fait de donner une réponse à la requête "trouver une solution optimale, ou prouver qu'il n'y en a pas" (dans ce cas, les contraintes dures ne peuvent pas toutes être satisfaites). C'est un problème NP-dur.

Sémantique des valuations

L'ensemble E des valuations est utilisé pour définir une violation graduelle des contraintes et de l'incohérence. L'opérateur $\succ$ définit un ordre total sur les valuations de E. L'élément maximum $\top$ définit la violation d'une contrainte dure ; il indique donc l'incohérence complète d'une instanciation. L'élément minimum $\perp$ définit la satisfaction d'une contrainte ; il indique la cohérence complète d'une instanciation.

L'opérateur d'agrégation $\otimes$ permet de combiner les valuations des contraintes insatisfaites d'une instanciation. La propriété de monotonie garantit qu'une instanciation qui satisfait un VCSP donné recevra une valuation plus faible qu'une instanciation qui viole une ou plusieurs contraintes. L'élément $\top$ est absorbant ; cela garantit qu'une instanciation est jugée totalement incohérente dès qu'une contrainte dure est violée. L'élément $\perp$ est neutre ; en effet, la prise en compte d'une contrainte satisfaite ne modifie en rien la valuation d'une instanciation.

Un cadre algébrique très proche de celui des VCSP est défini dans [BMR95]. La différence repose principalement sur le fait que seul un ordre partiel est établi pour les valuations. Une comparaison de ces deux cadres est faite dans [BFM+96].

Le formalisme VCSP est un cadre générique où s'instancient plusieurs sous-cadres, suivant le choix de l'opérateur d'agrégation $\otimes$ et le sens accordé aux valuations.

2.1.2 Classes de VCSP

Avant de présenter quelques classes de VCSP, nous avons besoin de définir d'autres propriétés pour l'opérateur d'agrégation.

2.1.2.1 Autres propriétés de l'opérateur d'agrégation

Deux autres propriétés importantes pour l'opérateur d'agrégation $\otimes$ sont à distinguer :

- **monotonie stricte** : $\forall a, b, c \in E$, si $(a \succ c)$ et $(b \neq \top)$ alors $(a \otimes b) \succ (c \otimes b)$; c'est une propriété qui parait "raisonnable" ; elle garantit que toute diminution locale du nombre de contraintes molles violées d'une instanciation contribue à une amélioration globale de la valuation de cette instanciation (dans le cas où il n'y a pas de contraintes dures violées) ;

- **idempotence** : $\forall a \in E$, $a \otimes a = a$; aucune distinction n'est faite entre une instanciation violant une contrainte de valuation a et une autre violant plusieurs contraintes de même valuation a (ce comportement est appelé "*effet de noyade*") ; c'est un comportement classique dans le cadre des CSP.

Remarque 4 L'idempotence et la monotonie stricte sont des propriétés incompatibles dès que $|E| > 2$. Il n'existe qu'un seul opérateur d'agrégation idempotent, l'opérateur max. $\diamond$

2.1.2.2 Classes

Différentes classes de VCSP ont été proposées, elles permettent d'instancier dans le cadre VCSP quelques cadres principaux qui ont déjà été proposés par ailleurs dans la littérature :

- classe des VCSP **classiques** , notée $\wedge$-VCSP ; l'objectif consiste à satisfaire toutes les contraintes ; cette classe est équivalente aux CSP ; l'opérateur d'agrégation idempotent est le *et-logique* (identique au max pour $E = \{\perp, \top\}$) ;

- classe des VCSP **possibilistes** [Sch92] et des VCSP **flous** [DFP93], notée *max*-VCSP, qui associe aux contraintes une priorité ; l'objectif consiste à minimiser la priorité maximale des contraintes violées ; l'opérateur d'agrégation idempotent est le *max* ; par exemple, dans un problème d'appariement d'image qui consisterait à retrouver dans une banque d'image celle qui correspond le mieux à l'image recherchée, le critère d'appariement doit être plus sensible à une grosse différence (contrainte de forte priorité violée) qu'au cumul de plusieurs petites différences ; c'est précisément le rôle de l'opérateur *max* ; par ailleurs, les VCSP flous sont une extension des VCSP possibilistes ; une contrainte floue est représentée par un ensemble de n-uplets autorisés avec un certain degré d'acceptation (égal à la priorité associée au n-uplet, cf. 2.1.3.2) par la contrainte ;

- classe des VCSP **additifs** [FW92], notée Σ-VCSP, qui associe aux contraintes un coût ; l'objectif consiste à minimiser la somme des coûts des contraintes violées ; l'opérateur d'agrégation monotone strict est la *somme* ; c'est le cadre le plus courant en optimisation sous contraintes ; dans l'exemple du robot (cf. figure 2.1), les contraintes expriment des préférences aux quelles nous associons un coût de 1 ; la solution $\{$*Chaussure* $\mapsto$ *tennis, Habit* $\mapsto$ *jeans, Tee-shirt* $\mapsto$ *blanc*$\}$ est optimale et a un coût de 1 (la contrainte $\{$*(talons, blanc)*$\}$ est insatisfaite) ;

- classe des VCSP **lexicographiques** [FDP95], notée *lex*-VCSP ; elle associe à chaque contrainte une priorité l_i, ou plutôt, un **multi-ensemble** [2] formé du singleton $\{l_i\}$; l'objectif consiste à minimiser le nombre de contraintes insatisfaites de priorité maximale, puis, à nombre égal, minimiser le nombre de contraintes insatisfaites de priorité inférieure, et ainsi de suite jusqu'à la priorité la plus faible ; l'opérateur d'agrégation monotone strict est l'*union de multi-ensemble* et l'ensemble des valuations est ordonné par la comparaison lexicographique entre multi-ensembles ($\succ^*$) ; la classe *lex*-VCSP peut être vue comme une amélioration de la classe *max*-VCSP car elle considère non seulement le maximum de valuation des contraintes violée, mais aussi le nombre de contraintes violées à cette priorité maximale, puis successivement aux priorités inférieures ; la classe *lex*-VCSP peut aussi être vue comme une amélioration de la classe Σ-VCSP à laquelle serait ajouté une hiérarchie sur les contraintes (optimisation en additif, d'abord sur la hiérarchie la plus importante, ensuite sur des hiérarchies de moins en moins importantes) ; l'opérateur lexicographique

[2] Un multi-ensemble est un ensemble qui peut contenir plusieurs fois le même élément.

est également utilisé dans le cadre de la *Programmation logique avec une hiérarchie de contraintes* [BMMW89, WB93].

La figure 2.2 résume la structure de valuation utilisée par chacune de ces classes.

Classe	Notation	E	$\succ$	$\perp$	$\top$	$\otimes$	Prop.
CSP classiques	$\wedge$-VCSP	{vrai, faux}	faux $\succ$ vrai	vrai	faux	$\wedge$	idem.+m. strict
VCSP possibilistes	max-VCSP	$\overline{\mathbb{N}}$	$>$	0	$+\infty$	max	idem.
VCSP lexicographiques	lex-VCSP	$\overline{\mathbb{N}}^*$	$>^*$	$\emptyset$	$\{+\infty\}$	$\cup$	m. strict
VCSP additifs	Σ-VCSP	$\overline{\mathbb{N}}$	$>$	0	$+\infty$	$+$	m. strict

Fig. 2.2 - Différentes classes de VCSP

$\overline{\mathbb{N}}$ désigne l'ensemble des entiers naturels auquel on a ajouté l'élément $+\infty$. $\overline{\mathbb{N}}^*$ désigne l'ensemble des multi-ensembles d'éléments de $\overline{\mathbb{N}}$. $>^*$ désigne l'opérateur de comparaison lexicographique[3] sur ce multi-ensemble.

D'un point de vue efficacité algorithmique, il apparaît que les propriétés de l'opérateur d'agrégation jouent un rôle essentiel. Nous verrons dans la section 2.2 que si l'opérateur est idempotent (cas des VCSP possibilistes, et bien évidement des CSP classiques), les techniques de renforcement de cohérence utilisées dans le cadre des CSP classiques peuvent être étendues. S'il ne l'est pas (cas des VCSP lexicographiques et additifs), ces techniques ne peuvent pas être aussi facilement étendues.

Ceci est confirmé dans le chapitre 4 par des résultats expérimentaux qui montrent une nette différence de complexité entre les VCSP possibilistes, d'une part, et les VCSP lexicographiques et additifs, d'autre part. Malheureusement, nombre de problèmes du monde réel font appel à des VCSP additifs.

2.1.3 Critique du formalisme VCSP

La critique porte d'une part sur une comparaison du formalisme VCSP par rapport à une définition très générale des problèmes d'optimisation. D'autre part, nous donnons un aperçu de certains manques ou particularités dans le formalisme pour traiter concrètement des problèmes d'optimisation qui ont abouti à développer des modèles alternatifs.

2.1.3.1 Notions générales d'optimisation

Définition 13 ———————————————————————————

Problème d'optimisation *[AL97] :*
chercher un état u^* élément d'un ensemble E, appelé espace d'état, qui soit minimal vis à vis d'une fonction f définie sur E et à valeurs dans l'ensemble des réels R , appelée fonction objectif, de telle façon que :

$$u^* \in E \quad / \quad f(u^*) = \min_{u \in E} f(u)$$

[3]Soit E un ensemble totalement ordonné par $\succ$. Soit E^* l'ensemble des multi-ensembles d'éléments de E. L'ordre $\succ^*$ sur E^*, induit par $\succ$, peut être défini de la façon suivante : soient $a, b \in E^*$, soit m l'élément maximum de $a \cup b$ selon $\succ$, soient a_m et b_m les sous-ensembles de a et b constitués des éléments égaux à m, $a \succ^* b \Leftrightarrow [|a_m| > |b_m|] \vee [(|a_m| = |b_m|) \wedge (a - a_m \succ^* b - b_m)]$.

u^* est une solution optimale du problème et $f(u^*)$ est l'optimum.

Cela devient un problème d'**optimisation combinatoire** lorsque l'espace d'état E est fini. De plus, s'il y a des contraintes qui réduisent l'ensemble des solutions potentielles E à un sous-ensemble de E, alors c'est un problème d'**optimisation sous contraintes** .

Le cadre VCSP est un cadre opératoire pour l'optimisation sous contraintes tel que l'espace d'état soit $d_1 \times \ldots \times d_n$, la fonction objectif soit $f(u) = \otimes_{c \in C_{unsat}(u,P) \wedge \varphi(c) \prec \top} \varphi(c)$, et les contraintes qui restreignent l'espace d'état soient les contraintes dures $\{c/\varphi(c) = \top\}$.

2.1.3.2 Modèles alternatifs

Différentes fonctions de valuation

Nous avons vu précédemment que les valuations sont appliquées aux contraintes par la fonction φ. D'autres modèles considèrent des valuations associées à des n-uplets, ou à des variables. Examinons dans ces trois modèles comment se calcule la valuation d'une instanciation :

- valuation sur les contraintes ; formalisme appelé *Valued Constraints Constraint Satisfaction Problems* (VCCSP) ; identique au formalisme VCSP ;

$$\varphi(A, P) = \bigotimes_{c \in C_{unsat}(A,P)} \varphi(c)$$

- valuation sur les n-uplets de valeurs ; formalisme appelé *Valued Tuples Constraint Satisfaction Problems* (VTCSP) ; cela permet de faire varier la valuation d'une contrainte suivant l'affectation des variables ; une valuation $\varphi(Y, c_i)$ est associée à chaque n-uplet interdit $Y \in d_{i_1} \times \ldots \times d_{i_{n_i}} - r_i$ d'une contrainte $c_i = (v_i, r_i)$;

$$\varphi(A, P) = \bigotimes_{c_i \in C / c_i = (v_i, r_i)} \varphi(A(v_i), c_i)$$

- valuation sur les variables ; formalisme appelé *Valued Variables Constraint Satisfaction Problems* (VVCSP) [VB95] ; une instanciation partielle est évaluée en fonction des variables non instanciées ; une variable v non instanciée a une valuation égale à $\varphi(v)$; une variable instanciée à une valuation égale à $\perp$; il y a en plus des contraintes dures entre les variables ;

$$\varphi(A, P) = \bigotimes_{c \in C / \varphi(c) = \top} \varphi(c) \;\otimes \bigotimes_{v \in V / v \text{ non instanciée}} \varphi(v)$$

Il est toujours possible de transformer un problème valué sur les variables ou sur les n-uplets en un problème VCSP, valué sur les contraintes.

Dans le cas d'une valuation sur les n-uplets, il suffit de décomposer une contrainte en autant de sous-contraintes qu'il y a de n-uplets interdits de valuations différentes, chaque sous-contrainte ayant la valuation des n-uplets qu'elle interdit.

Dans le cas d'une valuation sur les variables, il suffit d'introduire par variable une valeur spéciale de rejet et une contrainte molle unaire interdisant cette valeur rejet avec une valuation égale à $\varphi(v)$.

VCSP dynamiques

Le formalisme VCSP suppose une connaissance complète et figée du problème. Il fait l'hypothèse que l'environnement extérieur, dont est issu le problème d'optimisation, est statique. La notion de CSP **dynamique** [DD88] considère l'évolution d'un problème CSP par l'ajout ou le retrait de contraintes au cours du temps. Entre chaque modification des contraintes, la résolution est faite de manière statique. Des informations pertinentes, appelées *Nogoods* (instanciations partielles incohérentes), sont apprises et réutilisées durant les résolutions successives [SV94, VS94, VS95] . Une extension du modèle et des algorithmes pour l'optimisation de problèmes dynamiques dans le cadre des VCSP **dynamiques** est développée dans [Dag97, DV96].

Sémantique pour les valeurs et les variables : modèle objet

Du fait d'avoir un formalisme très général, la représentation d'un problème en CSP est une structure à plat qui ne contient aucune information sur une quelconque hiérarchie fonctionnelle entre les éléments du modèle, ni aucune indication sur la signification des éléments vis à vis du problème réel. Ces informations sont essentielles pour la mise au point d'heuristiques pertinentes, ainsi que l'optimisation du code.

Une solution consiste à encapsuler le formalisme VCSP dans un modèle à objets qui associe une classe différente aux éléments variable, valeur, contrainte et structure de valuation. Suivant le type de problème concret à résoudre, une liste d'attributs dépendant du domaine d'application est ajoutée à chaque classe. Ces attributs peuvent renseigner diverses heuristiques *ad-hoc*.

2.2 Méthodes de renforcement de cohérence

La principale notion introduite dans cette section est la propriété de cohérence locale. Les méthodes de renforcement de cohérence transforment un problème en un problème équivalent ayant atteint un niveau de cohérence locale choisi par la méthode. Le renforcement de cohérence permet de simplifier la résolution.

Nous présentons une propriété de cohérence locale, la k-cohérence forte, dans le cadre CSP où les techniques de renforcement de cohérence ont été initialement développées. Puis nous exhibons des problèmes CSP particuliers qui peuvent être résolus polynomialement une fois appliqué un renforcement de cohérence locale.

Nous verrons ensuite quelles difficultés surgissent pour étendre les notions de cohérence locale, ainsi que les algorithmes de renforcement de cohérence, dans le cadre des VCSP.

2.2.1 Dans le cadre CSP

2.2.1.1 Propriétés de cohérence locale

Nous définissons d'abord la propriété de k-cohérence, qui est aussi une propriété de cohérence locale :

Définition 14 ———————————————————————————
k-cohérence *[Fre78] :*

un CSP *est dit k-cohérent si pour tout sous-ensemble de k variables $(x_1, \ldots, x_k)$, pour toute instanciation A cohérente des $(k-1)$ variables $(x_1, \ldots, x_{k-1})$, il existe une valeur $v \in d_k$ telle que l'instanciation $A \cup \{x_k \mapsto v\}$ soit cohérente.*

Pour être réellement utile, cette propriété est renforcée, de façon récursive aux niveaux inférieurs :

Définition 15 ---
k-cohérence forte *[Fre78] :*
un CSP *est dit fortement k-cohérent ssi, $\forall i$, $1 \leq i \leq k$, il est i-cohérent.*

Remarque 5 Un CSP comprenant n variables et qui est fortement n-cohérent est dit **globalement cohérent** . ◇

Remarque 6 Pour un CSP qui est fortement k-cohérent, il existe une instanciation partielle cohérente quel que soit le sous-ensemble de k variables. Ainsi, un CSP globalement cohérent est satisfiable, i.e. il admet une solution. ◇

Le renforcement de cohérence consiste à transformer un CSP en un CSP simplifié équivalent (même ensemble de solutions) et qui soit k-cohérent fort. L'opération réalisant cette simplification est appelée fermeture par k-cohérence forte. Elle consiste à ajouter des contraintes induites d'arité maximale $k-1$ jusqu'à ce que le CSP simplifié vérifie la k-cohérence forte. Lors de la fermeture, il est possible de détecter si un CSP ne peut pas être rendu k-cohérent. Dans ce cas, le CSP est insatisfiable.

Pour un CSP comprenant n variables et des domaines de taille d, [Coo89] propose un algorithme établissant la fermeture par k-cohérence forte avec une complexité temporelle et spatiale polynomiale, en $O(n^k \times d^k)$.

2.2.1.2 Exemple : l'arc-cohérence pour les CSP binaires

Dans le cas des CSP binaires, des petits niveaux de k-cohérence ont été largement étudiés : l'**arc-cohérence** (2-cohérence forte, *arc-consistency* en anglais) et la **chemin-cohérence** (3-cohérence forte, *path-consistency* en anglais). Nous reformulons la propriété d'arc-cohérence :

Définition 16 ---
Arc-cohérence *:*
un CSP *binaire $P = (V, D, C)$ est arc-cohérent ssi pour toutes les variables x_i de V :*

- *le domaine d_i n'est pas vide (cela assure la 1-cohérence) ;*
- *toute valeur $v \in d_i$ peut se prolonger sur tout autre domaine d_j d'une variable x_j, reliée par une contrainte à x_i, en une instanciation cohérente de deux variables (cela assure la 2-cohérence) ;*

Plusieurs algorithmes, notés de façon génériques *ACx*, établissent l'arc-cohérence, le dernier en date étant *AC7* [BFR95] qui a une complexité temporelle en $O(e \times d^2)$ et spatiale en $O(e \times d)$, avec $e = |C|$, le nombre de contraintes du CSP et d, la taille de ses domaines. L'algorithme *AC-Inference* [Ŕ95] améliore encore les performances lorsque les contraintes ont certaines propriétés particulières : contraintes *irréflexives, réflexives, anti-commutatives, commutatives ou bidirectionnelles.*

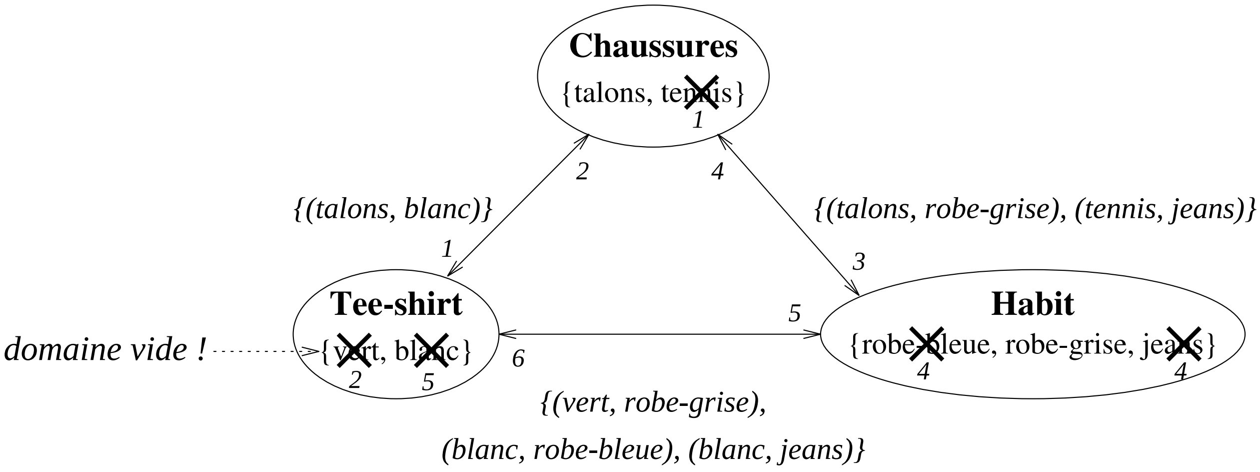


Fig. 2.3 - Renforcement d'arc-cohérence pour le problème du robot : CSP arc-incohérent

La figure 2.3 montre l'élimination de valeurs par application d'un renforcement d'arc-cohérence pour le problème du robot (cf. figure 2.1). L'algorithme employé est *AC3* qui maintient à jour une liste de contraintes binaires directionnelles[4]. Au début, la liste contient toutes les contraintes (notées de 1 à 6 dans la figure). *AC3* retire la première contrainte de la liste et vérifie si les valeurs de la première variable ont au moins un **support** [5] dans la deuxième variable. Par exemple, la valeur *tennis* n'a pas de support dans *Tee-shirt*, elle est supprimée lors de l'examen de la contrainte 1. Lorsqu'une valeur est supprimée, les contraintes qui pointent sur la variable en question nécessitent d'être re-examinées, elles sont rajoutées dans la liste. Ce mécanisme permet une **propagation des contraintes** non triviale. L'algorithme se termine quand la liste est vidée, ou lorsqu'un domaine devient vide (en anglais, *wipe out*), c'est le cas, après l'examen de la contrainte 5, pour le domaine de *Tee-shirt*. Un domaine vide signifie que le CSP ne peut pas être rendu arc-cohérent, donc qu'il est insatisfiable.

Au lieu de l'utiliser uniquement comme phase de pré-traitement, il est possible d'effectuer le filtrage par arc-cohérence durant la phase de recherche d'une solution ; plusieurs travaux démontrent l'intérêt de cette approche, appelée *Maintien d'Arc-Cohérence* (*MAC*) [SF94, Ŕ95, BR96].

Des formes affaiblies d'arc-cohérence sont définies pour réaliser durant la recherche un filtrage avec une complexité temporelle plus faible. En contrepartie, l'arc-cohérence n'est plus complète-

[4]Chaque contrainte binaire est examinée deux fois, suivant que le **filtrage des valeurs** s'applique à la première ou à la deuxième variable de la contrainte. C'est pourquoi, les contraintes sont doublement numérotées dans la figure. Par exemple, la contrainte entre *Chaussure* et *Tee-shirt* est numérotée 1, pour filtrer le domaine de *Chaussure*, et 2, pour filtrer le domaine de *Tee-shirt*.

[5]Un support d'une valeur $val_i \in D(x_i)$ pour une contrainte $C = (\{x_i, x_j\}, r)$ est une valeur $val_j \in D(x_j)$ telle que $(val_i, val_j) \in r$.

ment assurée. Un premier exemple d'affaiblissement est le *Directional Arc Consistency* [DP88] qui réalise un filtrage uniquement selon un ordre donné des variables. Suivant cet ordre, des valeurs sont supprimées si elles ne peuvent pas se prolonger sur des domaines d'autres variables intervenant *après* dans l'ordre. Un autre exemple très utilisé d'affaiblissement est le *Forward Checking* [HE80]. Il réalise un filtrage par arc-cohérence affaiblie car il ne considère que les contraintes quasi-instanciées, c'est à dire les contraintes dont toutes les variables sauf une sont instanciées.

2.2.1.3 Classes polynomiales de CSP

Une classe polynomiale définit un ensemble de problèmes qui peuvent être résolus en temps polynomial. Un exemple de classe polynomiale est l'ensemble des CSP globalement cohérents. En effet, d'après la définition 15, toute instanciation partielle cohérente d'un sous-ensemble des variables peut être étendue en une instanciation complète cohérente sans retour-arrière[6].

Suivant le type de contrainte, il suffit de vérifier la k-cohérence à un niveau k inférieur à n pour être sûr que le CSP est globalement cohérent. Nous donnons quelques définitions indiquant le type d'une contrainte binaire exprimée en extension[7] :

Définition 17

[vD94]
Une contrainte binaire est m-**tight** *(resp.* m-**loose** *) si chaque colonne et chaque ligne de la matrice* $\{0,1\}$ *représentant la contrainte a au plus (resp. au moins)* m *valeurs à 1.*

Une contrainte binaire est **row convex** *si dans chaque ligne de la matrice* $\{0,1\}$ *représentant la contrainte, les valeurs à 1 sont consécutives.*

Le théorème permettant de passer d'une cohérence locale à une cohérence globale est le suivant :

Théorème 6

[vD94]
Un CSP *binaire est globalement cohérent si*

- *toutes ses contraintes sont m-tight et il est fortement* $(m+2)$*-cohérent ;*
- *toutes ses contraintes sont m-loose, tous ses domaines sont de taille inférieure ou égale à d et il est fortement* $(\frac{d}{d-m})$*-cohérent ;*
- *tous ses domaines sont de taille inférieure ou égale à d et il est fortement* $(d+1)$*-cohérent ;*
- *il existe un ordre sur les domaines tel que toutes les contraintes soient row convex et le* CSP *est chemin-cohérent (3-cohérent).*

[6] Suivant le principe de retour en arrière (en anglais, *backtracking*) sur les choix d'affectation précédents lorsque l'instanciation partielle courante ne peut pas être étendue. Ce principe est à la base des méthodes exactes de génération de solution, cf. section 2.3.1.

[7] Pour une contrainte binaire exprimée en extension, la représentation peut être une matrice dont les indices de ligne et de colonne correspondent à une valeur dans les domaines ordonnés des variables de la contrainte. La matrice est composée de 0 ou de 1 suivant que le couple de valeurs est interdit ou permis.

[Dv95] étendent ces résultats aux CSP naires, en définissant une notion de m-cohérence relationnelle locale voisine de la notion de k-cohérence dont la localité est établie non pas sur un sous-ensemble de k variables mais sur un sous-ensemble de m contraintes.

Il existe d'autres classes polynomiales de CSP qui sont déterminées uniquement suivant la structure du graphe des contraintes. Pour un CSP binaire dont le graphe est un arbre, il existe un algorithme polynomial pour le résoudre. En effet, il suffit d'appliquer un algorithme d'arc-cohérence puis générer directement une solution sans retour-arrière.

2.2.2 Extension au cadre VCSP

L'extension des techniques de renforcement de cohérence nécessite de redéfinir la notion de cohérence locale dans le cadre valué. Nous examinons une extension de l'arc-cohérence dans les cas où l'opérateur d'agrégation est, ou n'est pas, idempotent.

2.2.2.1 Extension pour un opérateur d'agrégation idempotent

Définition 18
Arc-cohérence pour VCSP binaire idempotent *[SFV97] :*
un VCSP ayant un opérateur d'agrégation idempotent est arc-cohérent ssi :

- *il existe, pour toute variable, une valeur qui définit une instanciation de valuation locale inférieure à $\top$;*
- *toute instanciation d'une variable peut-être étendue sur n'importe quelle autre variable en une instanciation de deux variables ayant la même valuation.*

Il n'existe qu'un seul opérateur d'agrégation idempotent, c'est le *max*. Redéfinie pour cet opérateur dans le cadre *max*-VCSP, le renforcement d'arc-cohérence se traduit par l'ajout d'une contrainte unaire valuée induite $C_{unary}(v, val)$ pour chaque valeur val de chaque variable v. Le calcul de la valuation de ces contraintes induites, initialement défini dans [RHZ76], vaut :

$$\varphi(C_{unary}(v, val)) = \max_{\forall v' \in V}(\min_{\forall val' \in v'}(\varphi(\{v \mapsto val, v' \mapsto val'\})))$$

L'utilisation des contraintes induites permet de calculer un minorant, noté lb (en anglais, *lower bound*), de l'optimum d'un problème *max*-VCSP :

$$lb = \max_{\forall v \in V}(\min_{\forall val \in D(v)} \varphi(C_{unary}(v, val)))$$

Un algorithme polynomial réalisant un renforcement d'arc-cohérence pour un opérateur idempotent est décrit dans [Sch92] dans le cadre des VCSP possibilistes et est étendu au cadre des *Semiring-based Constraint Satisfaction Problems* dans [BMR95].

2.2.2.2 Extension pour un opérateur d'agrégation non idempotent

Le fait d'ajouter une contrainte à un VCSP non idempotent modifie en général la distribution de valuation de ses instanciations complètes. En revanche, ce n'est pas forcément le cas pour un opérateur idempotent. Le renforcement de cohérence décrit section 2.2.2.1 ne modifie pas la distribution de valuation d'un VCSP idempotent.

Dans le cadre d'un opérateur d'agrégation non idempotent, la notion de renforcement de cohérence se traduit par des mécanismes permettant de déduire un minorant de l'optimum en un temps polynomial, soit avant la recherche d'une solution optimale, soit pendant. Nous verrons dans les chapitres suivants que cet objectif de produire un minorant est au coeur de nos travaux.

Les résultats suivants montrent quelques tentatives d'extension de l'arc-cohérence aux VCSP non idempotents.

[FW92] définissent un renforcement d'arc-cohérence sans propagation. Ils calculent pour chaque valeur val de chaque variable v, une valuation, dite *Arc Consistency Count*, égale à l'agrégation de toutes les valuations des contraintes portant sur v et qui ne peuvent pas être satisfaites avec $\{v \mapsto val\}$. Chaque valuation est un minorant de l'optimum du sous-problème où val est affecté à v.

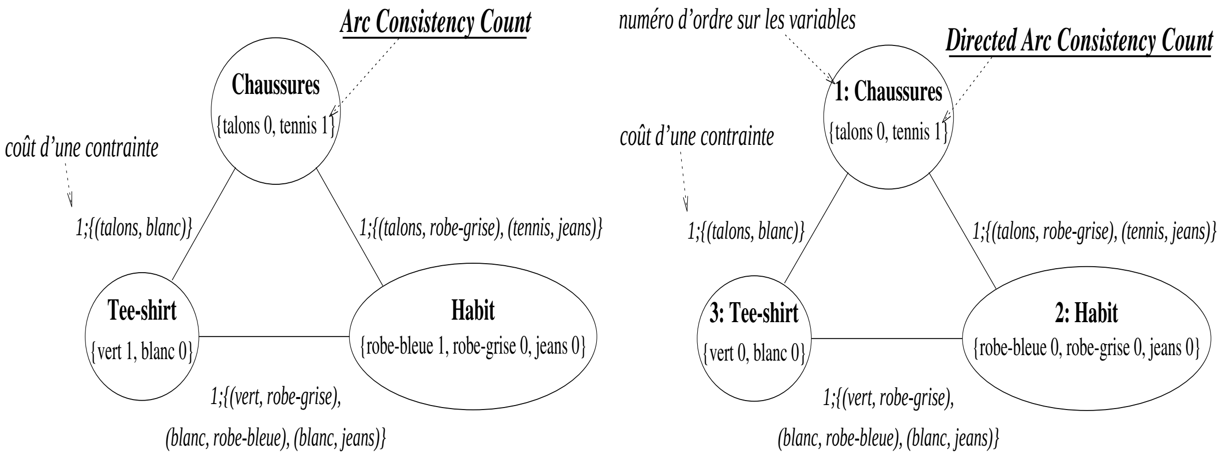


Fig. 2.4 - Renforcement d'arc-cohérence et d'arc-cohérence dirigée pour le problème du robot, dans le cas valué

Une forme affaiblie d'arc-cohérence, le *Directed Arc Consistency* correspond aussi à une valuation induite, à l'instar des *Arc Consistency Counts*, mais son calcul ne considère pour chaque variable que les contraintes binaires qui relient cette variable à une autre variable intervenant après en fonction d'un ordre des variables fixé. Cette technique est associée à un autre mécanisme de renforcement de cohérence, le *Forward Checking* valué [FW92] pour aboutir à un algorithme de recherche de solution assez performant [Wal94, Wal96b, LM95, LM96b] (voir section 2.3.1).

Enfin, une autre façon de définir le renforcement de cohérence consiste à chercher une relaxation optimale[8] dont la fermeture par arc-cohérence soit non vide.

[8] Une relaxation d'un problème consiste à lui retirer des contraintes. Une relaxation est optimale si elle minimise

Théorème 7

[SFV95]

Soit un VCSP binaire avec un opérateur d'agrégation monotone strict, et une valuation l ; alors le problème de décision "existe-t-il une relaxation du VCSP de valuation inférieure à l et qui ne soit pas arc-incohérente ?" est un problème NP-complet[9].

Ainsi, "il est aussi difficile (en terme de classe de complexité) de déterminer le nombre minimum de contraintes qu'il faut enlever pour restaurer l'arc-cohérence que de déterminer le nombre minimum de contraintes qu'il faut enlever pour restaurer la cohérence" [SFV95].

2.2.2.3 Classes polynomiales de VCSP

La majeure partie des classes polynomiales définies pour les CSP s'étendent bien aux VCSP ayant un opérateur idempotent. En effet, il existe une transformation simple pour résoudre un *max*-VCSP en une succession de résolutions de CSP classiques. Soit le VCSP possibiliste noté $P = (V, D, C, S, \varphi)$. Il suffit de résoudre successivement un CSP composé de V, D et C' tel que $C' = \{c \in C / \varphi(c) \geq l\}$, avec l, le niveau de valuation considéré. La résolution commence au premier niveau, où seules les contraintes de priorité maximale sont considérées : $l = \max_{c \in C} \varphi(c)$. Si la résolution du CSP correspondant trouve une solution, celle-ci est stockée, puis le processus est répété au niveau l' juste inférieur à l. Sinon, le processus s'arrête ; la valuation optimale de P est égale à la priorité du niveau courant ; le niveau précédent contient une solution optimale. Il serait aussi possible d'utiliser un parcours dichotomique des niveaux.

Dans le cas des VCSP non idempotents, des structures en hyper-arbre du graphe des contraintes permettent d'appliquer un algorithme polynomial [FW92], issu de la programmation dynamique [BB72], pour les résoudre. Il n'y a pour l'instant aucun résultat à notre connaissance sur l'existence d'autres classes polynomiales. Par exemple, [Sch94] démontre que déterminer s'il existe une instanciation de coût inférieur à l pour un problème Σ-VCSP binaire dont les domaines sont tous de taille inférieure ou égale à deux est NP-complet, alors qu'il est résolu polynomialement avec un renforcement par chemin-cohérence dans le cas d'un CSP (cf. théorème 6).

2.3 Méthodes de résolution

Nous présentons les deux grandes classes de méthodes pour résoudre un problème d'optimisation sous contraintes. Ce sont d'une part les méthodes complètes (cf. section 2.3.1) qui produisent une solution optimale et la preuve de son optimalité ; en contrepartie, le temps de résolution est exponentiel dans le pire cas. D'autre part, les méthodes incomplètes (cf. section 2.3.2) produisent généralement de bonnes solutions en un temps raisonnable ; mais sont incapables de garantir la qualité des solutions produites. Nous examinons section 2.3.3 des schémas de coopération des deux grandes classes de méthodes.

Les méthodes de résolution sont principalement étudiées dans le cadre des VCSP. Plus précisément, c'est la classe des Σ-VCSP qui est visée par les techniques décrites sections 2.3.2 et 2.3.3. Les problèmes Σ-VCSP sont choisis comme problèmes cibles pour l'application de techniques ANYTIME. Néanmoins, certaines méthodes de résolution sont directement issues des CSP.

la valuation des contraintes retirées.

[9] Une propriété supplémentaire de $\otimes$ est nécessaire pour établir la preuve de ce théorème : $\forall a, b \in E,\ a, b \prec \top \Rightarrow a \otimes b \prec \top$. Cette propriété est respectée dans les VCSP lexicographiques et additifs.

Dans ce cas, nous présentons, en le précisant dans le texte, leur application initiale au cadre CSP.

Dans les trois sections suivantes, nous décrivons plus en détail des extensions originales de méthodes existantes, développées durant la thèse. Dans la section 2.3.1, il s'agit des heuristiques dynamiques Γ et Ω, applicables aux Σ-VCSP. Dans la section 2.3.2, c'est un algorithme hybride combinant recuit simulé et descente en gradient. Enfin, dans la section 2.3.3, c'est une coopération étudiée dans le cadre CSP d'une méthode incomplète, le recuit en champs moyen, qui est exécuté de façon répétée par une méthode complète pour fournir au cours de la recherche des heuristiques dynamiques.

2.3.1 Méthodes complètes

2.3.1.1 Parcours arborescent

Les méthodes complètes assurent une exploration systématique de l'espace de recherche. Cet espace est construit de manière arborescente. Sa construction s'effectue de la racine aux feuilles. La racine de l'arbre de recherche correspond à un état initial où aucune variable n'est affectée. Une feuille correspond à un état d'arrêt où toutes les variables sont affectées. Un noeud de l'arbre correspond à l'affectation d'une valeur à une variable ; de chaque noeud, part $|d|$ branches correspondant aux $|d|$ valeurs possibles du domaine de la nouvelle variable à affecter. Enfin, un chemin de la racine à un noeud correspond à une instanciation partielle des variables.

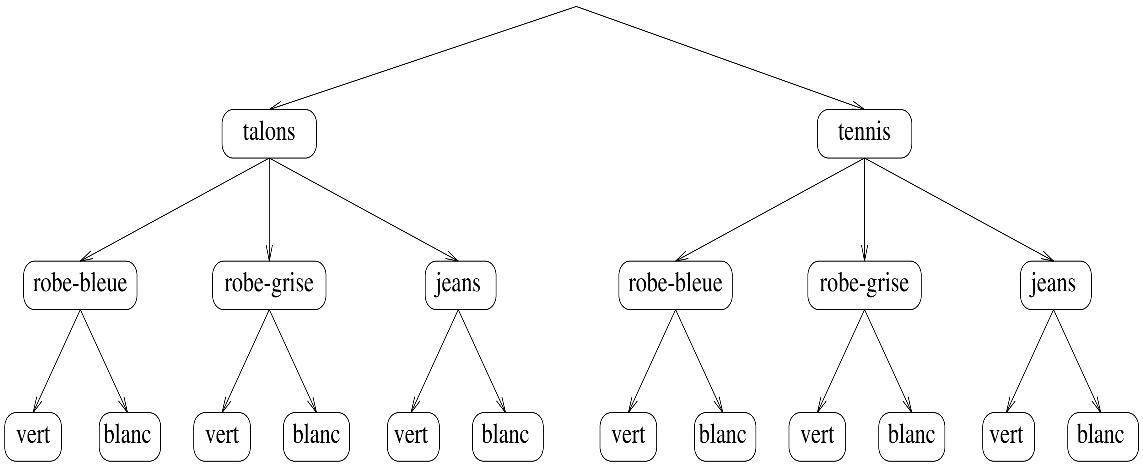


Fig. 2.5 - Arbre de recherche complet pour le problème du robot

La figure 2.5 représente l'arbre de recherche associé au problème de l'habillage de robot (cf. figure 2.1). Un ordre statique, c.-à-d. le même dans toutes les branches, a été utilisé pour l'ordre d'affectation des variables $(Chaussure, Habit, Tee-shirt)$, ainsi que pour l'ordre de sélection des valeurs (ici, les ordres initiaux des domaines). Des ordres dynamiques, c.-à-d. différents suivant le chemin déjà parcouru, peuvent aussi être pris ; nous en examinerons les avantages dans une section suivante.

Les méthodes complètes ont plusieurs façons de parcourir un arbre de recherche ; indépendamment des ordres statiques ou dynamiques :

- **parcours en profondeur d'abord** (en anglais, *depth first*) ; parcourt en premier les noeuds les plus à gauche et les plus profonds ; lorsque l'algorithme arrive sur un noeud d'arrêt (une feuille), il effectue un **retour-arrière** (en anglais, *backtrack*) au noeud précédant l'arrêt, puis repart en profondeur sur les valeurs suivantes de la variable d'arrêt, et si elles ont toutes été explorées, effectue à nouveau un retour-arrière ; il n'y a qu'un seul chemin (une seule instanciation partielle) à mémoriser à tout instant ;

- **parcours en largeur d'abord** (en anglais, *breath first*) ; parcourt toutes les valeurs de la première variable, puis de la seconde, etc.., jusqu'à la dernière variable ; il y a, à la profondeur i de l'arbre, $d^{(i-1)}$ chemins à mémoriser ;

- **parcours du "meilleur" en premier** (en anglais, *best first*) ; effectue un compromis entre recherche en largeur et recherche en profondeur ; maintient à jour une liste des *noeuds pendants*, qui correspond à la frontière entre les noeuds déjà développés et les noeuds à développer ; suivant un certain critère évaluant chaque noeud, l'algorithme développe un noeud parmi les noeuds pendants ayant la meilleure valeur du critère, à l'instar de l'algorithme A^* [AS94] ; il y a autant de chemins à mémoriser que de noeuds pendants.

Du fait de besoins en mémorisation trop importants des chemins déjà effectués (ou d'une partie seulement pour le *best first*), les parcours en largeur d'abord ou du "meilleur" en premier sont rarement utilisés pour la résolution des CSP comme des VCSP. Le parcours, utilisé comme base pour les algorithmes présentés par la suite, est donc celui en profondeur d'abord.

Un parcours complet de l'arbre de recherche implique de développer $\sum_{i=1}^{n} d^i = (d^n - d)/(d-1)$ noeuds pour un VCSP à n variables, avec des domaines de taille d. Cependant, plusieurs chemins de l'arbre ne conduisent pas à une solution. Pour tout problème VCSP, il est possible de calculer un minorant des valuations de toutes les instanciations complètes issues d'une instanciation partielle, il est appelé **minorant de la valuation globale d'une instanciation partielle** . Il suffit de calculer un minorant du noeud courant pour savoir s'il est nécessaire ou non d'examiner ses fils. Nous voyons dans la section suivante comment calculer un minorant.

2.3.1.2 Calcul d'un minorant de la valuation globale d'une instanciation partielle

Nous présentons quatre façons de calculer un minorant $lb(A, P)$ de la valuation globale $\varphi(A, P)$ d'une instanciation partielle A d'un problème $P = (V, D, C, S, \varphi)$. Le calcul d'un minorant $lb(A, P)$ consiste à agréger la valuation des contraintes qui sont violées directement par A ou qui seront sûrement violées quelle que soit l'extension de A en une instanciation complète. Dans le cadre des VCSP à opérateur d'agrégation monotone strict, il est important que les valuations des contraintes insatisfaites ne soient pas agrégées plusieurs fois ; de manière à être sûr que lb est bien un minorant de la valuation globale.

- *backward checking* $lb_{bc}(A, P)$; ne considère que les contraintes portant sur des variables toutes instanciées par A, l'ensemble de ces contraintes est noté $C_{unsat}(A, P)$:

$$lb_{bc}(A, P) = \bigotimes_{c \in C_{unsat}(A,P)} \varphi(c)$$

- *forward checking* $lb_{fc}(A, P)$ [FW92] ; considère en plus les contraintes quasi-instanciées par A, telles qu'une seule variable ne soit pas instanciée ; l'ensemble de ces contraintes est noté $C_{quasi}(A, P)$; soit $V'(A)$, l'ensemble des variables non instanciées par A, appelées aussi ensemble de **variables futures** ; soit $C_{quasiunsat}(A \cup \{v \mapsto val\}, P)$, l'ensemble des contraintes appartenant à $C_{quasi}(A, P)$, complètement instanciées et insatisfaites par $A \cup \{v \mapsto val\}$:

$$lb'_{fc}(A,P) = \bigotimes_{v \in V'(A)} [\min_{val \in D(v)} \bigotimes_{c \in C_{quasiunsat}(A \cup \{v \mapsto val\}, P)} \varphi(c)]$$

$$lb_{fc}(A,P) = lb_{bc}(A,P) \otimes lb'_{fc}(A,P)$$

- *directed arc-consistency* $lb_{dac}(A, P)$ [Wal94] ; considère aussi les contraintes non instanciées suivant un ordre statique imposé aux variables, utilise les valuations obtenues par un renforcement d'arc-cohérence dirigée (cf. section 2.2.2) ; soit $C_{afterunsat}(\{v \mapsto val\}, P)$, l'ensemble des contraintes portant sur des variables de rang supérieur ou égal à v et qui ne peuvent pas être satisfaites avec $\{v \mapsto val\}$:

$$lb'_{dac}(A,P) = \bigotimes_{v \in V'(A)} [\min_{val \in D(v)} ((\bigotimes_{c \in C_{quasiunsat}(A \cup \{v \mapsto val\}, P)} \varphi(c)) \otimes (\bigotimes_{c \in C_{afterunsat}(\{v \mapsto val\}, P)} \varphi(c)))]$$

$$lb_{dac}(A,P) = lb_{bc}(A,P) \otimes lb'_{dac}(A,P)$$

- *Russian Doll Search* $lb_{rds}(A, P)$ [LV96] ; impose aussi un ordre statique sur les variables ; nécessite de connaître, par une résolution précédente, l'optimum du sous-problème $P'(A)$ composé des variables non affectées par A, noté $\varphi(P'(A))$;

$$lb_{rds}(A,P) = lb_{bc}(A,P) \otimes lb'_{fc}(A,P) \otimes \varphi(P'(A))$$

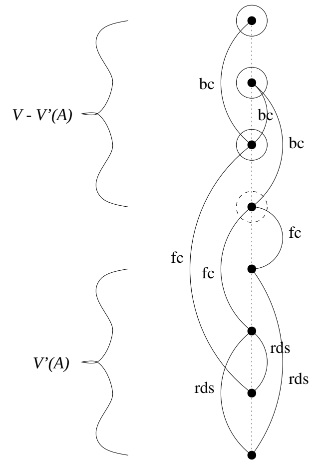


Fig. 2.6 - Répartition des contraintes pour le calcul du minorant $lb_{rds}(A, P)$

Quelle que soit A, nous avons les relations suivantes qui ordonnent les minorants du moins précis au plus précis : $lb_{bc}(A,P) \preceq lb_{fc}(A,P) \preceq lb_{dac}(A,P)$ et $lb_{bc}(A,P) \preceq lb_{fc}(A,P) \preceq lb_{rds}(A,P)$. En effet, la construction de ces minorants est une combinaison de minorants élémentaires indépendants. Par exemple, $lb_{rds}(A, P)$ combine les informations du *backward checking* lb_{bc}, du *forward checking* pur lb'_{fc} et de l'optimum du sous-problème $P'(A)$. Lorsque les contraintes sont toutes binaires, la décomposition du minorant lb_{rds} permet de prendre en compte toutes les contraintes du problème mais de façon indépendante[10], voir la figure 2.6. En revanche, nous ne pouvons pas combiner les *Directed Arc-Consistency Counts* avec $\varphi(P'(A))$, car ces deux valuations peuvent considérer les même contraintes.

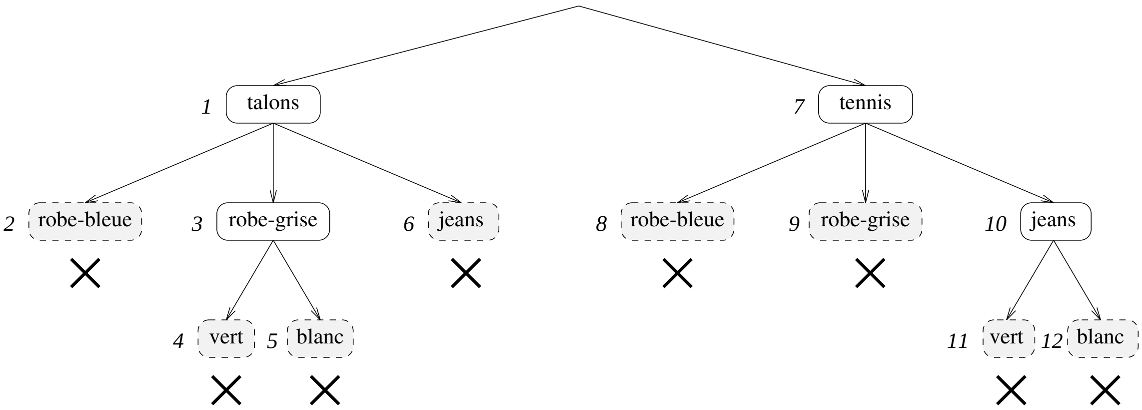


Fig. 2.7 - Arbre de recherche développé par l'algorithme *Backtrack*

Dans le cas des CSP classiques, le minorant d'une instanciation partielle vaut soit $\perp$, dans ce cas les fils du noeud correspondant seront développés, soit $\top$, dans ce cas c'est un noeud d'arrêt, il est coupé. Ainsi, un noeud d'arrêt peut être une feuille, c'est alors une solution du CSP, ou peut être une impasse (le sous-arbre ayant comme racine ce noeud ne contient aucune solution). La figure 2.7 montre l'arbre de recherche associé au problème du robot tel qu'il est effectivement développé par un méthode complète de parcours en profondeur d'abord avec un minorant calculé par *backward checking* ; cette méthode est appelée **algorithme** *Backtrack* . Les numéros en italique indiquent l'ordre d'exploration des noeuds ; une croix en dessous d'un noeud signifie que le minorant de ce noeud vaut $\top$.

Dans le cas général des VCSP, trouver une instanciation complète n'est pas suffisant, il faut aussi qu'elle soit de valuation minimum. L'extension de l'algorithme *Backtrack* au cas valué, appelé **algorithme** *Depth First Branch and Bound* , consiste à ajouter un paramètre supplémentaire, un majorant de l'optimum du problème. Ce majorant (en anglais, *upper bound*), noté ub, est initialisé à $\top$, puis diminue à chaque fois qu'une nouvelle instanciation complète A est trouvée : $ub = \varphi(A, P)$. Il permet de couper un noeud dont le minorant serait supérieur ou égal au majorant courant. De plus, à la fin du parcours de l'arbre de recherche, la valeur du majorant correspond à l'optimum $\varphi(P)$. Pour le problème d'habillage d'un robot, transformé en un Σ-VCSP où les contraintes ont un coût de 1, la figure 2.8 décrit l'arbre de recherche développé par l'algorithme *Depth First Branch and Bound* avec un calcul de minorant par *backward checking*. Les évolutions du minorant et du majorant sont indiquées, ainsi que l'ordre d'exploration des noeuds. Une croix dénote une impasse, lorsque $lb_{bc} \geq ub$. L'algorithme trouve deux instanciations complètes ; la deuxième est une solution optimale du problème, de valuation égale à 1 : $\{$*Chaussure* $\mapsto$ *talons, Habit* $\mapsto$ *robe-bleue, Tee-shirt* $\mapsto$ *blanc*$\}$.

[10] $lb_{rds}(A, P)$ reste un minorant des extensions complètes de l'affectation partielle A car l'optimum de $P'(A)$, $\varphi(P'(A))$, est calculé indépendamment des choix des valeurs affectées par A aux variables $V - V'(A)$.

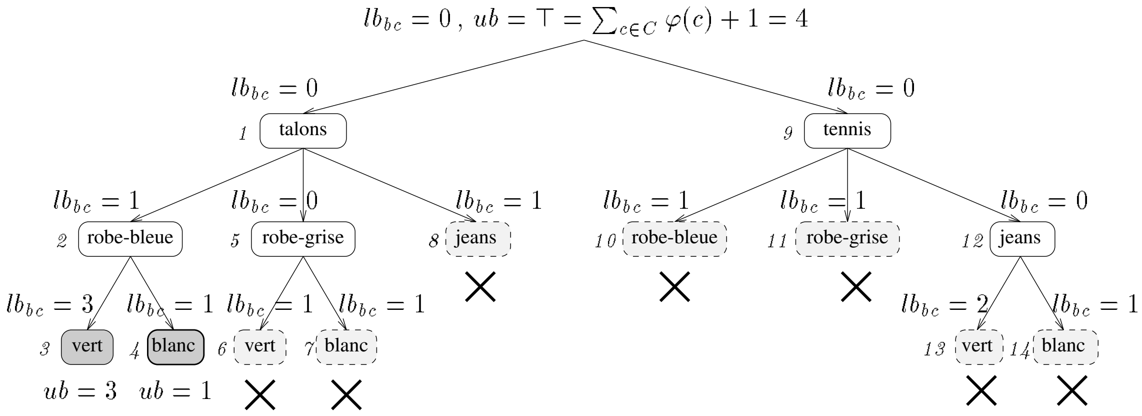


Fig. 2.8 - Arbre de recherche développé par l'algorithme *Depth First Branch and Bound*

2.3.1.3 Heuristiques sur l'ordre des variables et des valeurs ; expérimentations

Les heuristiques[11] sur l'ordre d'affectation des variables et sur l'ordre de sélection des valeurs ont pour objectifs de réduire l'arbre de recherche effectivement développé, tout en restant complet. La complétude est assurée tant qu'aucune variable ou valeur n'est omise. L'accélération est obtenue en guidant l'exploration de l'arbre vers une "bonne" solution, tout en essayant de couper au plus tôt les parties inintéressantes.

Pour l'ordre sur les variables, un premier principe consiste à choisir celle qui est la plus pénalisante (en anglais, *first fail principle*). Cela se traduit, pour les CSP, par le choix en premier d'une variable fortement contrainte : de degré[12] élevé, avec des contraintes de forte dureté. Un second principe consiste à choisir une variable ayant le plus petit domaine : cela permet de "resserrer" l'arbre près de la racine dans l'espoir de développer un arbre plus petit.

Largeurs versus nombre de *backtrack* ; heuristiques sur les variables

Un troisième principe consiste à limiter les retours-arrières. Pour cela, nous avons besoin de définir les notions de largeur et de largeur de bande, pour un CSP binaire[13] :

Définition 19
Largeur *[AS94] :*
étant donnés un CSP *binaire et un ordre sur ses variables, la* **largeur d'une variable** x_i *est égale au nombre de variables connectées à* x_i *qui la précèdent dans l'ordre. La* **largeur d'un ordre** *est le maximum de la largeur des variables dans cet ordre. La* **largeur d'un** CSP *est égale à la largeur minimale sur tous les ordres.*

Définition 20
Largeur de bande *[Tsa93] :*

[11] Nous ne parlons pas des heuristiques pour ordonner les vérifications de contraintes qui peuvent éventuellement apporter un gain en nombre de contraintes testées.
[12] Le degré d'une variable est égal au nombre de contraintes qui portent sur cette variable.
[13] Ces définitions de largeur et largeur de bande s'appliquent aussi bien aux CSP, qu'aux VCSP ; seule la structure du graphe des contraintes compte. C'est pourquoi nous parlons aussi bien de largeur d'un CSP que de largeur du graphe associé.

étant donnés un CSP *binaire et un ordre sur ses variables, la* **largeur de bande d'une variable** *x_i est égale à la distance maximale entre x_i et une autre variable x_j connectée à x_i ; la distance étant comptée en nombre de variables séparant x_i de x_j, suivant l'ordre. La* **largeur de bande d'un ordre** *est le maximum de la largeur de bande des variables dans cet ordre. La* **largeur de bande d'un** CSP *est égale à la largeur de bande minimale sur tous les ordres.*

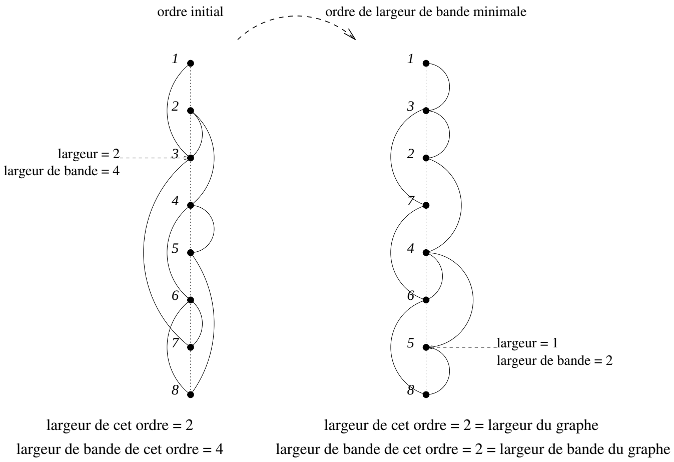


Fig. 2.9 - Exemple de largeur et largeur de bande pour deux ordres d'un même graphe de contraintes

Pour illustrer ces définitions, voir la figure 2.9 qui reprend le graphe de la figure 2.6. Dans l'exemple du robot (cf. fig. 2.1), la largeur du graphe associé est 2, ainsi que la largeur de bande.

Trouver un ordre de largeur minimale est un problème résolu en temps polynomial, $O(n^2)$ [Tsa93]. En revanche, trouver un ordre de largeur de bande minimale est un problème NP-dur. [GS84] proposent un algorithme de complexité spatiale et temporelle en $O(n^k)$ pour vérifier si un graphe de n noeuds a une largeur de bande de k. [Zab90] démontre que la largeur de bande d'un graphe est toujours supérieure ou égale à la largeur de ce graphe (vrai quel que soit l'ordre des variables). Ainsi la largeur d'un graphe est un minorant de sa largeur de bande.

Un graphe qui a une largeur de 1 est un graphe acyclique (un arbre ou une collection d'arbres). Nous retombons donc sur une classe polynomiale des CSP (cf. section 2.2.1.3), qui ne nécessite pour sa résolution aucun retour-arrière. Si la largeur est plus grande, la quantité de retour-arrière risque d'augmenter [Tsa93].

Théorème 8
[Fre82]
Un CSP *binaire de largeur L peut être résolu sans retour-arrière s'il est fortement k-cohérent avec $k > L$.*

Minimiser la largeur de bande correspond concrètement à rapprocher les variables qui sont fortement en relation entre elles. Cela a directement un effet sur le nombre de retour-arrière pour remonter aux causes d'un échec (dans le cas d'un CSP, ce nombre est borné par la largeur de bande de la variable qui correspond à un noeud d'arrêt).

Heuristiques statiques ou dynamiques ; heuristiques sur les valeurs

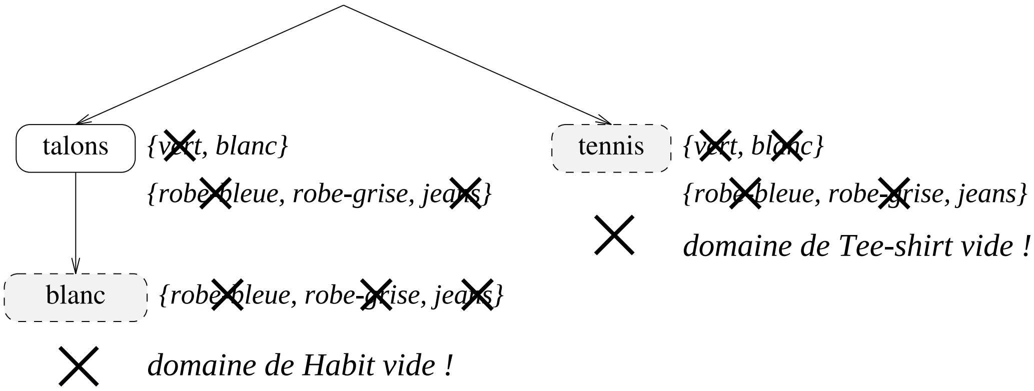


Fig. 2.10 - En CSP, arbre de recherche développé par l'algorithme *Forward Checking*, avec heuristiques

Lorsqu'un ordre est établi une fois pour toute, avant la phase de résolution, il s'agit d'un **ordre statique** ; En revanche, s'il y a utilisation des informations récoltées durant la recherche pour déterminer le prochain élément dans l'ordre, il s'agit d'un **ordre dynamique** . Une heuristique dynamique efficace consiste à sélectionner la prochaine variable qui a le plus petit domaine. La taille des domaines évolue en fonction de la propagation triviale des contraintes lors d'un renforcement de cohérence locale, après chaque affectation d'une variable. Dans le cas du *Forward Checking* pour CSP, il est possible de supprimer les valeurs des variables futures qui sont interdites par des contraintes quasi-instanciées. La figure 2.10 montre le parcours de l'arbre de recherche par l'**algorithme** *Forward Checking* avec une heuristique dynamique choisissant en premier la variable de plus petit domaine (*Chaussures*, puis *Tee-shirt* dans la branche de gauche ; *Chaussures*, puis *Habit* dans la branche de droite). À chaque noeud, les domaines des variables futures sont indiqués, ainsi que les valeurs interdites par vérification des contraintes quasi-instanciées. Par exemple, $C_{quasi}(\{Chaussures \mapsto talons\}, P) = \{\{(talons, blanc)\}, \{(talons, robe-grise), (tennis, jeans)\}\}$. Ici, les 2 noeuds d'arrêt sont dus à un domaine qui est vidé par application du *Forward Checking*.

Pour l'ordre sur les valeurs, le principe revient à choisir en premier celles qui offrent le plus de chance de conduire vers une "bonne" solution (en anglais, *best-first principle*). Les heuristiques sur les valeurs sont très dépendantes du domaine d'application du problème à résoudre. Par exemple, en ordonnancement, pour des contraintes disjonctives et des domaines de valeurs numériques, une heuristique consiste à sélectionner les valeurs alternativement de gauche à droite pour une variable, puis de droite à gauche pour la variable suivante, et ainsi de suite. En optimisation, pour chaque valeur, la valuation induite par renforcement de cohérence locale (cf. section 2.2.2), *Arc Consistency Count*, *Directed Arc Consistency Count* ou *Forward Checking Count*, noté *fcval*, offre une heuristique de choix des valeurs : choix en premier de la valeur ayant la plus petite valuation induite. Dans le cas d'un **algorithme** *Forward Checking* **valué** pour VCSP, la figure 2.11 montre l'arbre de recherche obtenu avec heuristique dynamique sur les variables, choix de la variable ayant le plus petit domaine en premier, et heuristique dynamique sur les valeurs, choix de la valeur ayant le plus petit *fcval* en premier. À côté d'un noeud, figurent les domaines des variables futures, avec pour chaque valeur d'un domaine, une valuation associée, son *fcval*. Le calcul d'un *fcval* d'une valeur val de $D(v)$ vaut :

$$fcval_{v,val} = \bigotimes_{c \in C_{quasiunsat}(A \cup \{v \mapsto val\}, P)} \varphi(c)$$

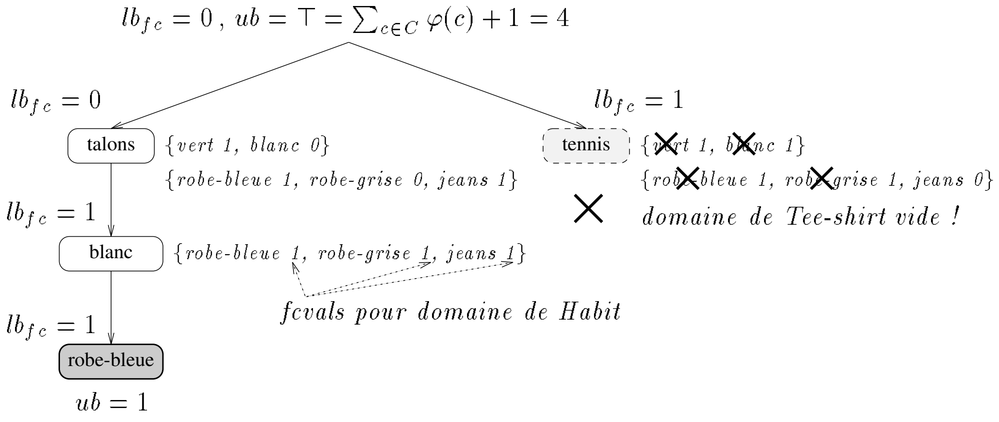


Fig. 2.11 - En VCSP, arbre de recherche développé par l'algorithme *Forward Checking* valué, avec heuristiques

Heuristiques dynamiques : exemples de statistiques sur les domaines

[FW92] proposent d'ordonner les variables suivant des statistiques récoltées sur les domaines. Ils utilisent les valuations induites *Arc Consistency Counts*, produites pour chaque valeur, pour calculer une moyenne de ces valuations par variable. Les variables futures sont ordonnées par valuation moyenne d'ACC décroissante. Cette moyenne de minorants ACC associées aux valeurs d'une variable v' non affectée, est une approximation de l'augmentation de la valuation d'une instanciation partielle étendue à v'.

Plutôt que des minorants par valeur, [KY89] proposent de calculer une valuation moyenne par valeur, puis une moyenne de ces valuations par variable. Cette moyenne des valuations, associée à une variable v' non affectée, est une approximation plus réaliste de l'augmentation de la valuation d'une instanciation partielle étendue à v'. Leurs calculs reposent sur les deux principes énoncés précédemment, à savoir le choix de la variable la plus contrainte en premier, suivant un critère appelé *criticality*, noté Ω, et le choix de la valeur la moins en conflit en premier, suivant un critère appelé *cruciality*, noté Γ.

Nous avons adapté ces critères, initialement appliqués à des problèmes d'allocation sous contraintes de ressources à des tâches pour la planification ou l'ordonnancement, aux Σ-VCSP[14].

Le critère $\Gamma_{v,val}$ dénote une moyenne pondérée de l'ensemble des valuations des extensions d'une affectation unique $\{v \mapsto val\}$ à deux variables $\{v \mapsto val, v' \mapsto val'\}$; la pondération correspond

[14] La restriction aux VCSP additifs permet de mélanger aisément des valuations et des probabilités.

à la probabilité de choisir la valeur val' dans le domaine de v'. Cette probabilité est simplement calculée suivant la taille du domaine, $\frac{1}{|D(v')|}$. Le calcul des Γ ne considère que les variables $V'(A)$ non affectées par l'instanciation partielle courante A :

$$\Gamma_{v,val} = \sum_{v' \in V'(A \cup \{v \mapsto val\})} \sum_{val' \in D(v')} \frac{\varphi(\{v \mapsto val, v' \mapsto val'\}, P)}{|D(v')|}$$

$$\Gamma_{v,val} = \sum_{v' \in V'(A \cup \{v \mapsto val\})} \sum_{val' \in D(v')} \frac{1}{|D(v')|} \sum_{c \in C_{unsat}(\{v \mapsto val, v' \mapsto val'\}, P)} \varphi(c)$$

Le critère Ω_v établit une combinaison empirique des Γ associés aux valeurs de son domaine. Nous combinons également les valuations induites par le *Forward Checking*, $fcval_{v,val}$; ces valuations portent sur des contraintes différentes de celles utilisées pour le calcul des Γ. Enfin la taille des domaines est utilisée pour départager des Ω_v similaires :

$$\Omega_v = \sum_{val \in D(v)} \frac{1}{\Gamma_{v,val} + fcval_{v,val} + \frac{1}{|D(v)|}}$$

L'heuristique choisit en premier la variable qui a le plus petit Ω. Puis la valeur qui a la plus petite somme $\Gamma_{v,val} + fcval_{v,val}$ est sélectionnée en premier.

L'heuristique de choix des variables que nous venons de présenter est un raffinement d'une heuristique statique assez connue qui consiste à classer par ordre décroissant les variables suivant un critère $\sum_{c_i \in C / c_i = (v_i, r_i) \wedge v \in v_i} \varphi(c_i) \times \text{dureté}(c_i)$ calculé pour chaque variable v. C'est pourquoi, nous appelons, dans les expérimentations suivantes, l'heuristique utilisant Ω et Γ : *(coût * dureté) dynamique*.

Expérimentation d'heuristiques dynamiques sur des Σ-VCSP aléatoires

Nous avons comparé plusieurs heuristiques dynamiques sur une série de problèmes additifs binaires générés aléatoirement (cf. annexe A.1). Les paramètres classiques de génération sont le nombre de variables (n), la taille des domaines (d), la connectivité du graphe des contraintes (c) et la dureté des contraintes (t). Une distribution uniforme des coûts est appliquée pour valuer les contraintes : $\varphi \in \{1, 10, 100, 1000\}$. L'algorithme de résolution est un *Forward Checking* valué. L'heuristique $h1$ correspond à celle utilisée pour l'exemple en figure 2.11, c'est à dire sélection de la variable de plus petit domaine puis à égalité de plus grand degré, et sélection de la valeur de plus petit $fcval$. L'heuristique $h2$ correspond à celle décrite précédemment, utilisant Γ et Ω. Enfin, l'heuristique $h3$ est identique à l'heuristique $h1$, sauf qu'en cas d'égalité des tailles de domaine et des degrés, les variables sont départagées suivant leur plus petit $fcval$: celle qui a le plus grand est sélectionnée en premier.

La figure 2.12 montre le temps d'exécution moyen pour 100 problèmes aléatoires générés avec $n = 20, d = 5, c = 50\%$. À la dureté 80%, $h2$ est 11,5 fois plus rapide que $h1$; elle l'est 17,5 fois plus à 90%. Nous voyons donc que malgré le surcroît important de calcul pour évaluer notre heuristique $h2$ à chaque noeud, il y a un net gain par rapport à $h1$. En revanche, $h3$ montre qu'il n'est pas si évident de trouver une heuristique efficace ; elle est même moins performante que $h1$ pour de fortes duretés.

La figure 2.13 montre le temps moyen avec écart-type pour $h1$ et $h2$ sur 100 Σ-VCSP binaires de 30 variables. L'écart-type est beaucoup plus fort pour $h1$ que pour $h2$; pour une dureté

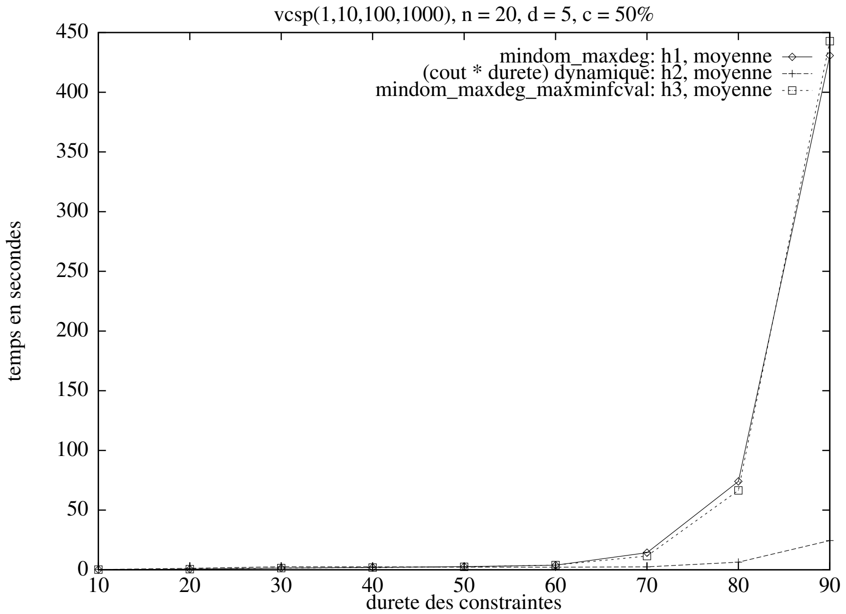


Fig. 2.12 - Comparaison du temps d'exécution mis par un *Forward Checking* valué, avec des heuristiques différentes ; moyennes sur 100 Σ-VCSP (20 variables)

de 60%, il vaut 1218 pour $h1$ contre 425 pour $h2$. Pour mieux se rendre compte de l'énorme variance observée entre deux instances de Σ-VCSP ; la figure détaille aussi les médianes 1/4 et 1/2. Une limite de temps d'exécution est fixée à 1 heure. Nous voyons que cette limite est déjà atteinte par la médiane 1/2 pour $h1$ à la dureté 70%. Ainsi l'intérêt d'une bonne heuristique est non seulement de réduire le temps d'exécution moyen mais aussi la variance sur le temps d'exécution. Ces figures mettent en valeur les problèmes que rencontre un système temps réel intelligent, cf. section 1.1.2.

2.3.1.4 Améliorations du mécanisme de recherche : méthodes rétrospectives

Les méthodes de renforcement de cohérence locale examinent les variables futures pour détecter plus tôt une incohérence, elles peuvent être vues comme des **méthodes prospectives** . Une autre catégorie de méthodes, dite **méthodes rétrospectives** , profite de l'exploration arborescente effectuée durant l'exécution de l'algorithme, et en particulier des noeuds d'échec, pour déterminer qu'une partie de l'instanciation partielle courante interdit de l'étendre en une instanciation complète cohérente (ou optimale, en VCSP).

Backtrack intelligent

Ce mécanisme concerne les méthodes de parcours en profondeur d'abord. Jusqu'à présent, le retour-arrière est fait de façon chronologique : en cas d'échec, le parcours remonte au noeud précédent, c'est à dire que la variable courante redevient non affectée et la variable précédente dans l'ordre est affectée à une autre valeur. Un retour-arrière intelligent consiste à ne pas revenir sur des variables qui ne sont pas cause de l'échec. Dans ce cas, la prochaine variable affectée n'est plus forcément la précédente dans l'ordre, mais éventuellement une bien antérieure.

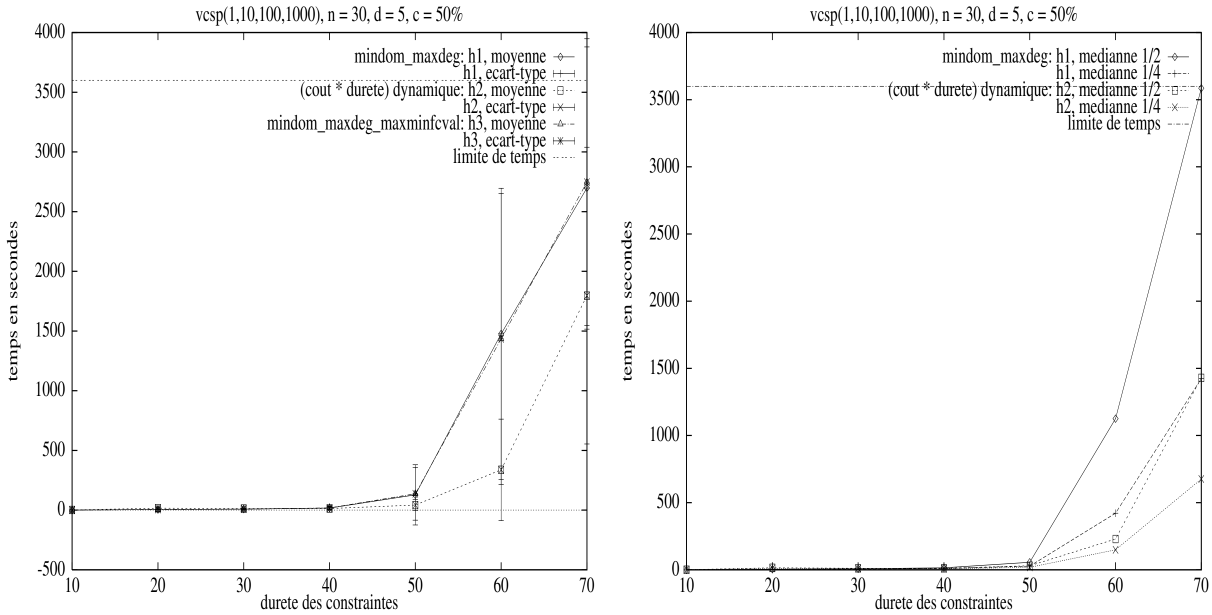


Fig. 2.13 - Comparaison du temps d'exécution mis par un *Forward Checking* valué, avec des heuristiques différentes ; moyennes, écart-types et médianes sur 100 Σ-VCSP (30 variables)

Les variables qui causent l'échec forment une instanciation A', portant sur un sous-ensemble des variables affectées, telle que : $\forall val \in D(v), \varphi(A' \cup \{v \mapsto val\}, P) \geq ub$, avec v la variable courante d'échec. Soit $(x_1^{A'}, \ldots, x_l^{A'})$ la liste des variables affectées par A' et numérotées suivant l'ordre d'instanciation. Le retour-arrière s'effectue sur la dernière variable dans cette liste, c'est à dire $x_l^{A'}$.

Suivant le calcul du sous-ensemble A', plusieurs algorithmes de retour-arrière intelligent ont été définis dans le cadre CSP : l'**algorithme** *Backjumping* [Gas79], le *Conflict-Directed Backjumping* [Pro93], le *Graph-Based Backjumping* [Dec90], le *Dynamic Backtracking* [Gin93, VS94]. Des comparaisons de ces algorithmes peuvent être trouvées dans [Pro93, FD94] ; une librairie, écrite en C pour traiter les CSP, implémente ces algorithmes (excepté le *Dynamic Backtracking*), ainsi que plusieurs renforcements de cohérence locale (*Forward Checking*, arc-cohérence, chemin-cohérence) [van94]. Une extension au cadre VCSP des algorithmes de type retour-arrière intelligent est étudiée dans [FW92, Sch94, Dag97].

Apprentissage par mémorisation de contraintes induites

Le principe consiste à utiliser la recherche de solution pour produire et mémoriser des contraintes induites. Ces contraintes servent, comme dans le cas d'un renforcement de cohérence, à détecter au plus tôt une incohérence dans l'instanciation partielle. Dans le cadre des CSP, ces contraintes sont des instanciations partielles globalement incohérentes et appelées *Nogoods*. Par exemple, dans le cas du retour-arrière intelligent qui permet aussi une mémorisation simple de contraintes induites, la sous-instanciation A' des variables affectées conduisant à un échec forme un *Nogood*. La mémorisation de contraintes induites pose le problème d'un compromis entre l'occupation mémoire de ces nouvelles contraintes et le gain en réduction du parcours de l'arbre de recherche. Une solution consiste à mémoriser uniquement des contraintes de faible arité (classiquement, contraintes unaires et binaires). Ces techniques d'apprentissage sont étudiées dans

[Dec90, SV94].

Une version coopérative de l'apprentissage, profitant de l'application de plusieurs heuristiques différentes pour ordonner les variables et les valeurs, est décrite dans [MV96]. D'autre part, l'intérêt de l'apprentissage pour des problèmes dynamiques est brièvement abordée dans la section 2.1.3.2.

Une extension des algorithmes d'apprentissage pour le cadre VCSP est développée dans [DV96, Dag97]. L'extension repose sur la notion de *Nogood* valué qui est un triplet (A', C', e), avec A', une instanciation partielle, C', un ensemble de contraintes (appelé "justification") et e, une valuation telle que e soit un minorant des valuations de toutes les extensions complètes de A' sur le problème réduit aux contraintes de C'.

Apprentissage par résolutions successives

L'**algorithme** *Russian Doll Search* [LV96] propose une autre forme d'apprentissage, valable pour les VCSP. Cet algorithme utilise un ordre statique sur les variables. Il calcule alors l'optimum des sous-problèmes réduits à la dernière variable dans l'ordre, puis aux deux dernières, puis aux trois, etc, jusqu'à résoudre le problème complet. Les optima des sous-problèmes inférieurs au rang i de l'ordre sont réutilisés pour la résolution du sous-problème de rang i par l'intermédiaire du calcul d'un minorant d'une instanciation partielle ; voir le calcul de $lb_{rds}(A, P)$ dans la section 2.3.1.2. Cet algorithme est un exemple de combinaison d'un schéma rétrospectif avec un schéma prospectif, car il utilise aussi les valuations produites par le *Forward Checking*.

2.3.2 Méthodes incomplètes

Nous parlons de méthodes incomplètes au sens où elles perdent toute garantie d'optimalité. Ces méthodes sont incapables de prouver qu'il n'existe pas de solution à un problème ; elles ne peuvent pas non plus garantir que la solution qu'elles produisent est une solution de valuation minimum (sauf en cas de solution dont la valuation rejoint un minorant du problème, connu auparavant) ou bien que la valuation de cette solution se trouve à une certaine distance bornée de l'optimum. Néanmoins, les méthodes incomplètes peuvent s'abstraire du côté rigide de la construction systématique d'une affectation partielle pour explorer de façon plus opportuniste l'espace de recherche.

2.3.2.1 Recherche locale

Ces méthodes effectuent une succession de modifications locales, c'est à dire des modifications de l'affectation d'un sous-ensemble des variables, pour une ou plusieurs instanciations complètes. L'instanciation de valuation minimum est mémorisée.

Une description détaillée des principales méthodes de recherche locale, ainsi que leur application sur des exemples de problèmes réels, peut être trouvée dans [AL97]. Une vue unificatrice de ces méthodes est proposée dans [VAL95]. Nous choisissons de ne présenter que les principes généraux, ainsi que quelques techniques donnant de bons résultats expérimentaux. Nous discernons les méthodes suivant qu'elles opèrent sur une ou plusieurs instanciations complètes.

Recherche locale à partir d'une seule instanciation complète

Le schéma générique d'une recherche locale est :

- répéter
  1. solution := instanciation-complète := Initialiser()
  2. répéter
     (a) voisinage := Créer(instanciation-complète)
     (b) si Accepte(Sélectionner(voisinage)) alors
        i. instanciation-complète := Sélectionner(voisinage)
        ii. si φ(instanciation-complète, P) $\prec$ φ(solution, P) alors solution := instanciation-complète
     (c) fin si
  3. tant que Continue() et $\varphi(\text{solution}, P) \succ \perp$
- tant que TempsCPU() < temps-limite et $\varphi(\text{solution}, P) \succ \perp$

Remarque 7 Un algorithme de recherche locale ne termine pas, sauf si une solution de valuation égale à $\perp$ est trouvée ; il est interrompu lorsqu'un temps-limite est atteint. ◇

La signification des variables et des fonctions est :

- instanciation-complète : contient une affectation courante de toutes les variables ; la meilleure instanciation trouvée est recopiée dans solution ;
- Initialiser() : génère une nouvelle affectation de toutes les variables ; réalise les choix des affectations soit de façon aléatoire, soit en fonction d'une heuristique ; la fonction peut éventuellement s'inspirer de la meilleure solution trouvée à l'étape précédente (stratégie *Averaging in* [SK93]) ;
- voisinage : correspond à un ensemble d'instanciations complètes proches de instanciation-complète ; la différence porte habituellement sur une seule variable (permutation de son ancienne valeur par une nouvelle), mais peut porter sur m variables suivant la nature du problème (par ex., modification des variables reliées par une contrainte dure d'arité m) ;
- Créer() : crée le voisinage à partir de instanciation-complète ; cette création peut être exhaustive ($n * (d-1)$ éléments pour un problème à n variables ayant des domaines de taille d), incomplète (par exemple, choix aléatoire d'une variable puis insertion des $d-1$ permutations de sa valeur pour ensuite accepter la permutation diminuant le plus la valuation, suivant l'heuristique *Min-Conflicts* [MJPL90]), ou totalement aléatoire (un seul élément choisi aléatoirement, stratégie du *Random Flip*) ;
- Sélectionner() : sélectionne une instanciation dans voisinage ; généralement choisit celle ayant la meilleure valuation ; mais peut aussi de temps en temps sélectionner aléatoirement une instanciation de moins bonne valuation qui modifie une variable reliée à une contrainte insatisfaite, suivant la stratégie du *Random Walk* [SK93] ;
- Accepte() : considère la différence de valuation entre l'instanciation courante et l'instanciation sélectionnée ; choisit d'accepter la nouvelle instanciation si la différence est positive ou nulle (un ensemble d'instanciation de même valuation forme un "plateau") ; peut accepter avec une différence négative suivant une certaine probabilité fixe ou évolutive ;

- Continue() : teste s'il est intéressant de continuer la recherche locale sur l'instanciation courante ; un critère d'arrêt peut être l'absence de modification de l'instanciation courante pendant un certain nombre d'itérations, ou bien le nombre de modifications successives à différence de valuation nulle (taille maximum de parcours d'un plateau), ou encore obtention d'un voisinage vide ;

- TempsCPU() : retourne le temps CPU consommé par l'algorithme.

Un voisinage créé de façon complète et une sélection du meilleur élément avec acceptation uniquement si la différence est positive ou nulle assure une sélection optimale dans le voisinage et correspond à un **algorithme de descente en gradient** .

Le comportement glouton de l'algorithme précédent le conduit dans un optimum local où il reste bloqué jusqu'à ce que Continue() retourne faux. Pour sortir d'un optimum local, il est nécessaire d'accepter une dégradation de la valuation de l'instanciation courante et donc de créer un voisinage pouvant contenir de telles instanciations (par exemple, par permutation aléatoire). La difficulté consiste alors à ne pas retomber dans le même optimum local lors des itérations suivantes. L'**algorithme** *Tabu Search* [GL93c, AL97] retient un liste des t derniers mouvements[15] (en anglais, **flip**) acceptées. Il interdit de choisir une nouvelle instanciation appartenant déjà à la liste *Tabu* (une fonction dite "d'aspiration" permet d'outrepasser cette règle si la valuation de l'instanciation interdite est nettement meilleur que la valuation courante). Le paramètre t permet d'éviter des cycles (série de modifications ramenant au même optimum local) de longueur $\leq t$, son réglage s'avère particulièrement important.

Une étude comparative de plusieurs raffinements autour de l'algorithme de descente en gradient est faite dans [GW93a] sur des problèmes SAT aléatoires et dans [Wal96a] sur des problèmes Σ-VCSP aléatoires.

Remarque 8 Les méthodes de recherche locale ne font que minimiser un certain critère, la valuation d'une instanciation complète. Elles sont incapables de garantir le respect des contraintes dures d'un VCSP. Il y a deux solutions pour traiter ces contraintes dures : soit les modifications locales s'effectuent dans un espace d'instanciations admissibles (utilisation de méthodes complètes pour respecter les contraintes dures), soit les contraintes dures sont traitées comme des contraintes molles, avec cependant une valuation rendue inférieure à $\top$ tout en restant supérieure à la valuation des contraintes molles (par exemple, en Σ-VCSP, $\forall c \in C_{dures}, \varphi(c) = \sum_{c' \in C_{molles}} \varphi(c')$). ◇

Exemple : l'algorithme de recuit simulé

L'**algorithme de recuit simulé** [AK89, AL97] choisit aléatoirement une instanciation initiale puis effectue une série de permutations aléatoires, en acceptant la permutation A' suivant sa différence de valuation avec l'instanciation courante A :
Accepte(A') :

- si $\varphi(A') \preceq \varphi(A)$ alors retourne vrai

- si $\varphi(A') \succ \varphi(A)$ alors retourne vrai avec une probabilité $P(A, A')$[16] sinon retourne faux

$$P(A, A') = e^{-\frac{\varphi(A') - \varphi(A)}{T}}$$

[15] Un mouvement est un changement d'affectation, pour une variable, d'une valeur à une autre.

[16] Le calcul de cette probabilité est issu d'une analogie avec la physique statistique. Soit S, l'ensemble de tous les états possibles d'un système, soit T, sa température, et soit $E(s)$, l'énergie d'un état $s \in S$. La probabilité que

Le calcul de l'exponentielle impose que la différence de deux valuations soit mesurable, c'est le cas pour la classe des Σ-VCSP. En revanche, de manière générale pour traiter un problème CSP par une méthode de recherche locale, il est nécessaire de transformer le problème en un problème additif où toutes les contraintes prennent un coût de 1 ; le problème est satisfiable si l'optimum vaut zéro, sinon il est indéterminé (satisfiable ou insatisfiable).

Le paramètre T, appelé température, contrôle le taux d'acceptation. Pour $T \rightarrow \infty$, toute instanciation est acceptée (comportement aléatoire), alors que pour $T \rightarrow 0$, seules des instanciations de différence positive ou nulle sont acceptées (comportement glouton). L'algorithme simule un phénomène physique de recuit, permettant à un système physique soumis à une température T d'abaisser son niveau d'énergie lorsque la température décroît lentement, jusqu'à atteindre un niveau stable (état solide ou cristallin) d'énergie minimale lorsque T s'approche de zéro. Ce schéma de descente en température progressif peut être obtenu en répétant une succession d'itérations à température constante (un palier) suivie d'un abaissement de la température (par exemple, suivant une suite géométrique qui multiplie la température par un nombre inférieur à 1)[17].

Une bonne température de départ est calculée de manière à avoir un taux d'acceptation (égal au nombre de permutations proposées sur le nombre de permutations acceptées) proche de 1 sur le premier palier. Par ailleurs, un critère d'arrêt peut être l'absence de permutation acceptée durant tout un palier.

Nous avons mis au point un algorithme hybride qui combine un algorithme de recuit simulé avec un algorithme glouton de descente en gradient. Cette hybridation permet d'améliorer le profil de qualité ; c. à d. trouver de meilleurs solutions plus tôt.

La descente en gradient de longueur g est effectuée lorsqu'une permutation A' est refusée par la probabilité $P(A, A')$. À la fin de la descente, une permutation A'' a été atteinte, l'algorithme teste à nouveau la probabilité $P(A, A'')$ pour décider si A'' est acceptée. Un principe similaire est employé dans la méthode *Variable-Depth Search* [AL97].

La figure 2.14 présente des profils de qualité obtenus sur 100 problèmes Σ-VCSP aléatoires ($n = 15, d = 10, c = 80\%, t = 80\%$) avec 4 paramétrages différents de l'algorithme de recuit simulé qui est éventuellement combiné avec une descente en gradient. Soit $ub(t)$, la meilleure valuation trouvée pour un problème au temps t et soit e, la somme des coûts des contraintes du problème ; la qualité moyenne pour un profil à un instant t est définie par la moyenne des rapports $(e - ub(t))/(e - \varphi(P))$ sur les 100 problèmes. De même, la qualité médiane 1/2 est obtenue en triant les 100 rapports et en prenant le 50ième rapport dans l'ordre, et ceci pour chaque instant t. L'optimum $\varphi(P)$ est préalablement calculé par une méthode exacte pour les 100 problèmes[18].

le système soit dans l'état s est donnée par la distribution de *Gibbs-Boltzmann* $P(s) = e^{-E(s)/T} / \sum_{s' \in S} e^{-E(s')/T}$. Lorsque la température décroît, les états de haute énergie ont leur probabilité qui décroît, à l'inverse, les états de basse énergie ont leur probabilité qui augmente. L'analogie est faite entre respectivement un système physique, un état, son énergie et, un problème d'optimisation combinatoire, une instanciation complète, sa valuation. L'algorithme n'accepte à une température donnée que les instanciations les plus probables, suivant la distribution de *Gibbs-Boltzmann*.

[17] Des résultats théoriques apportent la preuve de convergence d'un algorithme de recuit simulé "idéal" vers une instanciation globalement optimale [AL97] ; néanmoins en temps limité, l'implémentation d'un algorithme de recuit simulé constitue une méthode incomplète.

[18] L'algorithme du *Forward Checking* valué (idem à celui employé en fig. 2.11) prends en moyenne 46 secondes.

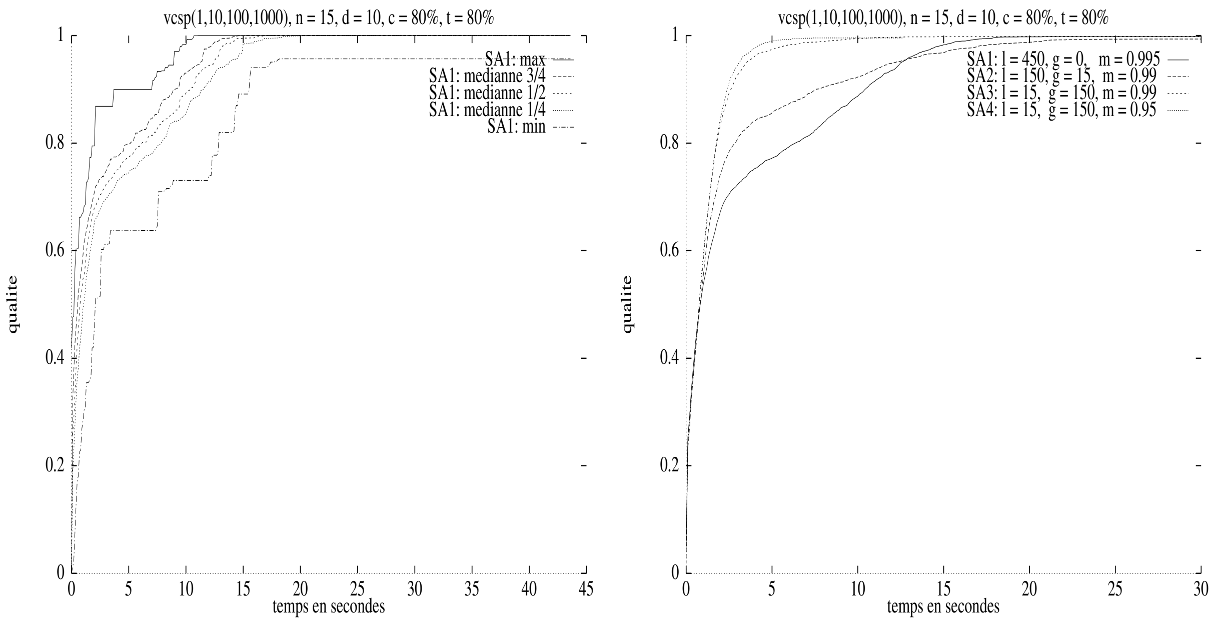


Fig. 2.14 - Profil de qualité de l'algorithme de recuit simulé suivant différents paramètres ; médianes et moyennes sur 100 Σ-VCSP (15 variables)

La figure de gauche montre comment se répartissent les médianes pour SA1, qui a une taille de palier égale à 3 fois le nombre de valeurs ($l = 450$), ne fait aucune descente en gradient supplémentaire ($g = 0$) et a un coefficient multiplicateur pour décrémenter la température de 0.995 ($m = 0.995$). Nous observons que la variance est relativement faible, et la majorité des problèmes sont résolus au bout de 15 secondes.

La figure de droite compare les différentes versions, en faisant varier le rapport entre le nombre de permutations d'un palier à température constante (l) et le nombre de permutations d'une descente en gradient (g). SA2 effectue des paliers assez longs avec peu de gradient ; elle s'avère moins performante que SA3, qui applique des proportions inverses, sur ces problèmes. SA4 effectue une descente en température plus rapide. Le tableau suivant résume le temps moyen pris pour chaque version[19], ainsi que la qualité moyenne obtenue après 30 secondes d'exécution :

version	**palier (l)**	**gradient (g)**	**coeff. mult. (m)**	**temps moyen**	**qualité à $t = 30\,s$**
SA1	450	0	0.995	27.8 sec	99.79 %
SA2	150	15	0.99	39.5 sec	99.36 %
SA3	15	150	0.99	20.7 sec	**99.84 %**
SA4	15	150	0.95	**7.9 sec**	99.56 %

Remarque 9 L'algorithme hybride redevient un algorithme de recuit simulé lorsque $g = 0$. Dans le cas inverse où g est grand, l'algorithme hybride devient un algorithme de descente en gradient avec *Random Flip* pour le voisinage et avec insertion de bruit *Random Walk* (à température élevée). ◇

[19] Le critère d'arrêt retenu est une absence de permutation sur tout un palier.

Recherche locale à partir d'un ensemble d'instanciations complètes

Plutôt que de travailler sur une seule instanciation complète, les **algorithmes génétiques** [Gol89, AL97] travaillent sur un ensemble d'instanciations complètes pour le faire évoluer en s'appuyant sur des procédés analogues à ceux observés en génétique. L'analogie est faite respectivement entre une instanciation, un ensemble d'instanciations et, un individu, une population d'individus.

Le codage d'une instanciation peut être simplement la séquence des valeurs affectées aux variables, dans un ordre statique. L'évolution est réalisée par sélection des individus, a priori ceux ayant une bonne valuation, par mutation de certains individus sélectionnés (similaire à un *Random Flip*) et par croisements entre deux individus (échange d'instanciations partielles entre deux instanciations complètes).

Un intérêt majeur des algorithmes génétiques réside dans l'ensemble de bonnes solutions proposé à l'utilisateur plutôt qu'une solution isolée. Néanmoins, il est difficile de trouver un sens clair à l'opérateur de croisement, dans le cadre des VCSP.

Une autre classe d'algorithmes considère un ensemble d'instanciations complètes de manière probabiliste. À chaque valeur val d'une variable v est associée une probabilité $p(v, val)$ telle que $\sum_{val \in D(v)} p(v, val) = 1, \forall v \in V$. L'**algorithme de recuit en champ moyen** [PS89, AL97] initialise les probabilités $p(v, val)$ à $1/|D(v)|$. Il effectue ensuite une descente en température en faisant évoluer toutes les probabilités à chaque palier, suivant la distribution de *Gibbs-Boltzmann*, à l'instar de l'algorithme de recuit simulé. Le calcul exact des probabilités étant hors de portée, l'approximation en champ moyen considère que chaque variable est contrainte par toutes les autres dans leur état moyen. L'énergie $E(v, val)$ d'une valeur val d'une variable v vaut :

$$E(v, val) = \sum_{v' \in V} \sum_{val' \in D(v)} \varphi(\{v \mapsto val, v' \mapsto val'\}) * p(v', val')$$

Lors d'une mise à jour, la probabilité $p(v, val)$ prend la valeur :

$$p(v, val) = \frac{e^{-E(v,val)/T}}{\frac{1}{|D(v)|} * \sum_{val' \in D(v)} e^{-E(v,val')/T}}$$

Un test de convergence à chaque fin de palier détermine l'arrêt de l'algorithme si le système est dans un état stable.

La figure 2.15 donne deux exemples d'évolution des probabilités pour une variable ayant 10 valeurs. Un état stable est atteint lorsque les probabilités sont toutes à zéro ou à $1/d'$, avec d' le nombre de probabilités non nulles. Dans le cadre des Σ-VCSP, [Cab96] montre que la méthode du recuit en champ moyen fournit un majorant α de la valuation optimale, ainsi qu'un ensemble d'affectations des variables (le produit cartésien des valeurs des d') sur lequel la moyenne de la fonction de coût est égale (strictement) à α. Cet algorithme est déterministe car il converge toujours vers le même ensemble de solutions pour un schéma de descente en température fixé.

Une comparaison d'un vaste panorama d'algorithmes de recherche locale (utilisant une ou plusieurs instanciations) est effectuée sur des problèmes d'ordonnancement d'atelier dans [VAL96].

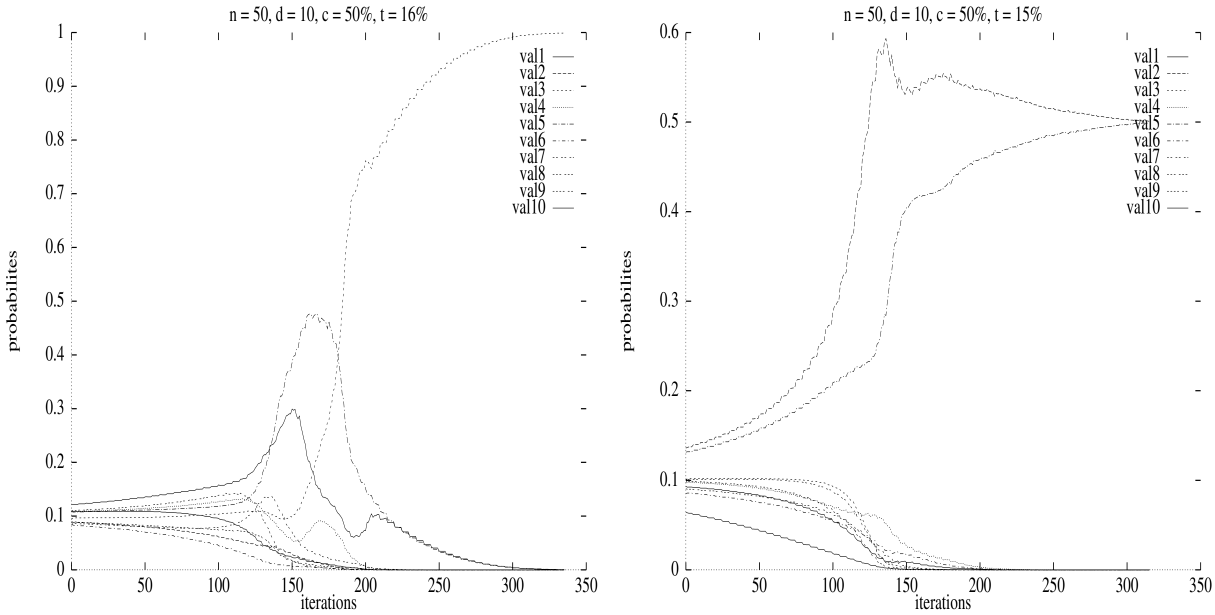


Fig. 2.15 - Évolution des probabilités affectées aux valeurs d'une variable, par l'algorithme de recuit en champ moyen appliqué à un Σ-VCSP (50 variables, $\forall c \in C, \varphi(c) = 1$)

2.3.2.2 Recherche arborescente rendue incomplète

Le principe consiste à partir de l'algorithme *Depth First Branch and Bound*, d'y appliquer éventuellement toute technique prospective ou rétrospective vue précédemment, de manière à récupérer les bonnes propriétés de faible occupation mémoire, de génération rapide d'une première solution (du fait de la recherche en profondeur d'abord), d'élimination des zones détectées incohérentes de l'arbre de recherche et de l'apport d'heuristiques guidant *a priori* vers une bonne solution. En revanche, un inconvénient majeur d'une recherche en profondeur d'abord, appelé *Early Mistake Problem*, réside dans le fait de partir vers une zone de l'arbre de recherche voulue par l'heuristique et d'y rester enfoncé tant que le retour-arrière n'est pas remonté aux premières variables affectées. L'idée novatrice de ces méthodes incomplètes repose sur la multiplicité des zones explorées de l'arbre de recherche, dans un temps limité. Cela devrait permettre de trouver de meilleures solutions qu'un *Depth First Branch and Bound* complet pour un temps d'exécution fixé. Pour cela, deux techniques sont combinées : exécutions multiples avec des heuristiques différentes et diversification dans les sous-arbres explorés par des coupes heuristiques dans le parcours de l'arbre.

Exécutions multiples

Initialement, un **algorithme glouton** tente de construire une instanciation complète en effectuant un choix localement optimal à chaque affectation de variable, c'est à dire en effectuant le meilleur choix vis à vis des informations disponibles [CLR94]. Un exemple d'algorithme glouton consisterait à construire une instanciation en utilisant les heuristiques Γ et Ω présentées dans la section 2.3.1.3 avec application d'un filtrage des valeurs par *Forward Checking*. Il suffit d'exécuter plusieurs algorithmes gloutons avec des heuristiques différentes pour multiplier les zones explorées de l'arbre.

Si aucune heuristique ne semble efficace, une méthode simple consiste à choisir aléatoirement parmi les valeurs, lors de la construction de l'instanciation ; c'est le principe de l'**algorithme** *Iterative Sampling* , expérimenté dans [Lan92, CB94b]. Plutôt que de choisir aléatoirement en haut de l'arbre, une autre manière de diversifier la recherche est d'utiliser un parcours complet en largeur en haut de l'arbre pour initialiser le début des chemins. Une autre façon de multiplier les ordres revient à guider les choix suivant une confiance limitée dans une heuristique. Cela se traduit par des affectations aléatoires biaisées : la probabilité de sélectionner une valeur est une fonction décroissante de la confiance en cette valeur, c-à-d son numéro d'ordre dans son domaine trié par une heuristique. Cette méthode est appelée *Heuristic-Biased Stochastic Sampling* [Bre96].

L'**algorithme** *Limited Discrepancy Search* (*LDS*) [HG95, Har95] part d'une heuristique et suppose qu'elle commet i erreurs, en itérant pour $i \in [0, n]$. À l'étape i, *LDS* développe toutes les instanciations distantes de i de l'instanciation obtenue par l'heuristique (étape $i = 0$).

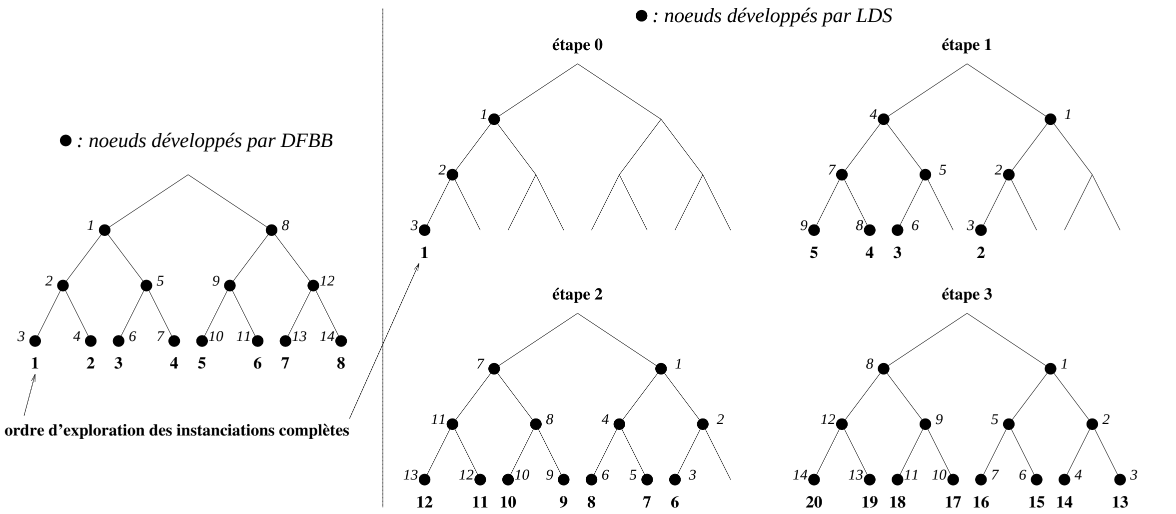


Fig. 2.16 - Trace d'exécution de l'algorithme *Limited Discrepancy Search*, comparé à l'algorithme *Depth First Branch and Bound*

La figure 2.16 montre le parcours d'un arbre binaire (domaines de taille 2) de profondeur 3 par un *Depth First Branch and Bound* (*DFBB*) et par *LDS* ; avec un ordre fixé pour les variables et une heuristique sur les valeurs qui choisit toujours en premier la branche de gauche. Cet algorithme est potentiellement complet s'il est exécuté jusqu'à l'étape égale à la profondeur de l'arbre (dans l'exemple, la profondeur 3). En pratique, il n'est exécuté dans un temps limité que jusqu'à l'étape 1 ou 2. Son efficacité est amélioré en le relançant avec des ordres différents sur les variables.

Une étude faite dans [Har95] montre que la probabilité d'erreur dans le choix des valeurs pour une heuristique donnée, en fonction de la profondeur dans l'arbre de recherche. Pour de nombreuses heuristiques, la probabilité d'erreur est plus grande proche de la racine que proche des feuilles de l'arbre de recherche. Dans ce cas, il serait judicieux de revenir d'abord sur les choix d'instanciation des premières variables, dès la rencontre d'un noeud d'arrêt dans le parcours de l'arbre.

Coupes heuristiques

Plutôt que de revenir à la racine à chaque noeud d'arrêt, comme vu précédemment, il est aussi intéressant d'explorer en profondeur autour du chemin initialement proposé par une heuristique. C'est ce que fait naturellement le *Depth First Branch and Bound*. Pour arrêter simplement ce mécanisme, [Har95] propose soit de borner le nombre de noeuds explorés avant retour à la racine (**algorithme** *Restart Depth First Search*), soit de borner la profondeur à laquelle le retour-arrière peut remonter (**algorithme** *Bounded Backtrack Search* (*BBS*)).

De manière à explorer une plus grande partie de l'arbre, autour du chemin initialement proposé par une heuristique, un mécanisme incrémental, appelé **algorithme** *Iterative Broadening* [GH92], consiste à simplifier le problème en réduisant le nombre de valeurs considérées par variable. Au départ, les domaines des variables sont réduits à une seule valeur, en choisissant la première dans l'ordre donné par une heuristique. À chaque fin d'exploration d'un problème simplifié, l'étape suivante i est exécutée avec une réduction moins forte des valeurs, en prenant en compte les i premières valeurs des domaines. Surtout intéressant pour des domaines grands, une adaptation à des domaines petits conduit à l'**algorithme** *Stochastic Iterative Broadening* [Har95] qui consiste à utiliser un facteur de branchement rationnel b : pour des domaines de taille 2, et un facteur $b = 1.1$, la deuxième valeur sera développée avec une probabilité 0.1.

[Har95] présente des comparaisons entre la plupart des méthodes présentées dans cette section sur un jeu de test de problèmes d'ordonnancement d'atelier [VAL96]. Une combinaison de *LDS* et *BBS* obtient les meilleurs résultats parmi ces méthodes arborescentes incomplètes, en terme de profil de qualité. De plus *LDS+BBS* se compare favorablement, sur ce jeu de test, avec d'autres méthodes incomplètes de type recherche locale, en terme de qualité des solutions obtenues (distance moyenne à un minorant de l'optimum).

2.3.2.3 Schéma d'approximation

Un schéma d'approximation est un algorithme capable de trouver une solution pour une classe de problèmes d'optimisation avec une garantie sur la distance maximale à l'optimum de la solution fournie. Le terme "schéma" précise que la distance maximale est un paramètre de l'algorithme : il est choisi par l'utilisateur. C'est un schéma d'approximation polynomial si la résolution prend dans le pire cas un temps polynomial. Le schéma est dit complètement polynomial lorsque le temps de résolution est borné par un polynôme dépendant de l'instance et aussi du paramètre de distance maximale.

[Lau96] démontre qu'il n'existe pas d'algorithme polynomial capable de résoudre l'ensemble des Σ-VCSP avec une garantie quelconque sur la distance à l'optimum.

Pourtant, il existe des schémas d'approximation en temps polynomial [ACP95, CK95] pour un ensemble de problèmes voisins, appelés *MAX-SAT* (recherche d'une solution maximisant le nombre de clauses satisfaites dans une formule propositionnelle) [Kau95]. Or, il existe une conversion simple entre un problème CSP et un problème SAT (satisfaction de toutes les clauses) [dK89, GR96] ; le passage à *MAX-SAT* permettrait de traiter des problèmes additifs dont les valuations des contraintes sont toutes de coût 1 (classe notée *MAX-CSP* [FW92]).

Les schémas d'approximation pour *MAX-SAT* garantissent d'être à moins de 25% de l'optimum en utilisant une amélioration d'un algorithme glouton [Joh74] par une relaxation en programmation linéaire [GW93b], et même à moins de 7% par une relaxation semi-définie en programmation non linéaire pour *MAX-2SAT* (toutes les clauses sont des disjonctions de deux littéraux) [FG95].

Cependant, ces schémas ne s'étendent pas directement aux Σ-VCSP à cause de la conversion, pour au moins deux raisons. D'une part, les clauses sont indifféremment satisfaites ou insatisfaites. Or la conversion d'un *MAX-CSP* en *MAX-SAT* impose de respecter certaines clauses indiquant que chaque variable de *MAX-CSP* doit être affectée. D'autre part, *MAX-CSP* ne traite pas les contraintes dures : elles ne sont en aucun cas assurées d'être satisfaites.

Soit P, un problème Σ-VCSP avec des domaines de taille d égaux à 3. [Lau96] décrit une méthode utilisant une relaxation semie-définie, résolvant le problème P transformé en un système d'équations quadratiques, pour en extraire une solution A telle que $\varphi(A,P) < 2.45 \times \varphi(P)$. Pour $d > 3$, sa méthode ne garantie seulement que $\varphi(A,P) < k \cdot \varphi(P)$.

2.3.3 Coopération des méthodes complètes et incomplètes

Nous séparons cette présentation suivant que l'objectif est d'aboutir à une méthode complète ou à une méthode incomplète. Une présentation d'un panorama plus exhaustif de schémas de coopération est faite dans [LL97b].

Méthodes complètes profitant des méthodes incomplètes

Une méthode complète est accélérée si elle part avec un bon majorant, ou si elle emploie de bonnes heuristiques sur les variables ou sur les valeurs.

Toute méthode incomplète est en mesure de produire une solution dont la valuation est un majorant de l'optimum et donc une borne de départ pour une recherche arborescente ; voir par exemple [CVMB96, Lob96].

Cependant, la connaissance d'un majorant n'accélère réellement le temps de résolution d'un *Branch and Bound* que si le majorant est très proche de l'optimum.

La figure 2.17 montre l'effet d'un majorant α donné comme coupe initiale à deux algorithmes : un algorithme *Forward Checking* valué avec heuristiques dynamiques et un algorithme *Russian Doll Search*. Les expérimentations portent sur 44 problèmes Σ-VCSP aléatoires binaires de petite taille (15 variables, 10 valeurs). Ces problèmes ont tous un optimum égal à 20. Nous remarquons que seule une coupe initiale proche de l'optimum permet de réduire le nombre de noeuds explorés et donc le temps de résolution.

Dans le cadre des problèmes de satisfaction, [MGS96] présentent une heuristique dynamique sur les variables particulièrement efficace pour détecter rapidement si un problème est incohérent. Ils enregistrent, pour chaque contrainte, le nombre de fois où elle est violée au cours d'une recherche locale. Les variables apparaissant dans les contraintes les plus violées sont insérées en premier dans l'ordre ; elles forment un “noyau dur” accélérant *a priori* la preuve de l'incohérence.

[CVMB96] proposent une heuristique statique sur les valeurs[20], obtenue par examen d'un état stable fourni par l'algorithme de recuit en champ moyen. Dans l'exemple de la figure 2.15,

[20] L'algorithme de recuit en champ moyen fournit aussi des indications pour construire un ordre sur les variables : vitesse de convergence des variables, ainsi que nombre de valeurs dans d' par variable.

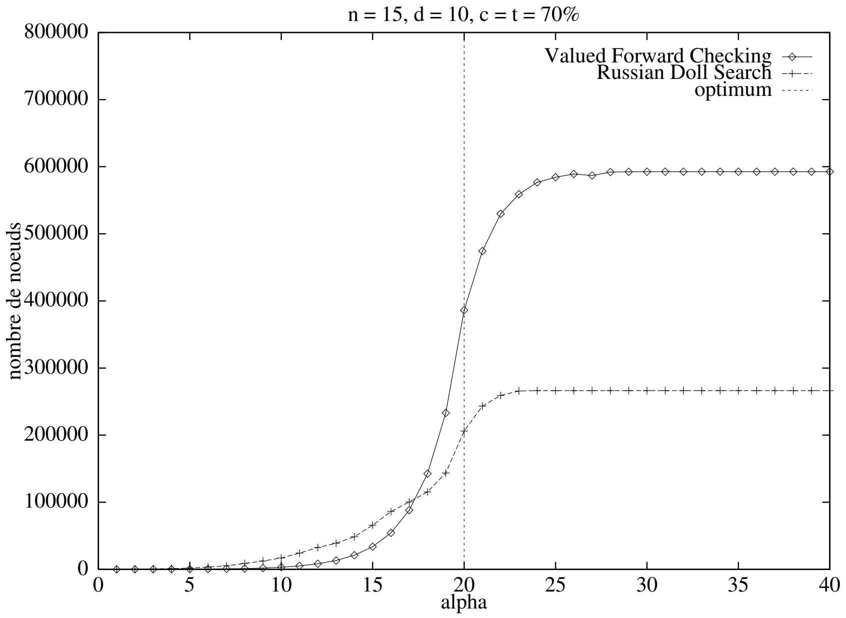


Fig. 2.17 - Effet d'une coupe initiale pour des méthodes complètes ; nombre moyen de noeuds pour chercher une solution optimale de valuation strictement inférieure à α ou prouver qu'il n'y en a pas

l'ordre sur les valeurs est obtenu en classant les probabilités $p(v, val)$ peu avant la convergence de l'algorithme. L'ordre pour la figure de gauche (dureté $t = 16\%$) est 3,1,5,7,4,9,6,10,2,8 ; l'ordre pour la figure de droite (dureté $t = 15\%$) est 2,5,4,6,1,3,10,8,7,9.

Nous avons étudié une application du recuit en champ moyen à chaque noeud développé par une recherche arborescente. Il s'agit d'obtenir une heuristique dynamique sur les variables et les valeurs. La difficulté réside dans le bon compromis à trouver entre le surcroît de calcul et en contrepartie la réduction de l'arbre de recherche pour avoir effectivement un gain en performance[21].

Un compromis consiste à effectuer une descente en température assez brutale et à calculer cette heuristique uniquement sur les noeuds proches de la racine (zone où la probabilité d'erreur pour des heuristiques plus simples est la plus grande). Une comparaison des calculs effectués montre que les heuristiques Γ et Ω, présentées en 2.3.1.3, effectuent un travail similaire à un algorithme d'approximation en champ moyen exécuté sur une seule itération ; ce travail revient à examiner une probabilité de conflit par valeur. L'heuristique *Min-Conflicts* proposée dans [MJPL92] examine les conflits par valeur par rapport à une seule instanciation complète servant de référence.

Les expérimentations ont été faites sur des problèmes CSP. Nous ajoutons un renforcement d'arc-cohérence sur les noeuds où le recuit en champ moyen est exécuté[22]. Ainsi, la recherche locale bénéficie d'un problème simplifié au niveau des valeurs (les valeurs incohérentes étant supprimées). La figure 2.18 compare des algorithmes *Forward Checking* avec trois heuristiques

[21] La même problématique est abordée dans [DP88] pour une heuristique fondée sur le nombre de solutions des extensions possibles de l'instanciation partielle courante d'un CSP, à partir de la construction *partielle* d'un arbre couvrant.

[22] L'implémentation du maintien d'arc-cohérence est réalisée simplement par *MAC3* ; cependant ce maintien n'est utilisé que sur peu de noeuds ; donc le choix de *MAC3* semble guère pénalisant.

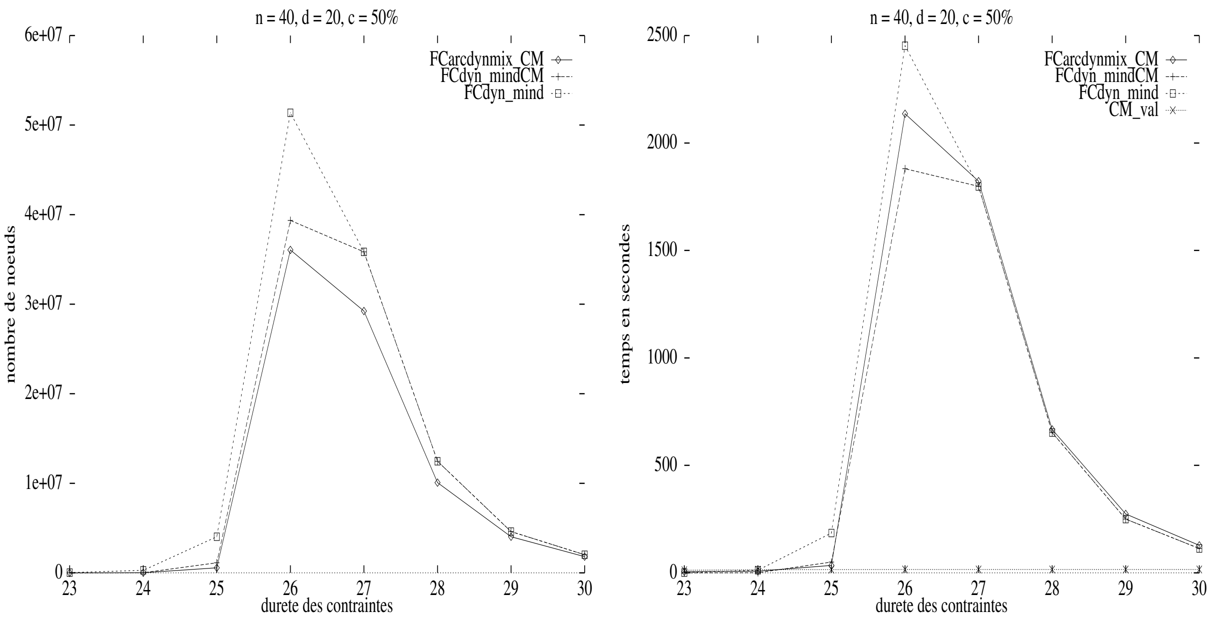


Fig. 2.18 - Exemples de coopération entre méthodes complètes et incomplètes ; moyennes des nombres de noeuds et des temps d'exécution sur 100 CSP (40 variables)

différentes :

- FCdyn_mind : heuristique dynamique classique pour les variables (*domaine min, degré max*), aucune heuristique pour les valeurs (ordre initial) ;
- FCdyn_mindCM : heuristique dynamique classique pour les variables (*domaine min, degré max*) et heuristique statique pour les valeurs, obtenue par une phase de pré-traitement, notée CM_val, effectuant un recuit en champ moyen ;
- FCarcdynmix_CM : maintien d'arc-cohérence (*MAC3*) et heuristique dynamique pour les variables et les valeurs par recuit en champ moyen sur les noeuds proches de la racine, sinon identique à FCdyn_mindCM.

En terme de nombre moyen de noeuds explorés, l'heuristique dynamique que donne le champ moyen guidé par le renforcement d'arc-cohérence apporte un gain non négligeable avant, pendant et après le pic de complexité (zone de transition entre les problèmes cohérents et les problèmes incohérents) que se soit par rapport à FCdyn_mind ou FCdyn_mindCM. En revanche, en temps de résolution, le gain de FCarcdynmix_CM par rapport à FCdyn_mind est plus faible et seulement avant et pendant le pic ; le gain de FCarcdynmix_CM par rapport à FCdyn_mindCM a lieu uniquement avant le pic (en moyenne, FCarcdynmix_CM prend 34.8 secondes à la dureté 25, contre 50.3 secondes pour FCdyn_mindCM, sans compter le temps du pré-traitement).

Méthodes incomplètes profitant des méthodes complètes

La section 2.3.2.2 décrit des techniques qui partent d'une méthode de recherche arborescente, y intègrent des mécanismes heuristiques et aléatoires issus des méthodes de recherche locale pour aboutir à un algorithme incomplet, plus performant en temps limité.

Un autre type de schéma de coopération consiste à faciliter le travail d'une méthode de recherche locale. Une façon de simplifier un problème est d'effectuer en pré-traitement un renforcement d'arc-cohérence sur les contraintes dures pour éliminer des valeurs incohérentes. [KD95] réalisent un pré-traitement par renforcement de chemin-cohérence affaiblie pour des problèmes de satisfaction structurés, ce qui a pour effet de mieux guider un algorithme de recherche locale, grâce aux nouvelles contraintes induites par le renforcement.

Enfin, des méthodes complètes peuvent être employées pour créer le voisinage d'une méthode de recherche locale, comme c'est le cas pour l'algorithme *Guided Local Search* qui crée un arbre de recherche limité en taille, en faisant un parcours "meilleur" en premier et en sélectionnant le meilleur noeud pendant [VAL96]. De même, une méthode complète peut être utilisée pour se déplacer dans un espace d'instanciations complètes admissibles (respectant les contraintes dures).

2.4 Expérimentations des méthodes complètes sur des Σ-VCSP : principales difficultés observées

Nous nous intéressons dans cette section uniquement à mieux comprendre les principales difficultés rencontrées dans le cadre des Σ-VCSP, pour un opérateur d'agrégation non idempotent[23]. C'est ce cadre des problèmes d'optimisation additifs que nous choisirons comme cadre applicatif pour les chapitres suivants. Les résultats qui suivent ont également été présentés dans [VdG97].

2.4.1 Difficulté par rapport à la satisfaction

Nous avons aperçu sur l'exemple du robot, une faible différence de nombre de noeuds développés par un algorithme *Backtrack* (12 noeuds, fig. 2.7) dans le cadre de la satisfaction, par rapport à un algorithme *Depth-First Branch and Bound* (14 noeuds, fig. 2.8) dans le cadre de l'optimisation. Sur des problèmes de plus grande taille, cette différence s'accentue largement.

Pour comparer expérimentalement satisfaction et optimisation, nous étudions le comportement d'un algorithme *Forward Checking* (valué, dans le cas de l'optimisation) avec des heuristiques dynamiques classiques. Les performances sont comparées sur 100 problèmes aléatoires binaires de taille moyenne pour une dureté uniforme des contraintes allant de 0% à 100%.

La figure 2.19 de gauche montre un pic de complexité observé sur des problèmes de satisfaction de contraintes à la frontière entre cohérence (pour une dureté inférieure à 26%) et incohérence (pour une dureté supérieure à 26%). Ce pic s'observe aussi bien lorsque la dureté des contraintes varie et la connectivité du graphe des contraintes reste fixe, que lorsque la connectivité varie avec une dureté fixée. Avant et après le pic, le CSP est relativement facile à résoudre : soit une solution est rapidement trouvée ; soit la preuve de l'incohérence est établie en instanciant quelques variables.

Pour un problème CSP, le pic de complexité semble intervenir lorsque le nombre estimé E de solutions du CSP est égal à 1 [Smi94]. Dans ce cas, une formule permet de relier la dureté et la connectivité, pour un CSP binaire aléatoire [Pro94] :

$$E = d^n(1-t)^{c\frac{n(n-1)}{2}} = 1 \Leftrightarrow t = 1 - d^{\frac{-2}{c(n-1)}}$$

[23] Une comparaison avec d'autres classes de VCSP sera faite dans la section 4.2 sur des simplifications de problèmes.

Le passage à l'optimisation fait disparaître cette notion de pic de complexité pour la remplacer par une évolution beaucoup plus régulière, voire exponentielle, de la difficulté de résolution lorsque la dureté ou la connectivité augmentent. Dans la zone de cohérence, l'évolution d'un Σ-VCSP est similaire à celle d'un CSP, les mécanismes de prise en compte de l'incohérence ralentissent seulement d'un facteur constant les temps de résolution. En revanche, à partir de l'incohérence, la résolution d'un Σ-VCSP devient incroyablement plus dure que celle d'un CSP.

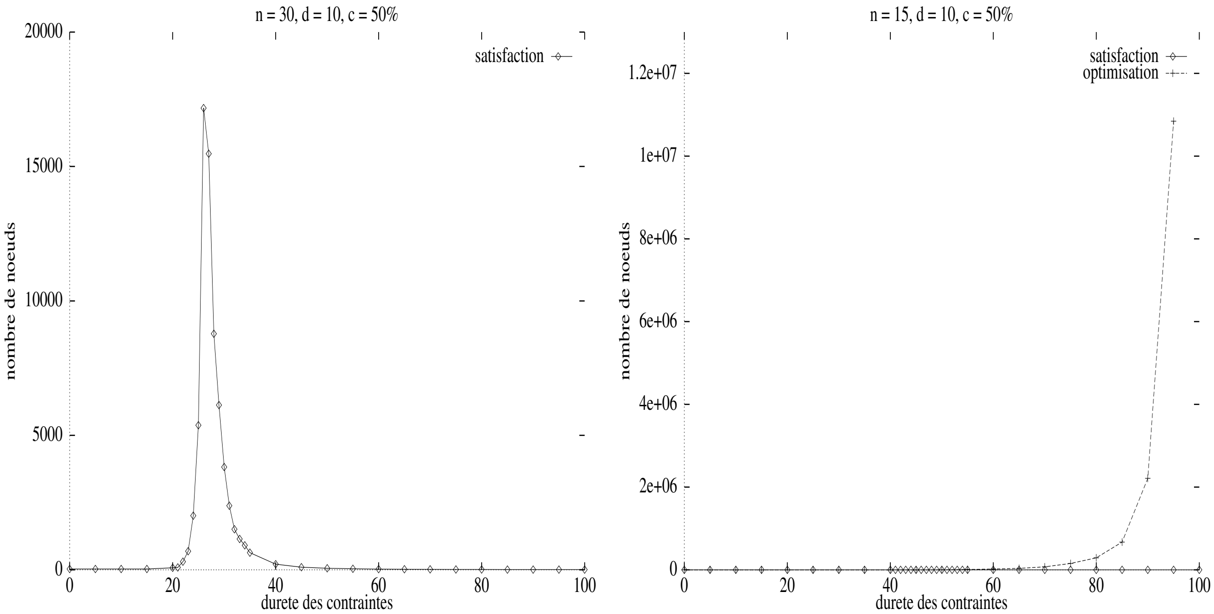


Fig. 2.19 - Pic de complexité en satisfaction ; moyennes sur 100 problèmes $\wedge$-VCSP (30 variables) résolus par *Forward Checking* ; et "montagne" de complexité en optimisation ; moyennes sur 100 problèmes Σ-VCSP (15 variables) résolus par *Forward Checking* valué

La figure 2.19 de droite compare la résolution par un algorithme *Forward Checking* d'un même problème, en satisfaction, et en optimisation (contraintes de coût égal à 1, opérateur d'agrégation additif). Le problème implique seulement 15 variables contre 30 pour la figure de gauche. Néanmoins, en moyenne, plus de 2 millions de noeuds sont nécessaires pour résoudre en optimisation un problème à la dureté 90%, alors que la résolution ne prends pas plus de 600 noeuds en satisfaction, quelles que soient les duretés testées.

Qu'en est-il avec d'autres algorithmes exacts, utilisant une meilleure évaluation de la valuation globale d'une instanciation partielle ? La figure 2.20 montre, sur les mêmes problèmes à 15 variables, l'évolution en nombre moyen de noeuds pour un algorithme *Forward Checking* valué avec *Directed Arc-Consistency Counts* et pour un algorithme *Russian Doll Search* ; l'heuristique statique sur les variables choisit en premier celle de plus grand degré. Bien que le nombre de noeuds diminue à partir d'une dureté élevée (ici, supérieure à 95%)[24], l'évolution reste globalement similaire à un simple *Forward Checking* valué, c'est à dire augmente fortement pour des problèmes incohérents.

[24] Cette décroissance ne ressemble guère à une transition de phase, comme le suggère [LM96b] ; d'ailleurs, aucune décroissance n'est observée lorsque la connectivité augmente.

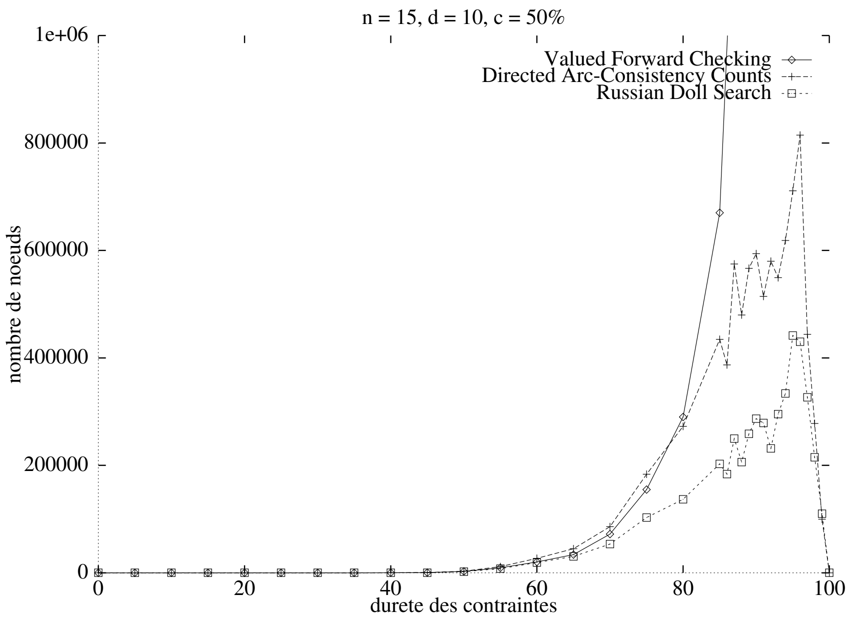


Fig. 2.20 - Comportement de méthodes complètes en optimisation ; moyennes sur 100 problèmes Σ-VCSP (15 variables)

2.4.2 Difficulté à obtenir l'optimum

Pour mieux comprendre ce qui rend si difficile la résolution d'un problème Σ-VCSP , nous présentons une autre façon de poser le problème de la résolution.

Soit P un Σ-VCSP, soit $\varphi(P)$ son coût optimal et soit $P(\alpha)$ le problème de décision NP-complet suivant : existe-t-il une instanciation complète des variables de P telle que son coût soit *strictement* inférieur à α ?
La réponse à ce nouveau problème est soit non, dans ce cas α est un minorant de l'optimum, soit oui, et dans ce cas α est un majorant.

La figure 2.21 décrit une moyenne en nombre de tests de contraintes pour 44 problèmes ($n = 15, d = 10, c = t = 70\%, \forall c \in C, \varphi(c) = 1$) ayant tous un optimum $\varphi(P) = 20$. La courbe qui décrit un pic est obtenue par un algorithme *Forward Checking* valué avec heuristiques dynamiques. La courbe de droite (notée *SA*) est obtenue par un algorithme de recuit simulé. En abscisse, c'est le paramètre α qui varie de 1 à 40.

Nous observons à nouveau une transition de phase située autour de l'optimum, et séparant les problèmes dont la réponse est non, des problèmes dont la réponse est oui.
Cette courbe nous éclaire sur plusieurs points :

- le problème le plus dur à résoudre est le problème $P(\varphi(P))$; il consiste à prouver qu'il n'y a pas de solution de valuation strictement inférieure à $\varphi(P)$; il est plus dur que le problème $P(\varphi(P)+1)$ correspondant à trouver une solution optimale ;
- une méthode exacte comme l'algorithme du *Branch and Bound* qui produit une solution optimale et une preuve de son optimalité doit nécessairement résoudre les deux problèmes $P(\varphi(P))$ et $P(\varphi(P)+1)$ qui sont précisément les deux plus durs ;

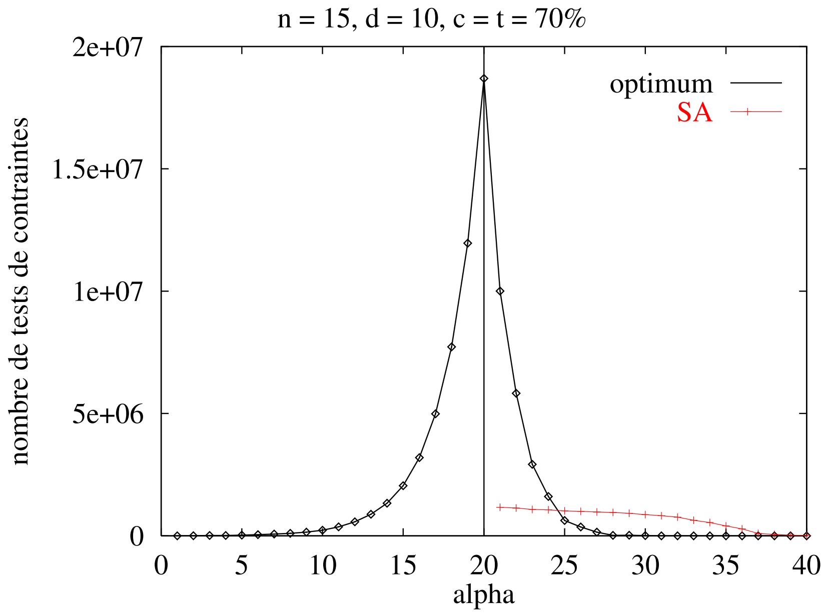


Fig. 2.21 - Pic de complexité en optimisation ; nombre moyen de tests de contraintes pour chercher une instanciation complète de valuation strictement inférieure à α

- à la même distance de l'optimum, produire un minorant semble plus difficile que produire un majorant ; c'est d'autant plus vrai lorsqu'une méthode incomplète est utilisée pour produire plus rapidement une bonne solution ;
- la difficulté pour encadrer l'optimum semble croître de façon exponentielle lorsque l'encadrement se resserre.

Remarque 10 Les observations que nous venons de faire sont également valables quelle que soit la classe des VCSP choisie. Cependant, dans le cas d'un opérateur d'agrégation non idempotent, cas des Σ-VCSP, nous manquons justement d'outils pour travailler au pic de complexité $\varphi(P)$. En effet, les calculs d'un minorant de la valuation globale d'une instanciation partielle ne sont pas assez pertinents pour couper efficacement dans l'espace de recherche. ◇

Conclusion

Finalement, chercher l'optimum et surtout prouver son optimalité revient à s'attaquer aux problèmes les plus difficiles. En temps contraint, il est seulement possible d'approcher l'optimum. Les méthodes incomplètes du type recherche locale sont de bonnes candidates pour produire un majorant. Les méthodes complètes permettent de garantir un minorant ; mais elles sont a priori plus lentes. Nous verrons dans la seconde partie, des techniques pour accélérer la production de minorants. Le principe repose sur l'utilisation d'une méthode complète appliquée à une simplification du problème.

Chapitre 3

Optimisation sous contraintes en temps limité

Dans ce chapitre, nous étudions la résolution de problèmes d'optimisation sous contraintes dans un contexte opérationnel où le temps de résolution est limité.

Premièrement, nous étudions la résolution d'un seul problème en temps limité. Il s'agit de produire à temps des résultats partiels de bonne qualité. Dans une approche interruptible, il est important de pouvoir **évaluer la qualité** de ces résultats et de **prédire l'évolution de la qualité au cours du temps**. Des éléments de réponse ont été également présentés au *Workshop on Building Resource-Bounded Reasoning Systems* [dGV97b].

Deuxièmement, nous abordons le cadre de systèmes temps réel composés d'un **ensemble de problèmes d'optimisation indépendants**. Nous proposons une **politique d'ordonnancement** fondée sur l'étude précédente du compromis entre temps et qualité. Nous présentons les résultats expérimentaux de notre approche appliquée aux problèmes de l'ordonnancement des prises de vue du satellite SPOT5.

3.1 Résolution d'un problème d'optimisation : compromis entre temps et qualité

Nature de la contrainte temporelle

Dans cette introduction, nous examinons le type de contrainte temporelle portant sur une résolution et ce que cela implique en terme d'approche algorithmique et de contrôle temporel.

Il s'agit de trouver une solution à un problème d'optimisation soumis à une échéance temporelle. Dans le cas où l'**échéance est stricte**, le contrat temporel est connu au début de la résolution. Il s'agit de produire la meilleure solution possible dans le temps imparti. C'est par définition une **approche contractuelle**.

En revanche, si l'**échéance est relative**, la résolution continue jusqu'au moment où la pénalité entraînée par le dépassement de l'échéance dépasse l'intérêt d'améliorer la qualité de la solution. C'est une **approche interruptible fondée sur la notion d'utilité**.

Enfin l'échéance peut être révisable. C'est le cas lorsque l'environnement extérieur dont dépend la contrainte temporelle se modifie, entraînant une mise à jour de l'échéance. Dans ce cas, la résolution doit proposer plusieurs solutions au cours du temps. C'est par définition une approche interruptible.

Dans tous les cas, il faut être capable de produire des résultats partiels quelle que soit la durée allouée à la résolution. De même, il est important d'évaluer la qualité de ces résultats, soit pour certifier un approche contractuelle, soit pour guider une approche interruptible.

Remarque 11 Dans un système temps réel intelligent, une tâche peut produire des résultats partiels ou exacts (cf. la notion de qualité de la section 1.1.3.2). Pour un problème d'optimisation, un résultat partiel correspond à une solution admissible sous-optimale du problème. Un résultat exact ou complet correspond à une solution optimale.

◇

3.1.1 Obtention de résultats partiels

3.1.1.1 Comparaison des méthodes complètes et incomplètes

Nous avons vu dans le chapitre 2 que les problèmes d'optimisation sous contraintes peuvent s'exprimer sous la forme de *Problèmes de Satisfaction de Contraintes Valuées*. Ce formalisme se caractérise par un fort pouvoir d'expression. Son caractère générique a permis de développer des méthodes de résolution indépendantes des domaines d'application. Il existe deux grandes classes de méthode : les méthodes complètes, capables de produire une solution prouvée optimale, et les méthodes incomplètes (cf. section 2.3).

Les méthodes complètes ont comme base l'algorithme *branch and bound* avec une recherche arborescente en profondeur d'abord[1]. Les méthodes incomplètes sont, soit des recherches locales, telles que l'algorithme du recuit simulé, la recherche tabou ou les algorithmes génétiques, soit des recherches arborescentes rendues incomplètes, telles que l'algorithme *Limited Discrepancy Search* ou l'algorithme *Iterative Broadening*.

Par la suite, nous ferons l'hypothèse que le problème à résoudre est un problème d'optimisation pur, sans contrainte dure, ou bien, s'il y a des contraintes dures, qu'il est facile de les satisfaire . Dans cette hypothèse, les méthodes complètes ou incomplètes produisent toujours une ou plusieurs solutions avant de terminer.

La figure 3.1 résume le comportement généralement observé des méthodes complètes et des méthodes incomplètes. Des expérimentations montrant ce comportement ont été effectuées dans [WF95, Ver95]. Chaque courbe représente l'évolution en fonction du temps de la valuation de la meilleure solution dernièrement produite par une méthode. L'optimum est représenté par une ligne horizontale en pointillé. Rappelons que la résolution des problèmes VCSP vise à minimiser la violation des contraintes, donc les valuations des solutions sont des majorants de l'optimum.

Le temps de calcul T_{mi} d'une méthode incomplète est connu et configurable par un critère d'arrêt. Par exemple, pour une recherche locale simple de type descente aléatoire (cf. *Random Flip* dans la section 2.3.2.1), T_{mi} est borné[2] par le produit du nombre de *flips* par le temps d'exécution d'un *flip*. En revanche, le temps de calcul T_{mc} d'une méthode complète est inconnu et évolue

[1] Les recherches en largeur ou meilleur en premier ne sont pas considérées du fait de leur importante occupation mémoire et du fait quelles ne produisent pas rapidement une première solution.

[2] Dans le cas où le problème est satisfiable, la résolution peut s'arrêter avant la durée prévue de la recherche.

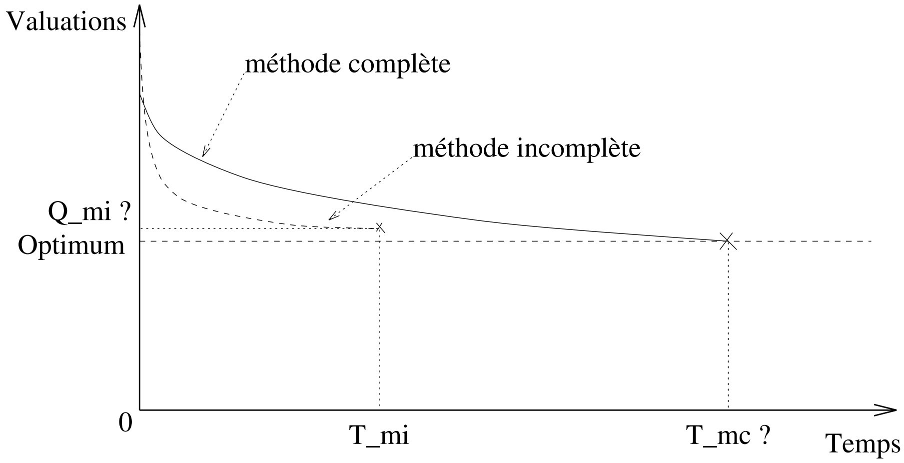


Fig. 3.1 - Comparaison des méthodes complètes et incomplètes

de façon exponentielle en fonction de la taille du problème. Au pire, la recherche arborescente explore tout l'espace de recherche, soit d^n noeuds, où n est le nombre de variables et d la taille des domaines.

Nous observons généralement que pour un temps de calcul donné bien inférieur à T_{mc}, les méthodes incomplètes produisent des solutions de meilleure valuation que les méthodes complètes. Ce constat se vérifie d'autant plus que la taille des problèmes augmente. Dans certains cas, cependant, les méthodes complètes peuvent s'avérer plus efficaces que les méthodes incomplètes : soit lorsqu'il existe une heuristique bien adaptée au problème, soit sur les petits problèmes (moins de 20 variables pour des Σ-VCSP binaires aléatoires), soit sur des problèmes très structurés (par exemple, ayant une très faible largeur de bande), soit lorsque le problème comporte beaucoup de contraintes dures en plus des contraintes de préférence car le renforcement et le maintien de cohérence locale sur les contraintes dures accélère les méthodes complètes.

Par ailleurs, dans les deux cas, méthodes complètes ou incomplètes, nous constatons aussi que la diminution de la valuation des solutions produites est plus forte en début de résolution qu'à la fin. De plus, les méthodes complètes bénéficient d'heuristiques de choix des valeurs permettant de produire une première solution généralement meilleure que celle d'une méthode incomplète[3].

Lorsque la méthode complète termine, une solution optimale est trouvée, sa valuation est égale à l'optimum. En revanche, lorsque la méthode incomplète termine, nous ne savons pas à quelle distance de l'optimum se trouve la solution retournée. Durant la recherche de solution, dans les deux cas, ces méthodes ne donnent aucune information sur la distance qui reste entre la meilleure solution trouvée et l'optimum. Cette information sur la qualité de la solution courante produite par une résolution est l'objet de notre étude et sera abordée à partir de la section 3.1.2.

Remarque 12 Une méthode locale est incapable de prouver l'optimalité d'une solution. Elle peut donc perdre du temps à essayer d'améliorer une solution optimale, ou une solution très proche de l'optimum et de qualité jugée suffisante pour être délivrée à l'environnement extérieur (le système ou un utilisateur).

[3] Dans le cas d'un algorithme de recuit simulé, la première solution est générée aléatoirement. Il faut quelques *flips* avant d'améliorer sensiblement sa valuation.

◇

Bilan

Dans une approche contractuelle, les méthodes incomplètes représentent le meilleur choix pour résoudre en temps limité un problème, à l'exception de certains cas faciles pour les méthodes complètes.

Dans une approche interruptible, les méthodes incomplètes classiques posent plusieurs problèmes dus à la variation imprévisible du temps alloué au calcul :

- comment régler les paramètres de la méthode incomplète et donc le temps T_{mi} ? En effet, une méthode incomplète arrêtée avant la fin de sa durée d'exécution peut fournir de moins bons résultats qu'une autre méthode incomplète prévue pour terminer plus tôt ;
- que faire lorsque l'échéance temporelle est très courte ?
- que faire lorsque l'échéance temporelle s'allonge, accordant plus de temps aux calculs ? Certaines méthodes incomplètes ne produiront pas de meilleurs résultats, même si leur de temps de calcul est rallongé.

La section suivante tente d'apporter des solutions à ces interrogations.

3.1.1.2 Adaptation des méthodes incomplètes au temps limité

Classement des méthodes incomplètes

A l'instar de l'approche *Design-to-time* [GL93b] (cf. section 1.2.1), il est possible d'exhiber un ensemble de méthodes incomplètes offrant différents compromis entre temps et qualité. Ainsi, pour résoudre un problème avec une certaine échéance temporelle stricte, nous avons à notre disposition un ensemble de méthodes et pour chacune la possibilité de faire varier ses paramètres de configuration, donc son comportement en fonction du temps.

Plusieurs expérimentations des méthodes incomplètes sur différents types de problèmes ont été réalisées : sur des problèmes aléatoires [Wal96a], sur des problèmes d'ordonnancement d'atelier [VAL96, Har95], sur des problèmes du CELAR [THL95, CVMB96], sur des problèmes SPOT5 [VB95]. Ces résultats permettent de classer les méthodes suivant deux axes : le temps et la qualité. Dans tous les cas, l'ordre est assez dépendant des problèmes à résoudre et de la façon dont sont codées les méthodes. De plus, il n'y a pas une seule méthode offrant la meilleure solution en le minimum de temps, mais différentes méthodes offrant la meilleure solution pour un intervalle de temps donné. Nous donnons un classement très empirique et incomplet des méthodes incomplètes décrites dans la section 2.3.2. Nous avons découpé le temps en 5 intervalles continus, le premier démarrant au temps zéro. A chaque intervalle sont associés les méthodes qui nous semblent les plus efficaces :

1. algorithmes gloutons guidés par une heuristique ;
2. recherche locale simple (descente en gradient) ; méthodes arborescentes incomplètes (par exemple, les algorithmes *Limited Discrepancy Search* et *Heuristic-Biased Stochastic Sampling*) ;
3. algorithme de recuit en champs moyen ;

4. algorithme de recuit simulé ; algorithme *Tabu Search* ; algorithme *Guided Local Search* [VT96, TV97]

5. algorithmes génétiques.

Simplification des problèmes

Lorsque l'échéance temporelle est très courte et que la taille du problème à résoudre est très grande, les méthodes incomplètes ne peuvent plus produire une solution à temps. Il faut réduire la complexité du problème et appliquer la résolution sur une simplification du problème.

Les simplifications utilisent des connaissances sur le domaine d'application pour savoir ce qu'il est important de conserver et ce qui peut être ignoré. La phase de simplification travaille donc en amont du formalisme VCSP, dans un modèle plus riche de représentation du problème. De façon générale, [SP94] propose quatre axes de simplification pour des problèmes de planification : élagage, ordre, approximation et portée (cf. section 1.2.1.1).

Ces techniques peuvent se traduire en terme de modification des composantes (V, D, C, S, φ) d'un VCSP (cf. Simplifications avec modification du problème [Ver95]) :

- ensemble des variables V : retirer des variables dont la valeur est jugée moins importante (par exemple, en planification, les dernières variables correspondant à des décisions d'actions lointaines) ; affecter des variables ; décomposer l'ensemble des variables en sous-ensembles distincts ;

- ensemble des valeurs D : éliminer des valeurs jugées mauvaises ; regrouper des valeurs (avec simplification correspondante des contraintes) et obtenir une discrétisation plus grossière ; à l'inverse, ajouter des valeurs (accroître les ressources en planification) ;

- ensemble des contraintes C : retirer des contraintes d'importance secondaire ; diminuer la dureté des contraintes ;

- structure de valuation S : simplifier l'objectif à minimiser (par exemple, passer d'une structure additive à une structure possibiliste) ;

- fonction de valuation φ : modifier la valuation associée aux contraintes pour simplifier le paysage énergétique (à l'instar des techniques *Clause Weight* [SK93] et *Guided Local Search* [Vou97] servant à sortir des optima locaux).

L'inconvénient de ces simplifications est que les bonnes solutions du problème simplifié ne sont pas forcément de bonnes solutions pour le problème initial. Ceci est atténué lorsque des connaissances spécifiques au domaine d'application du problème permettent de guider les simplifications. Par exemple, en planification, les variables de décision correspondant à des actions lointaines pourraient être ignorées dans un premier temps (travail à horizon limité). Cependant, cela se fait au détriment de la généricité.

Un argument en faveur des simplifications est la difficulté de modélisation des problèmes qui impose souvent d'effectuer une première série de simplifications par rapport au modèle réel. Dans un système temps réel composé d'une phase d'acquisition des données du problème puis d'une phase de résolution du problème, les deux phases sont traitées de façon incomplète. La première phase peut ne fournir qu'un modèle simplifié. Donc, de toutes façons, l'optimisation se fera sur un problème simplifié par rapport au problème réel. De plus, lorsque l'environnement évolue, modifiant dynamiquement les données du problème, une résolution très simplifiée mais rapidement réactualisée est plus intéressante qu'une résolution plus optimale d'un problème dépassé. C'est une des motivations de l'approche entrelaçant planification à profondeur limité et exécution du plan partiel (cf. figure 1.10 de la section 1.2.3.1).

Algorithme interruptible

Nous avons vu que les méthodes incomplètes et les simplifications de problèmes peuvent être sélectionnées en fonction du contrat temporel. Pour passer d'une approche contractuelle à une approche interruptible, nous utilisons le théorème 1.3.3.3. Il suffit d'enchaîner des résolutions avec des contrats de plus en plus longs, en sélectionnant la méthode la mieux adaptée à chaque contrat.

Par exemple, le modèle *Great* [Mou93] applique des approximations successives de moins en moins grossières conduisant à un raisonnement de plus en plus précis (cf. section 1.2.2). Nous pouvons appliquer ce même principe de simplifications successives aux problèmes VCSP et le combiner avec une succession de méthodes incomplètes et de valeurs de leurs paramètres de configuration.

En plus de la possibilité de s'adapter à des contrats temporels révisables, la solution d'enchaîner des résolutions s'avère également efficace lorsque le contrat est long. En effet, pour plusieurs méthodes incomplètes, à partir d'une certaine durée du contrat, il est plus intéressant de ré-exécuter plusieurs fois la même méthode (qui repart de zéro ou profite des solutions précédentes) plutôt que de faire une seule exécution avec des paramètres ajustés pour tirer au maximum parti du temps de contrat.

3.1.2 Informations sur la qualité des résultats partiels d'une résolution et leur évolution

Notion de qualité

La qualité d'un résultat partiel est une mesure objective d'un critère évaluant ce résultat. Elle permet de comparer deux résultats entre eux et aussi d'évaluer un résultat partiel par rapport à un résultat complet. Les principaux critères sont l'exactitude, la spécificité et la certitude (cf. section 1.1.3.2).

En optimisation, c'est principalement le critère d'exactitude qui est mesuré et qui est fonction des contraintes satisfaites. Le critère de spécificité correspondrait au degré de détails considérés dans l'expression du problème ; il dépendrait des simplifications réalisées. Enfin, le critère de certitude nécessiterait la prise en compte de probabilités associées à l'existence des contraintes du problème.

Pour les VCSP, la valuation d'une solution est une mesure du critère d'exactitude. Plus précisément, la qualité, notée ρ', est une fonction complémentaire de la valuation. Soit une solution A d'un problème P ; $C_{sat(A,P)}$, l'ensemble des contraintes satisfaites par A dans P et $\omega = \otimes_{c \in C} \varphi(c)$, l'agrégation de la valuation de toutes les contraintes de P :

$$\rho'(A, P) = \bigotimes_{c \in C_{sat(A,P)}} \varphi(c) = \omega - \varphi(A, P) \qquad (3.1)$$

Cette définition ne permet pas d'évaluer une solution par rapport à l'optimum. Pour cela, il est nécessaire de connaître l'optimum, ou un minorant de l'optimum. Nous définissons la notion de **qualité normée exacte** , notée ρ, et de **qualité normée approchée** , notée ϱ, suivant que nous connaissons l'optimum $\varphi(P)$ de P ou seulement un minorant, noté $lb(P)$:

$$\rho(A,P) = \frac{\omega - \varphi(A,P)}{\omega - \varphi(P)} \tag{3.2}$$

$$\varrho(A,P) = \frac{\omega - \varphi(A,P)}{\omega - lb(P)}, \text{ avec } lb(P) \leq \varphi(P) \tag{3.3}$$

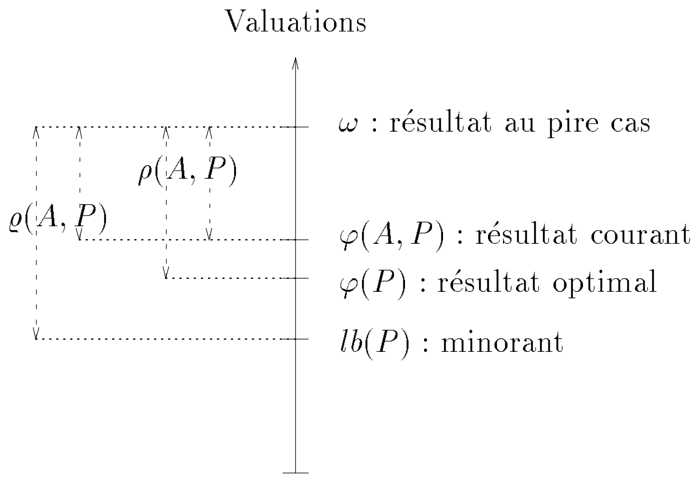


Fig. 3.2 - Qualité normée exacte ρ et approchée ϱ d'une solution A d'un problème P

La qualité normée est donc le rapport entre la distance à l'optimum de la solution et une distance maximale, correspondant au pire cas (cf. figure 3.2). De plus, la qualité approchée ϱ est un minorant de la qualité exacte ρ : $\forall A, P, \varrho(A,P) \leq \rho(A,P)$. Enfin, la qualité normée est bien une mesure variant entre 0, correspondant à la violation de toutes les contraintes, et 1, correspondant à la satisfaction maximale des contraintes (au sens de l'opérateur d'agrégation). Notons que dans le cas où l'optimum de P est égal à l'agrégation des valuations de toutes les contraintes ($\varphi(P) = \omega$) alors nous posons que la qualité est égale à 1.

Dans une approche interruptible, il est nécessaire de pouvoir estimer la qualité normée d'une solution et de prédire son évolution en fonction du temps. Les connaissances sur l'évolution de la qualité peuvent être acquises soit par une phase d'apprentissage hors contrainte de temps, cf. section 3.1.2.1, soit par une estimation durant la résolution, cf. section 3.1.2.2. Enfin, nous examinerons quelles garanties logiques et temporelles offrent ces deux approches, cf. section 3.1.2.3.

3.1.2.1 Profils de qualité en optimisation ?

La phase d'apprentissage consiste à résoudre une grande variété de problèmes issus du domaine d'application à traiter. L'évolution de la qualité normée exacte ρ en fonction du temps est enregistrée, moyennée et classée pour obtenir des profils de qualité discrets probabilistes (cf.

section 1.3.1.2) pour chaque classe de problèmes.

Cet apprentissage pose de nombreuses difficultés :

- le choix de critères pertinents pour déterminer rapidement à quelle classe appartient un nouveau problème à résoudre ; ces critères doivent permettre de sélectionner le bon profil de qualité ;
- le volume de problèmes à résoudre durant la phase d'apprentissage pour extraire des profils suffisamment fiables ;
- la variance trop forte sur certains profils qui unît à la validité des informations qu'ils apportent ; la variance est parfois due à quelques cas exceptionnellement plus durs à résoudre, au sein d'une même classe de problèmes ; cette variance interdit de considérer des profils plus simples tels les profils analytiques ou les profils discrets ne conservant que l'information de qualité moyenne ;
- la spécificité de la base d'apprentissage qui doit être reconstruite ou étendue dès que la configuration des problèmes à traiter change ;
- le fait que l'optimum ne soit pas forcément calculable en temps raisonnable et donc que les profils de la qualité exacte ρ soient impossibles à construire ; dans ce cas, seuls des profils de la qualité approchée ϱ sont réalisables.

Remarque 13 L'algorithme de résolution interruptible obtenu par un enchaînement de méthodes incomplètes nécessite lui-même une phase d'apprentissage pour évaluer chaque méthode et déterminer le bon enchaînement en fonction de type de problème à résoudre.

$\diamond$

Remarque 14 Les profils de qualité dynamiques probabilistes, décrits par la fonction $\Delta t, q_t \mapsto Proba(q_{t+\Delta t})$, diminuent la variance. En contrepartie, il faut être capable d'évaluer la qualité de la solution à l'exécution. De plus, le couple de paramètre, $(\Delta t, q_t)$, au lieu de t seulement, augmente encore le volume d'expérimentations à effectuer pour avoir des probabilités significatives.

$\diamond$

Nous avons expérimenté l'acquisition de profils de qualité pour deux types de problèmes : les problèmes aléatoires et les problèmes SPOT5.

Profils pour des VCSP binaires aléatoires

Ces problèmes possèdent des critères de classification évidents : leurs paramètres de génération qui sont n, le nombre de variables, d, la taille des domaines, c, la connectivité du graphe des contraintes et t, la dureté des contraintes. Un paramètre supplémentaire a été ajouté : h, le pourcentage de contraintes dures par rapport au nombre total de contraintes $(n \cdot (n-1)/2)$, les autres contraintes étant molles. Toutes les contraintes molles ont une valuation de 1 ; c'est donc une classe de problèmes identiques aux *MAX-CSP* [FW92] lorsque $h = 0\%$.

Nous avons expérimenté une méthode incomplète très simple. Il s'agit d'un algorithme de type *iterative descent search* avec un voisinage obtenu en modifiant une seule variable et une sélection aléatoire parmi les meilleurs voisins. L'affectation initiale est aléatoire et dès que l'algorithme atteint un minimum local, il est réinitialisé. Cent problèmes ont été résolus avec une limite de temps fixée au plus à 500 secondes pour chacune des combinaisons de valeurs des paramètres de génération[4] :

$$\begin{aligned} n &\in \{5, 10, 20, 30, 40, 50\}, \\ d &\in \{2, 5, 10, 15, 20\}, \\ c &= 50\%, \\ t &\in \{10, 20, 30, 40, 50, 60, 70, 80, 90\}, \\ h &\in \{0, 25, 50, 75\} \end{aligned}$$

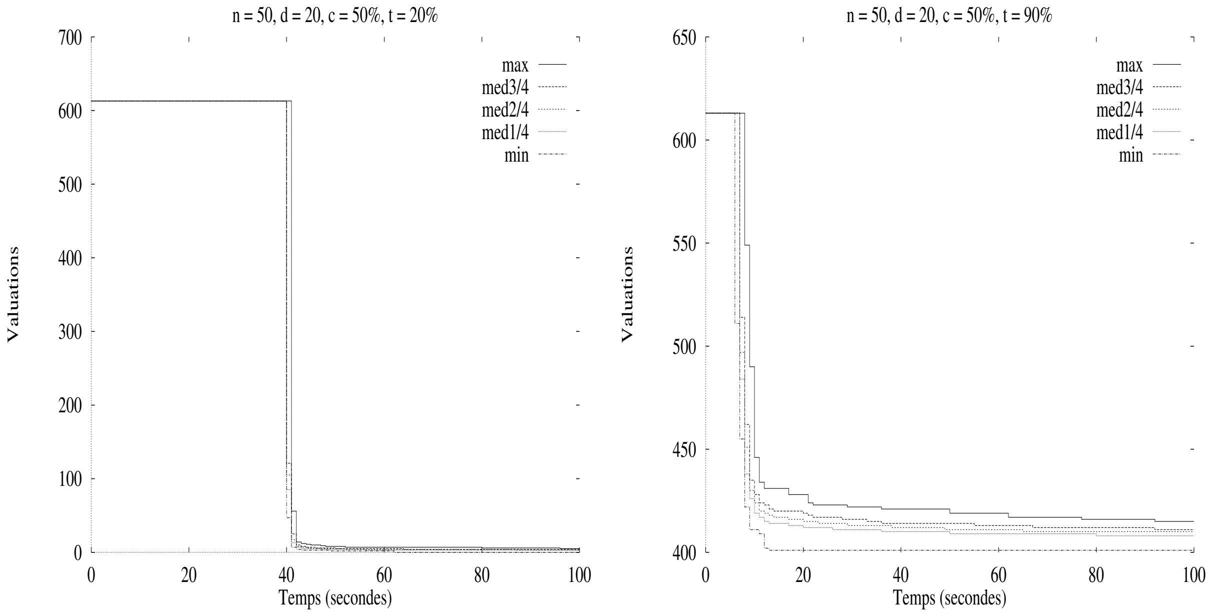


Fig. 3.3 - Profils de performance pour deux types de problèmes *MAX-CSP* aléatoires binaires, à dureté 20% et 90%, sans contraintes dures ($h = 0\%$)

Les résultats montrent des profils très précis : la variance est très faible, voir nulle, et l'optimum est pratiquement toujours atteint sur tous problèmes dont l'optimum a été prouvé (moins de 20 variables, ou sur la zone des problèmes cohérents). Sur des problèmes de plus grande taille, les méthodes complètes ne terminent pas. Plutôt que d'estimer la qualité pour pouvoir construire des profils de qualité approchée, nous présentons dans la figure 3.3 deux profils de performance, c-à-d une évolution moyenne de la valuation au cours du temps. A chaque instant est indiqué la valuation minimale, celle au premier quartile (médiane 1/2), celle médiane, celle au troisième quartile (médiane 3/4) et celle maximale (égale à la plus mauvaise valuation obtenue sur les 100 problèmes à cet instant). La méthode incomplète testée produit une amélioration brutale de la valuation (après un temps de chargement du problème) puis évolue très faiblement. Pour des problèmes presque cohérents (dureté égale à 20%), les valuations sont toutes proches de zéro à

[4] Cela a fait 2×108000 résolutions, par une méthode incomplète et par une méthode complète (un simple algorithme *branch and bound* avec *backward checking* et pas d'heuristiques) sur un réseau de stations *SUN* gérées par une application *PVM* spécialement écrite. Le temps global de résolution correspondrait à 230 jours sur une seule station *SUN Sparc5*.

la fin de l'exécution. La qualité est donc forcément bonne. En revanche, pour des problèmes fortement incohérent, un algorithme de recuit simulé, une autre méthode incomplète, améliore légèrement les résultats fourni par l'algorithme *iterative descent search* (environ 2%).

L'ajout de contraintes dures semble perturber d'avantage la méthode incomplète testée. Par exemple, pour $n = 40, d = 15, c = 50\%, t = 20\%$, tous les problèmes testés ont une solution de coût zéro, ce que trouve rapidement la méthode incomplète quand $h = 0\%$. En revanche, lorsque la proportion de contraintes dures augmente (uniquement en modifiant de façon aléatoire la valuation de $h \cdot n \cdot (n-1)/2$ contraintes des problèmes à $h = 0\%$), la méthode ne trouve plus toujours une solution cohérente. Il y a une erreur de 1 sur 2% des problèmes à $h = 25\%$, puis 18% à $h = 50\%$, et enfin 22% à $h = 75\%$.

Profils pour des problèmes réels SPOT5

Nous avons testé l'algorithme du recuit simulé sur une sélection de 10 problèmes SPOT5. A chaque fois, 10 résolutions sont effectuées pour chaque problème, sur un station *SUN Sparc5*. Les paramètres de configuration sont : $l = n \cdot d$, la taille d'un palier (égale au nombre de valeurs du problème), $g = 0$, la taille d'une descente en gradient (inefficace sur les problèmes SPOT5), $m = 0.99$, le coefficient de la suite décroissante pour la température (cf. "Exemple : l'algorithme de recuit simulé" section 2.3.2.1). C'est donc un algorithme de recuit assez lent qui est exécuté sur chaque problème.

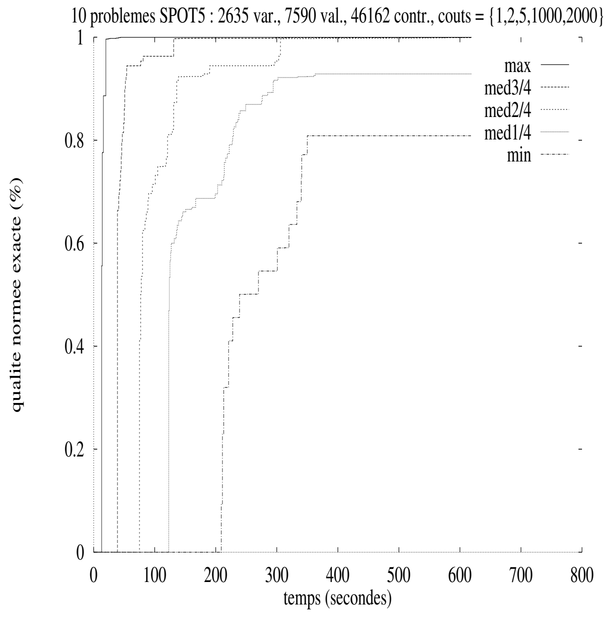


Fig. 3.4 - Profil de qualité sur 10 problèmes SPOT5

La figure 3.4 montre le profil de la qualité normée exacte ρ en fonction du temps avec son minimum, ses quartiles et son maximum. Ce profil reflète une variance très forte de l'évolution de la qualité entre les 100 exécutions de l'algorithme du recuit simulé. Bien sûr, chacun des dix problèmes a des caractéristiques différentes, comme le nombre de variables, le nombre de contraintes ou la distribution des coûts associés aux contraintes. Cependant, est-ce que ces critères sont suffisants pour différencier des problèmes ? Même si la phase d'acquisition génère un profil de qualité différent pour chaque problème SPOT5, opération très coûteuse, ce profil aura encore une variance assez forte dans certains cas. C'est ce que montre la figure 3.5. Ici, les profils sont établis uniquement à partir des 10 exécutions sur le même problème. Nous voyons que les médianes restent relativement distantes, surtout à la fin de l'algorithme, là où les meilleures solutions sont produites. Le fait qu'il y ait un mélange de contraintes dures et de contraintes de préférences semble augmenter la variance des méthodes incomplètes.

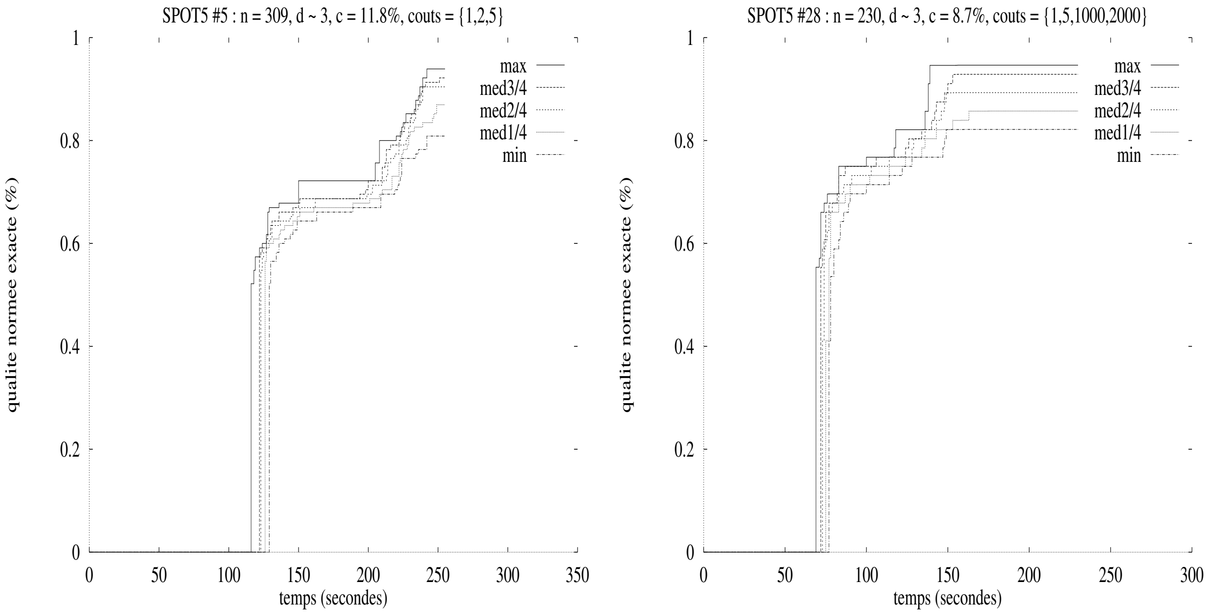


Fig. 3.5 - Profils de qualité pour deux problèmes SPOT5

3.1.2.2 Estimation à l'exécution

Estimation de la qualité courante

La qualité normée exacte est inconnue tant qu'une méthode complète n'a pas prouvée l'optimum. La qualité normée approchée ϱ est calculable, même si le minorant est nul, $lb(P) = 0$. Mais pour que cette qualité procure réellement une information pertinente, il faut un minorant qui s'approche de l'optimum.

La valeur $lb(P)$ d'un minorant est obtenue en effectuant une preuve. La preuve établit que le problème à résoudre viole au moins une certaine quantité de contraintes de valuation $lb(P)$. La construction d'une preuve est assurée par une méthode complète[5].

L'intérêt d'un minorant est d'être obtenu bien avant la fin d'une méthode complète appliquée sur le problème cible. Or, dans le cas général des VCSP, il n'existe pas de techniques garantissant l'obtention d'un minorant à $x\%$ de l'optimum en un temps borné polynomialement (cf. schéma d'approximation de la section 2.3.2.3). Les techniques pour produire des minorants reposent sur

[5] La construction de preuves par une méthode incomplète serait une voie de recherche à explorer.

des simplifications effectuées soit sur la méthode complète, soit sur le problème à traiter. La deuxième partie de la thèse étudie ces différentes simplifications. Nous pouvons déjà conclure que l'utilisation de méthodes complètes, même simplifiées, implique des temps de calcul exponentiels en fonction de la taille des problèmes et une difficulté supérieure pour produire des bons minorants par rapport à la production de bon majorants par des méthodes incomplètes.

Estimation de l'évolution de la qualité

A un instant t de la résolution, la distance à l'optimum $\varphi(P)$ de la meilleure solution trouvée A indique la progression maximale à venir de la qualité. La distance au meilleur minorant prouvé $lb(P)$ est un majorant de cette progression. L'estimation de l'évolution de la qualité nécessite d'estimer l'évolution du majorant $\varphi(A, P)$ et du minorant $lb(P)$.

Le temps pris par une méthode incomplète pour atteindre l'optimum est totalement inconnu. Nous ne connaissons que la durée d'exécution d'une méthode incomplète en fonction de ses paramètres de configuration. Seul un critère de difficulté à améliorer la qualité courante semble envisageable. Une phase d'apprentissage préalable peut nous fournir une classification grossière des méthodes en fonction des qualités des solutions atteintes. Le rang dans ce classement de la méthode incomplète en cours de résolution est un critère sur les chances d'amélioration de la qualité.

Un autre critère de difficulté serait de collecter le nombre de solutions trouvées en fonction de la valuation. Suivant un modèle probabiliste de décroissance de ce nombre, acquis préalablement par apprentissage, nous pourrions estimer le temps de résolution pour améliorer la qualité. Cette utilisation de l'historique de la résolution courante, combiné avec une base de connaissance, est proche des idées de profils dynamiques probabilistes décrits dans [HZ96] (cf. section 1.3.4).

Il existe *a priori* des méthodes plus efficaces pour estimer l'évolution du minorant $lb(P)$. En effet, pour certaines simplifications, nous pouvons en déduire à quel pourcentage de l'optimum sera le minorant. De plus, pour certaines méthodes complètes, la durée de résolution peut être estimée de façon assez fiable. Premièrement, le pourcentage de l'arbre de recherche exploré par une méthode complète fournit un indice de difficulté. Il est calculé en supposant que l'extension d'une instanciation partielle à une instanciation complète prenne le même temps pour chaque valeur de la dernière variable instanciée. Une autre approche repose sur des sondages dans l'arbre de recherche. Ces sondages permettent d'estimer, avec un intervalle de confiance, le nombre de noeuds développés et donc la durée de la résolution [Knu75, LL97c, LL98].

3.1.2.3 Prédictibilités logique et temporelle

Dans un système temps réel classique, le contrôle d'une tâche soumise à une échéance temporelle doit garantir que si la tâche est acceptée, elle sera complètement exécutée et respectera son échéance ou bien si la tâche est refusée, cet échec sera signalé au plus tôt par le contrôleur. C'est donc une prédictibilité temporelle qu'offre le système ; la prédictibilité logique, c-à-d le fait que le résultat fourni par la tâche soit bien correct, est automatiquement acquise (en supposant qu'une tâche qui s'exécute complètement retourne un résultat correct).

En revanche, dans le cas de systèmes temps réel avec des tâches d'optimisation, une tâche peut retourner plusieurs solutions acceptables pour différentes durées d'exécution. La prédictibilité temporelle garantit que la tâche retournera bien une solution acceptable à temps ou bien détecte un échec. La principale difficulté est reportée sur la prédictibilité logique : il faut garantir qu'une solution acceptable sera produite avant la fin de l'échéance.

Autant dire qu'obtenir à la fois la prédictibilité temporelle et logique de façon générique pour tous les problèmes d'optimisation nous semble impossible. En effet, cela nécessite de connaître une borne supérieure du temps d'exécution pour obtenir une solution acceptable quel que soit le problème. Rappelons que les profils de qualité ne conviennent pas car ils n'enregistrent que des informations probabilistes et n'offrent donc aucune garantie, d'un point de vue déterministe.

Respect des échéances

Si l'objectif est uniquement de produire une solution admissible et la meilleure possible dans le temps imparti, nous avons vu précédemment que les méthodes incomplètes sont un bon choix car elles peuvent adapter leur durée d'exécution. La difficulté reste de garantir la production d'au moins une solution admissible, c-à-d respectant toutes les contraintes dures.

Une tâche d'optimisation peut alors être décomposée en deux parties :

1. partie obligatoire : résolution par une méthode complète d'un CSP composé uniquement des contraintes dures du problème initial ; le temps de résolution peut être estimé soit par un apprentissage préalable, soit par une estimation dynamique (incluse dans le contrôleur) (par exemple, calcul de la complexité moyenne [Ver95] ou estimation par sondage [LL98]) ; en cas de sous-estimation, le non-respect de l'échéance est vu comme une panne du système et nécessite l'application d'une action réflexe d'urgence ; ces pannes sont *a priori* rares et interviennent plutôt lorsque le CSP est au niveau de la frontière entre cohérence et incohérence (zone du pic de complexité décrit dans la section 2.4.1) ;

2. partie optionnelle : résolution par une méthode incomplète de tout le problème ; éventuellement, utilisation de la solution de la partie obligatoire comme point de départ.

Cette décomposition reprend le concept de tâche imprécise décrit dans la section 1.3.2. Dans le cas d'un système temps réel composé de plusieurs tâches imprécises, nous avons présenté l'ordonnanceur *NORA* [SL92] (cf. section 1.3.2.2) qui assure une prédictibilité temporelle des parties obligatoires et cherche à maximiser le temps alloué aux parties optionnelles.

Seuil d'acceptation

Que signifie une solution acceptable ? Dans le cadre des VCSP, cela peut signifier une solution dont la valuation est suffisamment proche de l'optimum. En absolu, le **seuil d'acceptation** serait une distance maximale à l'optimum. En relatif, le seuil d'acceptation serait une qualité normée minimale. Dans les deux cas, la connaissance d'un majorant et d'un minorant de l'optimum est le seul moyen de garantir un seuil d'acceptation.

Dans une approche interruptible contrôlée par un utilisateur, la connaissance d'un encadrement de l'optimum permet de choisir si la résolution doit se poursuivre ou bien si elle a atteint un seuil d'acceptation.

3.2 Politiques d'ordonnancement pour un ensemble de problèmes d'optimisation

Nous étudions maintenant le cas plus complexe d'un système temps réel composé d'un ensemble de tâches d'optimisation devant se partager une unique ressource CPU.

Nous définissons d'abord ce système en précisant les objectifs visés, la façon de les évaluer et les résultats attendus.

Puis nous présentons notre **approche délibérative et interruptible** pour contrôler les résolutions. L'ordonnancement utilise des heuristiques construites en partie grâce à l'information d'encadrement progressif de l'optimum. Nous expérimentons cette approche sur un ensemble de problèmes SPOT5.

3.2.1 Système temps réel d'aide à la décision

3.2.1.1 Définitions et hypothèses simplificatrices

Définition 21 ———
Système temps réel d'aide à la décision *:*
système contrôlant un environnement extérieur à l'aide de tâches de prise de décision. L'évolution de l'environnement extérieur est perçu par des événements qui déclenchent/modifient/annulent des prises de décision et définissent/modifient des échéances temporelles sur la mise en oeuvre des décisions.

Remarque 15 L'aide à la décision signifie que le système propose des solutions via une interface à un utilisateur qui peut s'en servir ou les refuser. Par exemple, c'est le cas d'un copilote électronique qui, face à des menaces, propose des parades au pilote [ABCC+94a]. A l'inverse, un **système temps réel autonome** appliquera directement ses décisions.

◇

La prise de décision implique généralement l'exécution de trois phases successives :

- perception : acquisition des informations pour élaborer une situation observée à partir d'une situation réelle ;
- réflexion : élabore une ou plusieurs réponse(s) en résolvant un problème extrait de la situation observée ;
- mise en oeuvre de l'action : contrôle des actionneurs (cas des systèmes autonomes) ou interfaçage avec un utilisateur (cas des systèmes d'aide à la décision).

La conception d'un système temps réel d'aide à la décision bénéficie d'une séparation nette de la partie contrôle et de la partie calculatoire [HP85]. La partie contrôle traite les événements, dirige l'exécution des tâches par le biais d'un ordonnanceur et effectue les actions. L'ordonnanceur assure la répartition des ressources aux tâches et le respect des échéances temporelles. La partie calculatoire réalise les raisonnements (perception, analyse de situation, planification, ...) en temps limité. Il s'agit donc d'algorithmes contractuels ou interruptibles. Par exemple, c'est une méthode d'optimisation couplée avec une interface vers la partie contrôle. L'étude *Saturne* [ABCC+94b, ABdG95a, ABdG95b, AdG95, AF96, Bon97], menée au *CERT*, a proposé des solutions pour réduire la complexité de la partie contrôle par une décomposition en noyaux de contrôle écrits chacun dans un langage temps réel synchrone (Esterel [BG92], Lustre [CHPR91], Signal [LGGLBLM91]).

Par rapport à l'approche *Saturne* qui s'intéresse à modéliser la partie contrôle pour ensuite pouvoir la valider formellement, nous choisissons de simplifier cette partie en supprimant l'aspect traitement des événements pour se concentrer uniquement sur l'ordonnancement proprement dit. Nous supposons donc un ensemble d'hypothèses simplificatrices :

Hypothèse 1 ———

1. *la prise de décision est réduite à la résolution d'un problème d'optimisation sous contraintes, exprimé dans le formalisme des problèmes* VCSP **additifs**[6] *; la perception est donc immédiate et exacte, ainsi que l'extraction du problème ;*
2. *l'environnement extérieur est imprédictible ; activant ou invalidant des tâches de manière imprévisible ; l'échéance temporelle définie au déclenchement d'une tâche peut être modifiée durant son exécution ;*
3. *invariance des données d'un problème d'optimisation durant sa résolution ;*
4. *les tâches d'optimisation ne partagent aucune ressource, à l'exception du processeur ; les tâches sont indépendantes et sans contraintes de précédence ;*
5. *le système est exécuté sur un seul processeur ;*
6. *le système ne fait pas d'hypothèse sur le domaine d'application des problèmes à résoudre ;*
7. *une première solution acceptable est obtenue par une méthode gloutonne.*

Remarque 16 L'hypothèse 1.3 d'invariance des données du problème est assez restrictive. La prise en compte de la dynamicité des problèmes VCSP au sein d'une résolution serait une voie intéressante à explorer, par exemple, à partir des travaux [DD88, VS95, DV96, Dag97]. Notons que cette hypothèse est la même que celle qui est faite dans le cadre de la planification temps réel en environnement statique (cf. section 1.2.3.2). La différence avec ce cadre est que nous avons non pas un mais plusieurs problèmes à résoudre.

$\diamond$

Remarque 17 Le fait de pouvoir trouver facilement une première solution acceptable (cf. hypothèse 1.7) signifie qu'il n'y a pas de contraintes dures, ni de seuil d'acceptation.

$\diamond$

En résumé, le système doit résoudre un ensemble de problèmes Σ-VCSP ; chaque problème étant associé à une tâche. Le modèle de tâche est : apériodique, préemptible, interruptible, indépendant par rapport aux autres tâches, à échéance stricte ou révisable.

L'objectif consiste à maximiser l'utilité globale du système, avec respect des échéances temporelles. Nous allons voir dans la section suivante ce que signifie la notion d'utilité pour des problèmes d'optimisation.

[6] Le choix de la classe des Σ-VCSP est motivé par le fait que la majorité des problèmes d'optimisation sous contraintes utilisent un critère additif.

3.2.1.2 Comment mesurer l'efficacité d'un tel système ?

Notion d'utilité

Nous avons choisi le cadre des Σ-VCSP dans lequel sont exprimées des contraintes de préférence. Le coût associé à une contrainte indique son importance. Nous supposons que vis à vis de l'environnement extérieur (un autre système ou un utilisateur), l'intérêt d'une solution est uniquement jugée en fonction de la somme des coûts des contraintes satisfaites. Nous faisons donc une analogie monétaire entre des contraintes satisfaites et des gains financiers, c-à-d que le coût associé à une contrainte est assimilé à une somme d'argent, gagnée si la contrainte est satisfaite. Ainsi, nous pouvons comparer l'intérêt de deux solutions appartenant à deux problèmes différents uniquement par rapport à leurs valuations. Il s'agit de bien fixer les coûts associés aux contraintes pour privilégier tel ou tel problème vis à vis des autres.

Cela nous amène à définir les notions d'utilité intrinsèque et d'utilité globale pour un système restreint par les hypothèses 1 :

Définition 22 ———
Utilité intrinsèque d'une solution *:*
l'utilité intrinsèque d'une solution A d'un problème P est fonction de sa valuation $\varphi(A, P)$; elle est égale à $U_I(A, P)$, l'agrégation des valuations de toutes les contraintes satisfaites (identique à la qualité non normée de l'équation 3.1, section 3.1.2) :

$$U_I(A, P) = \sum_{c \in C_{sat(A,P)}} \varphi(c) = \omega - \varphi(A, P) \tag{3.4}$$

Ainsi l'utilité intrinsèque du résultat partiel fourni par une tâche d'optimisation à l'instant t est égale au bénéfice d'exploiter la meilleure solution produite pour le problème associé à la tâche.

Définition 23 ———
Utilité globale *d'un système :*
l'utilité globale des résultats produits par un système à un instant t est égale à $U_{glob}(t)$, la somme des utilités intrinsèques des meilleures solutions produites à t pour l'ensemble des N problèmes à résoudre :

$$U_{glob}(t) = \sum_{i=1}^{N} U_I(A_i, P_i) \tag{3.5}$$

Ainsi, l'utilité globale correspond au bénéfice d'exploiter l'ensemble des meilleures solutions produites à l'instant t. Il n'y a pas de notion de coût temporel, comme indiqué dans la section 1.1.3.4, car la contrainte opérationnelle prise en compte n'est pas un dépassement d'une échéance relative, mais un partage d'une ressource unique, le processeur. Ce partage est pris en compte par le fait de sommer les utilités intrinsèques.

Notion de dominance sur un intervalle temporel

Il est possible de comparer l'efficacité de différentes politiques d'ordonnancement à tout instant grâce à l'utilité globale.

Néanmoins, il se peut qu'une politique soit meilleure pendant un certain temps, et par la suite, qu'une autre politique devienne meilleure. C'est pourquoi nous définissons la notion de dominance d'une politique d'ordonnancement sur un intervalle temporel[7] :

Définition 24
Dominance d'une politique d'ordonnancement sur un intervalle temporel *:*
soit un système temps réel d'aide à la décision, réduit par les hypothèses 1 ; soit $\{PO_i, i \in [1, K]\}$*, un ensemble de* K *politiques d'ordonnancement pour ce système ; soit un certain environnement extérieur impliquant un ensemble de* N *problèmes* Σ-VCSP *à résoudre ; alors, une politique d'ordonnancement* PO_j *domine sur un intervalle temporel* $[t_{min}, t_{max}]$ *ssi :*

$$\forall t \in [t_{min}, t_{max}], \forall i \in [1, K], i \neq j, U_{PO_j}(t) \geq U_{PO_i}(t)$$

Dans cette définition de la dominance, un unique jeu de problèmes sert de test. Il serait aussi possible de comparer des politiques sur un ensemble de jeux de test différents ; cela donnerait par exemple, une dominance en moyenne d'une politique.

L'hypothèse 1.2 précise que les échéances sont révisables. Idéalement, une politique devrait dominer quelle que soit la date d'échéance. Néanmoins, nous pouvons supposer que la variation sur les échéances est bornée ; ce qui permet de déduire un intervalle temporel pour évaluer les politiques d'ordonnancement.

3.2.1.3 Résultats attendus

D'après les hypothèses 1, nous proposons une politique d'ordonnancement dans la section 3.2.2 qui procure les avantages suivants :

- dégradation progressive de l'utilité globale en cas de surcharge ;
- garantie d'optimalité en cas d'absence de contrainte temporelle ;
- aspect générique du système, indépendant des domaines d'application ;
- stratégie du meilleur effort d'abord pour faire face aux échéances révisables et utiliser au mieux le temps réellement disponible ;
- délibération sur l'utilité d'allouer le processeur à la résolution d'un problème plutôt que d'un autre ; cette délibération devant apporter une plus grande robustesse au système ;
- information sur la qualité normée globale en dynamique ; ce qui permet de garantir un seuil d'acceptabilité, hors contraintes temporelles ; cette qualité permet aussi une meilleure délibération lorsque les tâches sont presque terminées.

[7] Ainsi, nous adaptons le concept de dominance d'un algorithme interruptible, pressentie dans [Hor90] et définie dans [ABdG95a].

Remarque 18 Un minorant de la qualité normée globale est donné par la **qualité normée approchée globale** , notée ϱ_{glob}, qui est identique à la qualité normée approchée ϱ, mais au lieu de faire le calcul sur un seul problème, il est fait globalement sur l'ensemble des problèmes.

Soit A_i, la meilleure solution produite pour le problème P_i à l'instant t ; soit $lb(P_i)$, le meilleur minorant prouvé pour P_i ; soit ω_i, la somme des coûts des contraintes de P_i ; soit U_{max}, l'utilité globale optimale (cas où tous les optima des problèmes sont connus) :

$$\varrho_{glob}(t) = \frac{\sum_{i=1}^{N} \omega_i - \sum_{i=1}^{N} \varphi(A_i, P_i)}{\sum_{i=1}^{N} \omega_i - \sum_{i=1}^{N} lb(P_i)} \leq \frac{U_{glob}(t)}{U_{max}} \tag{3.6}$$

◇

Remarque 19 La prédictibilité logique et temporelle est assurée par l'hypothèse 1.7.

◇

3.2.2 Approche interruptible et délibérative

La stratégie du meilleur effort d'abord, pour faire face à un environnement imprédictible, implique l'utilisation d'algorithmes interruptibles. La connaissance dynamique de la qualité normée globale nécessite de calculer à l'exécution un majorant de l'utilité globale optimale U_{max}. Nous présentons dans la section 3.2.2.1, un algorithme interruptible, capable d'estimer sa qualité, pour résoudre un problème d'optimisation sous contraintes dans un système temps réel.

Ensuite, nous présentons dans la section 3.2.2.2, la politique d'ordonnancement retenue, qui est une approche délibérative fondée sur la notion d'utilité, estimée à l'aide de critères heuristiques.

Enfin, nous expérimentons cette approche sur un ensemble de problèmes SPOT5 dans la section 3.2.2.3.

3.2.2.1 Encadrement progressif de l'optimum

Chaque tâche interruptible, dans le système temps réel, a pour rôle de trouver de bonnes solutions à un problème et de prouver de bons minorants. Une tâche consiste donc à encadrer progressivement l'optimum du problème par un majorant et un minorant. Cet objectif existe déjà pour d'autres types de problèmes dans d'autres systèmes, comme par exemple, les problèmes d'optimisation linéaire en nombre entier [NW88] dans le système de résolution *CPLEX* [CPL95], ou bien les problèmes de réseaux de croyance [Pea86] dans le système délibératif *Protos* [Hor90].

La production de majorants et de minorants est réalisée par deux algorithmes interruptibles différents :

- production de majorants décroissants par un **enchaînement de méthodes incomplètes** ; nous réutilisons donc l'approche interruptible présentée dans la section 3.1.1.2 ;
- production de minorants croissants par un **enchaînement de méthodes complètes sur des simplifications** de moins en moins grossières, soit du problème, soit des objectifs de résolution ; nous reprenons donc l'approche *estimation de la qualité courante* de la section 3.1.2.2 ; techniquement, cela implique d'exhiber des méthodes de résolution plus simples que la résolution du problème initial par une méthode complète et qui fournissent

un minorant de l'optimum du problème initial ; il faut aussi déterminer un ordre de complexité croissant pour ces méthodes ; enfin, l'efficacité de l'enchaînement est amélioré par une réutilisation des résolutions successives ; tous ces aspects sont abordés dans la deuxième partie de la thèse (par exemple, cf. la section 4.3).

Remarque 20 Parmi les résultats attendus de la section 3.2.1.3, l'aspect générique du système, ainsi que l'obtention d'une garantie sur la qualité des solutions, hors contrainte temporelle, impose le choix d'un calcul des minorants à l'exécution. Les profils de qualité ne sont donc pas considérés.

◇

Remarque 21 Il y a des cas où les méthodes complètes sont plus efficaces que les méthodes incomplètes (cf. section 3.1.1.1). L'utilisation simultanée des deux méthodes est donc bénéfique.

De plus, pour garantir, hors contrainte temporelle, l'obtention de l'optimum et la preuve de son optimalité, la dernière méthode complète utilisée dans l'enchaînement est appliquée sur le problème cible sans simplification. La preuve d'optimalité permet de stopper les méthodes incomplètes et de terminer la tâche.

◇

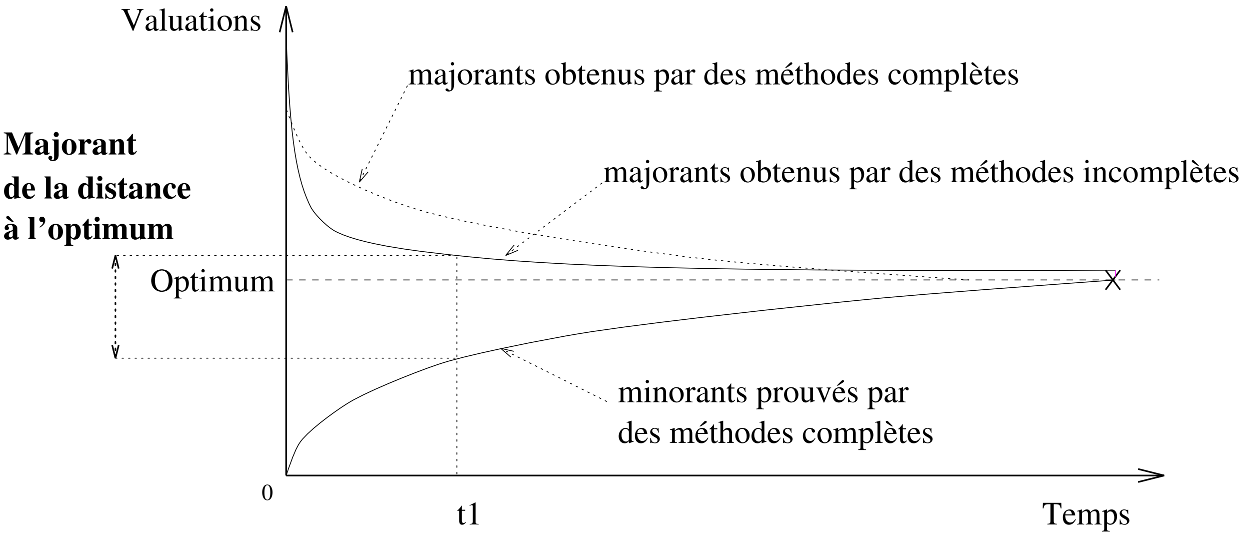


Fig. 3.6 - Encadrement progressif de l'optimum

La figure 3.6 résume notre approche, en montrant un encadrement progressif de l'optimum.

Les méthodes complètes et incomplètes s'intègrent assez bien au sein d'un algorithme interruptible. Parmi les propriétés attendues, listées dans [Zil96], d'un algorithme interruptible pour être employé dans un système temps réel, nous avons bien :

- la qualité croît de façon monotone ;
- généralement, la qualité croît plus au début de la résolution qu'à la fin ;
- il n'y a pas de problème pour obtenir la solution courante immédiatement (son stockage ne pose aucun problème, c'est juste l'espace mémoire d'une instanciation complète) ;

- il est facile de mettre un point d'interruption, fréquemment testé, dans les méthodes de résolution (délai d'interruption court) ;
- il n'y a pratiquement pas d'*overhead* de préemption (structures de données relativement réduites).

En revanche, la qualité d'une solution n'est pas forcément mesurable avec exactitude, même a posteriori : l'optimum peut être inaccessible en temps raisonnable. Il faut donc se contenter d'une qualité approchée (cf. équation 3.3, section 3.1.2). De même, la qualité courante n'est pas calculée en temps polynomial court : pour estimer la qualité courante, il faut résoudre des simplifications du problème cible par des méthodes complètes dont le temps risque de varier de façon exponentielle.

3.2.2.2 Critères heuristiques d'ordonnancement

Les critères que nous présentons ne tiennent pas compte des échéances temporelles associées aux tâches. Les échéances sont ignorées du fait de leur caractère révisable. Nous considérons donc un modèle de tâche très simplifié avec **une seule échéance temporelle globale révisable**, identique pour toutes les tâches. L'objectif de l'ordonnanceur est uniquement de partager le processeur aux tâches de façon à maximiser l'utilité globale.

Ordonnancement des tâches : maximisation de l'utilité

Une tâche consiste à encadrer progressivement l'optimum d'un problème. Cela se fait par l'enchaînement de résolutions par une méthode incomplète, pour les majorants, et par une méthode complète, pour les minorants. Lorsqu'une tâche est allouée au processeur, nous choisissons d'exécuter la prochaine méthode incomplète ou complète associée à la tâche jusqu'à sa terminaison. **Les résolutions s'exécutent donc séquentiellement**. C'est seulement à la fin d'une résolution que l'ordonnanceur reprend la main pour déterminer la prochaine tâche à exécuter.

Le choix de la prochaine tâche à exécuter porte sur celle qui offre une évolution espérée maximale de l'utilité globale en le minimum de temps. Ce compromis entre le gain escompté et le temps pour l'atteindre correspond à la notion d'utilité dans la théorie de la décision. Nous reprenons la définition de l'utilité donnée dans la section 1.1.3.4 :

$$U(t) = U_I(t) - CT(t) = f(Q_c(t)) - g(t)$$

Soit $\{O_i, i \in [1, N]\}$, une liste de N tâches actives ; soit T_i, la durée estimée de la prochaine résolution pour la tâche O_i ; soit P_i, le problème associé à la tâche O_i ; soit A_i et $lb(P_i)$ les meilleurs majorants et minorants de P_i produits à l'instant t ; enfin, soit A'_i et $lb'(P_i)$, les nouveaux majorants et minorants de P_i produits à $t + T_i$ si la tâche est exécutée ;
alors, l'utilité de sélectionner une tâche O_i s'écrit :

$$\begin{aligned} \Delta U(t, O_i) &= U(t + T_i, O_i) - U(t, O_i) & (3.7)\\ &= U_I(A'_i, P_i) - U_I(A_i, P_i) - \lambda \cdot T_i & (3.8) \end{aligned}$$

$U_I(A'_i, P_i) - U_I(A_i, P_i)$ représente le gain espéré pour l'utilité globale U_{glob} à l'instant $t + T_i$ par rapport à l'instant t. Ce gain est égal à la somme des coûts des contraintes nouvellement satisfaites. $\lambda \cdot T_i$ exprime le coût temporel. Le facteur λ permet de combiner les deux critères

: une valeur forte favorise les résolutions courtes, une valeur faible favorise les résolutions dont le gain est estimé le plus élevé.

Nous utilisons les heuristiques suivantes pour évaluer le gain $U_I(A'_i, P_i) - U_I(A_i, P_i)$ et le temps T_i :

- critère de gain maximum : distance à l'optimum approchée par $\varphi(A_i, P_i) - lb(P_i)$;
- critère de difficulté pour améliorer la solution courante : complémentaire de la qualité normée approchée, égal à $1 - \frac{\omega_i - \varphi(A_i,P_i)}{\omega_i - lb(P_i)}$, ou encore à $\frac{\varphi(A_i,P_i)-lb(P_i)}{\omega_i - lb(P_i)}$; nous faisons l'hypothèse que les chances de trouver des bonnes solutions diminuent en s'approchant de l'optimum ;
- critère d'estimation de la durée de la prochaine résolution : moyenne des durées des résolutions précédentes, égal à $\frac{\sum_{l=1}^{L_i} T_{il}}{L_i}$, avec $(T_{i1}, \ldots, T_{iL_i})$, la liste des durées des L_i résolutions précédentes effectuées sur le problème P_i.

Ainsi, nous obtenons le critère suivant indiquant l'intérêt de sélectionner une tâche O_i à l'instant t :

$$\Delta U(t, O_i) = \frac{\varphi(A_i, P_i) - lb(P_i)}{\omega_i - lb(P_i)} \cdot (\varphi(A_i, P_i) - lb(P_i)) - \lambda \cdot \frac{\sum_{l=1}^{L_i} T_{il}}{L_i} \tag{3.9}$$

A chaque fois que l'ordonnanceur reprend la main, il recalcule ce critère pour chaque tâche et choisit celle qui a la plus grande valeur du critère.

Une tâche est complètement terminée lorsque l'optimum est prouvé, c-à-d lorsque le majorant rejoint le minorant. Elle est alors retirée de la liste des tâches actives. Du fait de la connaissance d'un encadrement de l'optimum des problèmes, nous pouvons aussi retirer une tâche dont le seuil d'acceptation a été atteint. Ce seuil est donné en paramètre du système, sous la forme d'un pourcentage, noté ϵ_ϱ. Une tâche est retirée si sa qualité normée approchée dépasse ϵ_ϱ.

Remarque 22 Le critère pour estimer la durée de la prochaine résolution est assez simpliste. Nous avons vu dans la section 3.1.2.2 des pistes pour avoir une estimation plus précise. De plus, la moyenne fait un amalgame des temps de résolution pour améliorer le majorant et le minorant. Une séparation de ces deux objectifs améliorerait l'estimation mais nécessiterait de déterminer à l'avance, pour chaque problème, si la prochaine résolution sera pour améliorer le majorant ou pour améliorer le minorant. Néanmoins, la moyenne des résolutions précédentes reste une information pertinente pour différencier les problèmes entre eux. C'est le rôle du coefficient λ d'augmenter cette moyenne, puisque la prochaine résolution sera a priori plus lente car la complexité de la méthode de résolution est choisie croissante.

◇

Sélection de la méthode de résolution : répartir l'effort entre majorant et minorant

Lorsqu'une tâche est sélectionnée par l'ordonnanceur, nous pouvons alors tenter d'améliorer, soit le majorant du problème associé à la tâche, en exécutant une méthode incomplète, soit le minorant, en exécutant une méthode complète. Le choix entre une méthode complète ou incomplète se fait à l'aide de plusieurs heuristiques :

- progression alternée des deux bornes, de manière à réduire progressivement l'encadrement de l'optimum ; cela revient à équilibrer le nombre de méthodes complètes par rapport au nombre de méthodes incomplètes déjà exécutées sur un problème ;
- amélioration principalement de l'utilité intrinsèque de la meilleure solution connue du problème ; c'est donc une heuristique qui s'oppose à la précédente ; le compromis est donné sous la forme d'un pourcentage, noté β, indiquant le rapport minimum entre le temps consommé par les méthodes incomplètes et celui consommé par les méthodes complètes ; de même, un second rapport, noté β_{glob}, indique le rapport minimum entre le temps consommé par toutes les méthodes incomplètes sur tous les problèmes et celui consommé par toutes les méthodes complètes sur tous les problèmes ; β_{glob} permet de contrôler le compromis de temps entre méthodes complètes ou incomplètes de façon plus globale que β ; il permet de retarder les méthodes complètes au début de l'exécution du système temps réel ;
- lors de la première résolution sur le problème, choix d'une méthode incomplète, sauf si le gain estimé, égal à ω_i[8], est largement supérieur au gain médian sur l'ensemble des problèmes à résoudre (supérieur d'un facteur Γ) ; dans ce cas une méthode complète est préférée ; cette exception vise à vérifier si un problème qui aurait beaucoup de contraintes de coûts élevés peut effectivement produire un gain important pour U_{glob}.

Soit L_i^{mc} (resp. L_i^{mi}), le nombre de méthodes complètes (resp. incomplètes) déjà exécutées sur le problème P_i et $(T_{i1}^{mc}, \ldots, T_{iL_i^{mc}}^{mc})$ (resp. $(T_{i1}^{mi}, \ldots, T_{iL_i^{mi}}^{mi})$), la liste des durées d'exécution de ces méthodes. En supposant que les tâches ont étés triées dans l'ordre d'utilité ΔU décroissante, O_1 étant la tâche sélectionnée, nous exprimons le prédicat suivant indiquant si une méthode complète est choisie pour résoudre le problème P_1 :

$$\begin{aligned} LbOK = \ & \left(\sum_{l=1}^{L_1^{mi}} T_{1l}^{mi} \geq \beta \cdot (\sum_{l=1}^{L_1^{mc}} T_{1l}^{mc} + \sum_{l=1}^{L_1^{mi}} T_{1l}^{mi})\right) \ \wedge \\ & \left(\sum_{i=1}^{N}\sum_{l=1}^{L_i^{mi}} T_{il}^{mi} \geq \beta_{glob} \cdot (\sum_{i=1}^{N}\sum_{l=1}^{L_i^{mc}} T_{il}^{mc} + \sum_{i=1}^{N}\sum_{l=1}^{L_i^{mi}} T_{il}^{mi})\right) \ \wedge \\ & \Big((L_1^{mc} < L_1^{mi}) \ \vee \ ((L_1^{mc} = 0) \wedge (\varphi(A_1, P_1) > \Gamma \cdot (\varphi(A_{N/2}, P_{N/2}) - lb(P_{N/2}))))\Big) \end{aligned} \tag{3.10}$$

Bilan

Les heuristiques d'ordonnancement que nous avons présentées sont relativement simples et a priori intuitives. Par exemple, elles n'utilisent que des termes linéaires ou quadratiques. D'autres heuristiques pourraient être établies avec des termes exponentiels, reflétant la décroissance exponentielle de la valuation lors d'une résolution incomplète ou bien la croissance exponentielle du temps de résolution pris par les méthodes complètes successives. De même, le paramétrage de ces heuristiques est une affaire délicate. Les paramètres pourraient être adaptés dynamiquement par le système. C'est pourquoi, face à toutes ces solutions potentielles, nous avons défini la notion de dominance (définition 24) qui permet justement de départager différentes heuristiques.

Par ailleurs, nous avons fait le choix de ne pas interrompre les résolutions ; l'ordonnanceur n'ayant pour seul objectif que d'enchaîner au mieux les résolutions. Cette approche n'est valable que si les résolutions sont suffisamment courtes par rapport à l'échéance temporelle globale.

[8] Au départ, une problème P_i a un encadrement initialisé à ω_i : $\varphi(\emptyset, P_i) = \omega_i$ et $lb(P_i) = 0$.

Une autre approche serait d'allouer une tâche au processeur avec une limite de temps (par exemple, égale à la durée estimée de la prochaine résolution). Dans le cas où cette limite est dépassée, la tâche, et donc la résolution, est suspendue. L'ordonnanceur reprend alors son contrôle ; une autre tâche peut être ensuite exécutée. Dans cette approche, il serait également intéressant de prendre en compte le temps pris par l'ordonnanceur, à l'instar de l'approche décrite en section 1.3.4.

Enfin, une extension importante du système serait de prendre en compte différentes échéances temporelles pour chacune des tâches. Cela implique un ordonnancement des tâches au processeur sur un intervalle temporel allant de l'instant courant jusqu'à l'échéance la plus éloignée, à l'opposé de l'ordonnancement instantané que nous venons de présenter. L'inconvénient est l'absence de prévision à long terme fiable. A l'instar de l'ordonnanceur *Multi-Great* (cf. section 1.2.2.2) et de son extension [DMG97], nous pourrions effectuer une réservation temporelle du processeur de manière à garantir un minimum de temps aux tâches. Puis la réservation temporelle serait augmentée pour les tâches dont l'échéance est proche et qui présentent une forte amélioration potentielle de l'utilité globale (utilisation du critère ΔU).

3.2.2.3 Evaluation sur un ensemble de problèmes SPOT5

Nous expérimentons les heuristiques d'ordonnancement que nous venons de présenter sur une sélection de 10 problèmes SPOT5, pris parmi les plus durs à résoudre. Le système d'optimisation sous contraintes en temps limité reçoit ces 10 problèmes à l'initialisation. L'échéance temporelle est limité à une heure maximum. Le système doit maximiser l'utilité globale, de manière interruptible. Le système fournit également la qualité globale couranteϱ_{glob}, grâce à un encadrement progressif des optima des 10 problèmes. Notons que ce système ne répond pas directement à la formulation initiale des contraintes temporelles dans SPOT5. En effet, initialement, c'est une programmation journalière d'un seul problème avec une contrainte de temps de 1 heure qui était prévue. Néanmoins, sous ces conditions, le problème du partage du processeur peut également être important. En effet, chaque problème peut être décomposé en composantes connexes, ce qui introduirait un ensemble de sous-problèmes à résoudre en temps borné.

Nous avons développé une application pour simuler le système, à partir de notre librairie *C++* de stratégies de résolution, composée des méthodes de simplification et de résolution présentées dans la thèse. La simulation consiste à enregistrer à chaque itération de l'ordonnanceur l'utilité globale du système, le temps simulé, c-à-d la somme des temps de toutes les résolutions effectuées, et, pour chaque tâche O_i, $\varphi(A_i, P_i)$, son majorant, $lb(P_i)$, son minorant, $\sum_{l=1}^{L_i^{mi}} T_{il}^{mi}$, son temps de résolution par méthodes incomplètes, $\sum_{l=1}^{L_i^{mc}} T_{il}^{mc}$, son temps de résolution par méthodes complètes. Les paramètres de configuration, ainsi que leur valeur, de l'application sont :

- une liste de problèmes Σ-VCSP à optimiser : les instances SPOT5 numérotées 28, 5, 408, 412, 11, 507, 509, 42, 503 et 505 (cf. annexe A.2) ;

SPOT5	**nbvar**	**nbcontr**	ω_i	$\varphi(P_i)$	$U_I(P_i) = \omega_i - \varphi(P_i)$
28	230	5226	326158	270105	56053
5	309	5621	376	261	115
408	200	2232	9310	6228	3082
412	300	4348	48483	32381	16102
11	364	10108	60598	38478	22120
507	311	5732	42527	27390	15137
509	348	8624	55571	36446	19125
42	190	1394	263117	155050	108067
503	143	635	20209	11113	9096
505	240	2242	34353	21253	13100
Total =	2635	46162	860702	598705	261997

- une liste de méthodes complètes appliquées sur des simplifications de moins en moins grossières d'un problème :
  - chaque problème est simplifié en appliquant l'algorithme glouton, présenté en section 5.2.2.2, pour décomposer les variables en grappes ; les grappes ont une taille successivement de 50, 70 puis 100 variables (paramètre Tmax = 50, 70, 100, cf. expérimentation en section 5.2.2.3) ;
  - l'algorithme *Russian Doll Search* adapté aux contraintes unaires[9] est employé pour résoudre les problèmes simplifiés ; l'ordre chronologique initial des variables est respecté.
- une liste de méthodes incomplètes appliquées sur les problèmes cibles ; un algorithme de recuit simulé est employé avec des paramètres de configuration croissants :
  - $l = n \times d$, la taille d'un palier, égale aux nombre de valeurs, $g = 0$, la descente en gradient nulle, $m = 0.9$, le coefficient de décroissance de la température ;
  - $l = n \times d$, $g = 0$, $m = 0.95$;
  - $l = n \times d$, $g = 0$, $m = 0.98$;
  - $l = 2 \times n \times d$, $g = 0$, $m = 0.97$.
- des paramètres de configuration des heuristiques de l'ordonnanceur :
  - le temps de simulation limité à 1200 secondes ;
  - $\lambda = 500$, le coefficient pour combiner coûts et temps dans le calcul de l'utilité ΔU ;
  - $\beta = 92\%$, $\beta_{glob} = 40\%$, les coefficients pour privilégier le calcul des majorants par rapport à celui des minorants ;
  - $\epsilon_{\varrho} = 99\%$, le seuil d'acceptation (qualité normée approchée supérieure à 99%) permettant d'arrêter une tâche ;
  - $\Gamma = 5$, le coefficient pour comparer les encadrements de manière à forcer le calcul d'un minorant si l'écart est important (cf. dernière heuristique de la section 3.2.2.2).

[9]La valeur zéro affectée à une variable signifie que la prise de vue associée n'est pas faite. Une contrainte unaire valuée indique le coût de ne pas effectuer cette photographie. Ces contraintes sont prises en compte par le *forward checking*, lors du calcul du minorant d'une instanciation partielle. Il ne faut donc ajouter à ce minorant que la contribution lb'_{rds}, uniquement du sous-problème où le calcule du *forward checking* est nul.

L'ordonnanceur est décrit par le pseudo-code suivant :

```
ORDONNANCEUR( file.cop, file.cmd, file.imd, TempsLimite, λ, β, β_glob, ε_ϱ, Γ )
  chargement des N problemes P_i a partir du fichier file.cop
  creation des N taches O_i associees aux problemes P_i avec
          A_i = ∅, φ(∅, P_i) = ω_i, lb(P_i) = 0, L_i^mc = 0, L_i^mi = 0 et ΔU(0, O_i) = ω_i
  insertion des N taches dans TachesActives, une liste triee des taches actives

  chargement des L_max^mc methodes completes a partir du fichier file.cmd
  chargement des L_max^mi methodes incompletes a partir du fichier file.imd

  tant que (TachesActives ≠ ∅ ∧ (Σ_{i=1}^N Σ_{l=1}^{L_i^mc} T_il^mc + Σ_{i=1}^N Σ_{l=1}^{L_i^mi} T_il^mi) < TempsLimite) faire
     trie de la liste TachesActives par ΔU(t, O_i) decroissants (renomage des indices i)
     si (LbOK = vrai ∧ L_1^mc < L_max^mc) alors
       L_1^mc := L_1^mc + 1
       execution de la methode complete L_1^mc sur une simplification de P_1
       T_{1L_1^mc}^mc := T_{1L_1^mc}^mc + TempsResolution^10
     sinon
       L_1^mi := L_1^mi + 1
       si (L_1^mi > L_max^mi) alors
         execution de la methode incomplete L_max^mi sur P_1
       sinon
         execution de la methode incomplete L_1^mi sur P_1
       fin si
       T_{1L_1^mi}^mi := T_{1L_1^mi}^mi + TempsResolution
     fin si
     recalcule de Δ(t, O_1)
     si (ϱ(A_1, P_1) ≥ ε_ϱ) alors
       TachesActives := TachesActives − O_1
     fin si
  fin tant que
```

Remarque 23 Nous avons observé expérimentalement une amélioration de l'ordonnanceur si le calcul des $\Delta U(t, O_i)$ prend en compte le pourcentage de temps CPU consommé par la tâche O_i de manière à adapter dynamiquement le paramètre λ qui combine valuation et temps. Ainsi, nous remplaçons dans l'équation 3.9, λ par λ' : $\lambda' = \lambda \cdot (\sum_{l=1}^{L_i^{mc}} T_{il}^{mc} + \sum_{l=1}^{L_i^{mi}} T_{il}^{mi}) / (\sum_{j=1}^{N} \sum_{l=1}^{L_j^{mc}} T_{jl}^{mc} + \sum_{j=1}^{N} \sum_{l=1}^{L_j^{mi}} T_{jl}^{mi})$.

◇

La figure 3.7 montre l'évolution de l'encadrement global ($\sum_{i=1}^{N} \varphi(A_i, P_i)$, somme des majorants et $\sum_{i=1}^{N} lb(P_i)$, somme des minorants) de l'optimum des 10 problèmes ($\sum_{i=1}^{N} \varphi(P_i) = 598705$) au cours du temps. La simulation a été effectuée sur une station *SUN UltraSparc*. Pour $t \in [0, 52[$, l'ordonnanceur choisit de calculer un minorant du problème 28 avant d'ordonnancer le calcul d'un majorant du problème 42, puis un minorant du problème 42 et enfin un majorant du problème 28. Ensuite, pour $t \in [52, 222[$, l'ordonnanceur choisit de calculer des majorants pour les

[10] *TempsResolution* contient le temps en secondes de la dernière résolution effectuée sur le processeur.

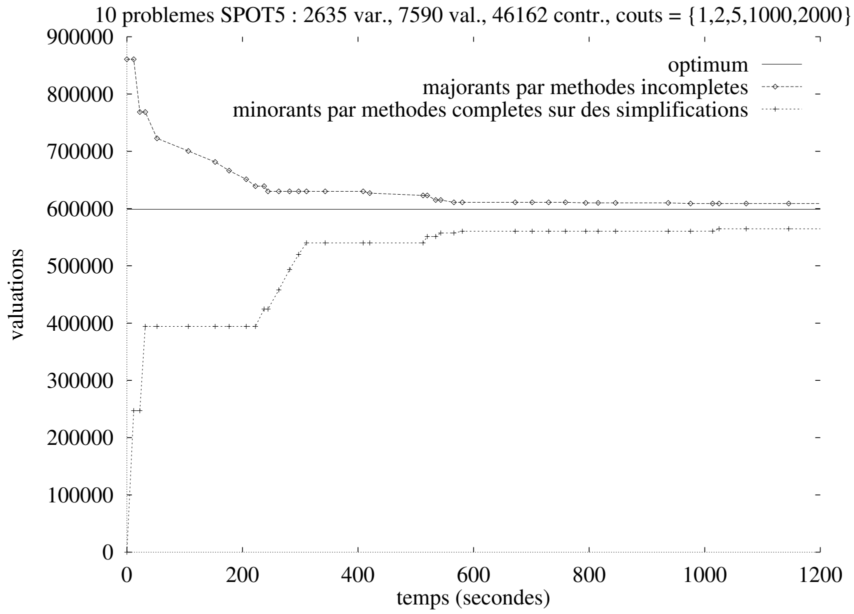


Fig. 3.7 - Encadrement progressif de l'optimum pour 10 problèmes SPOT5

problèmes 11, 509, 412, 507 et 505. Pendant cette période, la somme des minorants n'augmente pas. Puis, pour $t \in [222, 343[$, l'ordonnanceur choisit principalement de calculer des minorants pour les même problèmes 412, 509, 11, 507 et 505. A $t = 520$, la qualité normée approchée du problème 503 est égale à 99.85%, la tâche associée à ce problème est retirée de la liste des tâches actives. De même, à $t = 543$, la qualité normée approchée du problème 408 est égale à 99.6%, sa tâche associée est également retirée de $TachesActives$. Par ailleurs, la tâche associée au problème 5, qui a le plus faible ω_i, n'est allouée au processeur que assez tard, à $t = 730$.

La figure 3.8 montre l'évolution de la qualité normée approchée globale ϱ_{glob} en fonction du temps. Au bout d'environ 550 secondes, la qualité normée exacte pour l'ensemble des problèmes est garantie d'être supérieure à 80%. C'est à dire qu'au plus, le système doit améliorer la valuation globale de $\sum_{i=1}^{N} \varphi(A_i, P_i) - \sum_{i=1}^{N} lb(P_i) = 614950 - 557393 = 57557$ pour atteindre l'utilité globale optimale, U_{max} .

Dominance d'une politique d'ordonnancement

Nous avons comparé la politique d'ordonnancement utilisant un encadrement progressif de l'optimum, notée PO_1, avec les mêmes paramètres que précédemment, par rapport à d'autres politiques d'ordonnancement sans estimation de la qualité des solutions courantes, notées PO_2 et PO_3. Les exécutions ont toutes été faites sur une station *SUN UltrSparc*. Pour avoir un comportement similaire des méthodes incomplètes, d'une exécution d'une politique à l'autre, nous fixons avant chaque résolution la valeur initiale du générateur de nombres pseudo-aléatoires en fonction du paramètre L_1^{mi}. La figure 3.9 montre l'évolution de l'utilité globale en fonction du temps pour deux politiques d'ordonnancement, PO_1 et PO_2. La politique PO_2 ne calcule jamais de minorant (paramètres $\lambda = 10000$, $\beta = 110\%$, $\beta_{glob} = 110\%$). PO_2 est dominé par PO_1 dans l'intervalle temporel $[140, 1000]$. En fait, PO_2 alloue trop de temps CPU à la résolution des deux problèmes 28 et 42 qui ont un optimum beaucoup plus élevé que les autres problèmes.

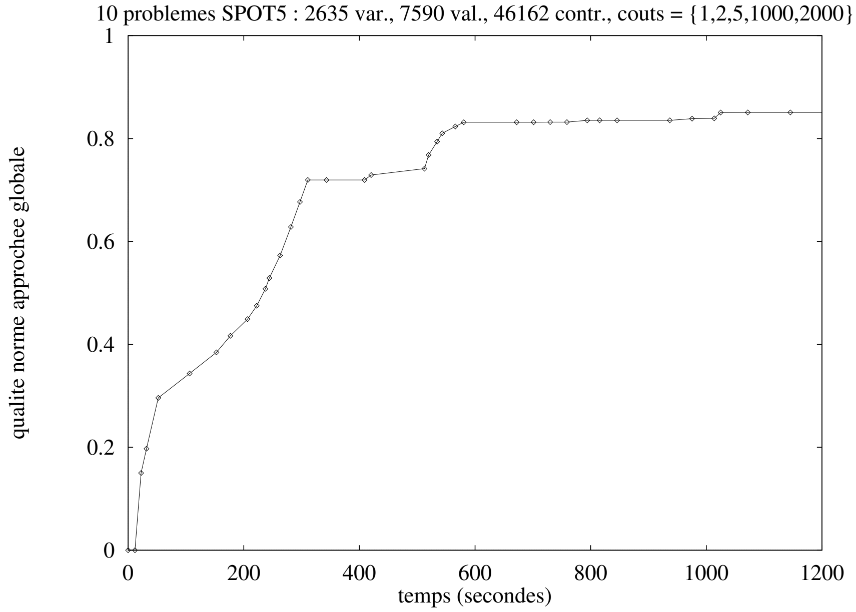


Fig. 3.8 - Evolution de la qualité normée approchée globale ϱ_{glob} pour 10 problèmes SPOT5

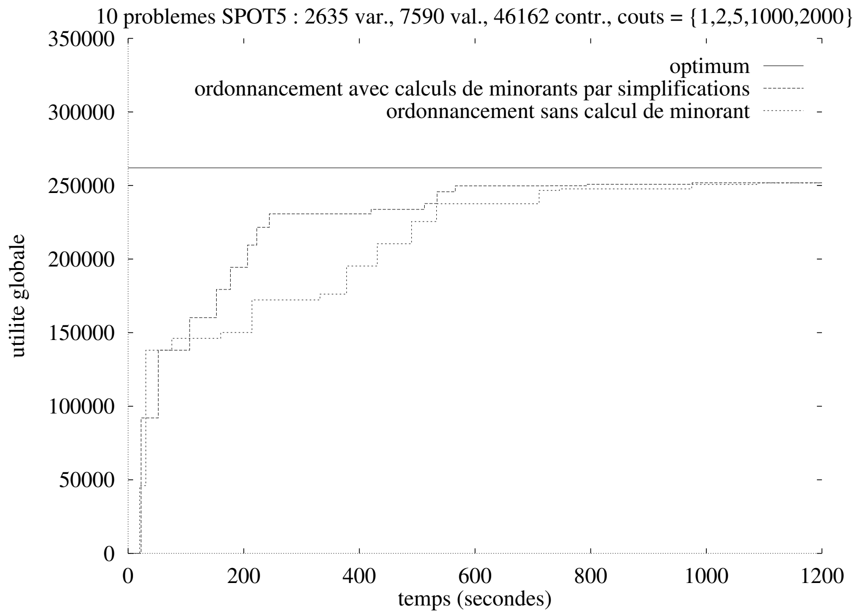


Fig. 3.9 - Evolution de l'utilité globale U_{glob} pour deux politiques d'ordonnancement, PO_1 et PO_2 ; notion de dominance

Pour améliorer le comportement de PO_2, nous modifions légèrement le calcul des $\Delta U(t, O_i)$. Le critère d'estimation de la durée de la prochaine résolution devient un critère de priorité pour effectuer un ordonnancement, noté PO_3 (paramètres $\lambda = 10000$, $\beta = 110\%$, $\beta_{glob} = 110\%$), proche d'une méthode dite "du tourniquet" (*round robin*) [Kra87] :

$$\frac{\sum_{l=1}^{L_i^{mc}} T_{il} + \sum_{l=1}^{L_i^{mi}} T_{il}}{\sum_{i=1}^{N} \sum_{l=1}^{L_i^{mc}} T_{il} + \sum_{i=1}^{N} \sum_{l=1}^{L_i^{mi}} T_{il}} \times (\sum_{l=1}^{L_i^{mc}} T_{il} + \sum_{l=1}^{L_i^{mi}} T_{il})$$

Le premier terme indique le pourcentage de CPU consommé par la tache O_i. Le second terme indique le temps total déjà consommé par la tâche. Ainsi, la tâche qui a consommé le moins de temps CPU et qui a un majorant élevé prendra le processeur. La figure 3.10 compare les politiques PO_1 et PO_3. Maintenant, les deux politiques sont assez proches. PO_1 continue à dominer PO_3 sur l'intervalle $[200, 330]$ mais se fait dominer sur l'intervalle $[330, 550]$.

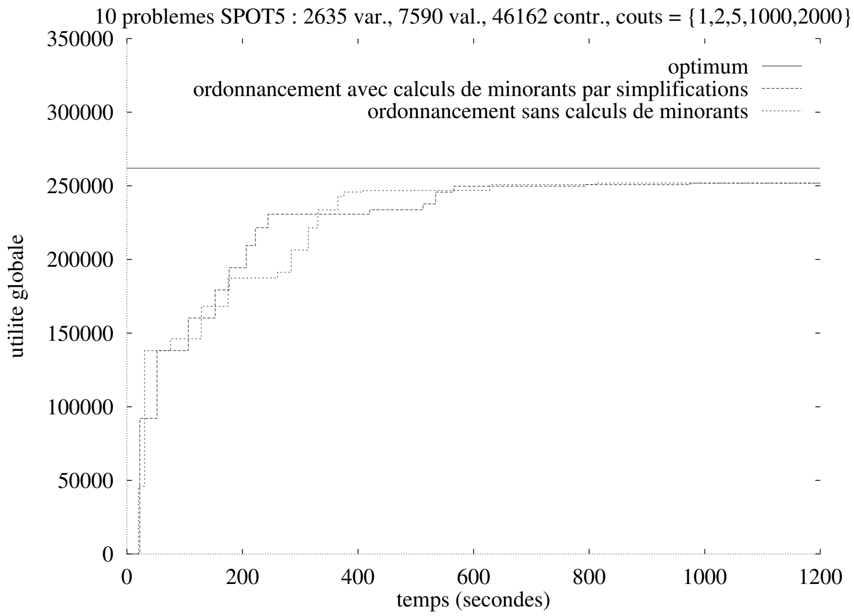


Fig. 3.10 - Evolution de l'utilité globale U_{glob} pour deux politiques d'ordonnancement, PO_1 et PO_3 ; notion de dominance

Nous avons testé une quatrième politique, notée PO_4 (paramètres $\lambda = 5000$, $\beta = 110\%$, $\beta_{glob} = 110\%$), qui n'effectue pas de calcul de minorant par résolution de simplifications mais essaie de prédire l'évolution des majorants. C'est à dire qu'un faux minorant est construit de manière heuristique par la formule suivante :

$$\textit{pseudo-lb}(P_i) = \frac{\varphi(A_i, P_i)}{1 + \frac{1}{L_i^{mi}}}$$

L'heuristique suppose au départ que l'optimum se trouve à peu près vers $\omega_i/2$ puis raffine cette hypothèse en rapprochant l'optimum du majorant courant $\varphi(A_i, P_i)$ plus le nombre de méthodes incomplètes déjà exécutées sur le problème P_i est grand. La figure 3.11 compare les politiques PO_1 et PO_4. La politique PO_4 domine PO_1 sur l'intervalle $[300, 550]$.

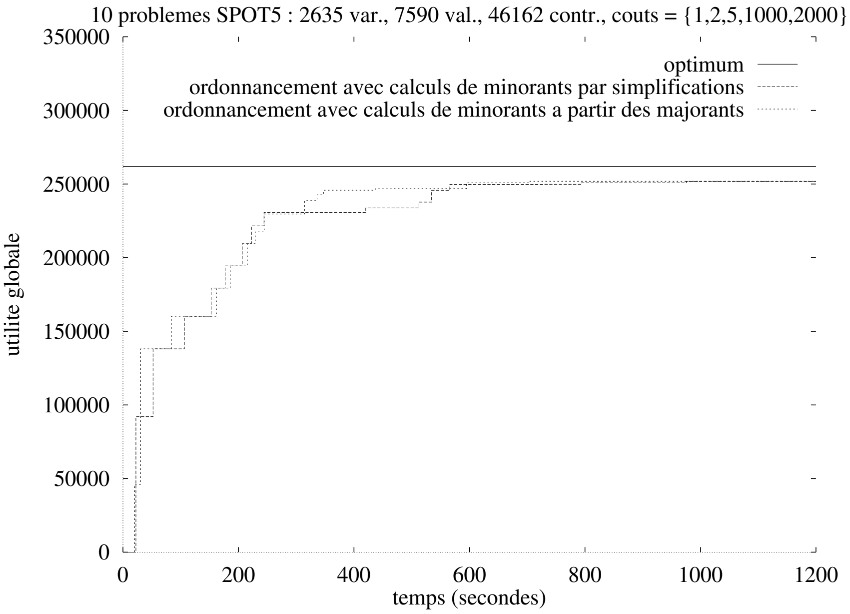


Fig. 3.11 - Evolution de l'utilité globale U_{glob} pour deux politiques d'ordonnancement, PO_1 et PO_4 ; notion de dominance

Entrelacement ou séquentialité ?

Nous avons fait le choix de séquentialiser les résolutions , plutôt que de les entrelacer. Ce choix permet d'obtenir une croissance plus rapide de l'utilité globale au début de l'exécution du système. En effet, l'algorithme du recuit simulé améliore brusquement la qualité des majorants. Un entrelacement aurait tendance à freiner cette progression dans le cas où un problème, parmi ceux entrelacés, a un optimum nettement plus élevé que les autres. La figure 3.12 montre l'évolution de l'utilité globale pour un système réduit à deux problèmes SPOT5, les instances 28 et 42. Une seule méthode incomplète est exécutée soit sur 28 et 42 en entrelacement, soit sur 28 puis sur 42, soit sur 42 puis 28. Sur cet exemple, le meilleur ordonnancement consiste à enchaîner la résolution du problème 42 puis celle du problème 28.

Conclusion

Les expérimentations menées sur un système optimisant un ensemble de problèmes SPOT5 nous ont permis de valider l'approche interruptible et délibérative fondée sur encadrement progressif de l'optimum. Ces expérimentations, ainsi qu'une comparaison avec d'autres méthodes d'ordonnancement plus simples fait ressortir plusieurs points :

- l'enchaînement de méthodes de complexité croissante permet d'avoir une évolution plus rapide de la qualité globale au début de l'exécution du système ; cela devrait donc permettre au système de réagir de manière progressive en cas de surcharge (tant que $t > 50$ secondes) ou d'échéances temporelles imprévues ;
- le calcul des minorant permet effectivement de mieux ordonner la résolution des problèmes en choisissant bien les plus importants en premier (par exemple, choix en premier du problème 42) et en détectant ceux pour qui l'optimum est quasiment atteint ;

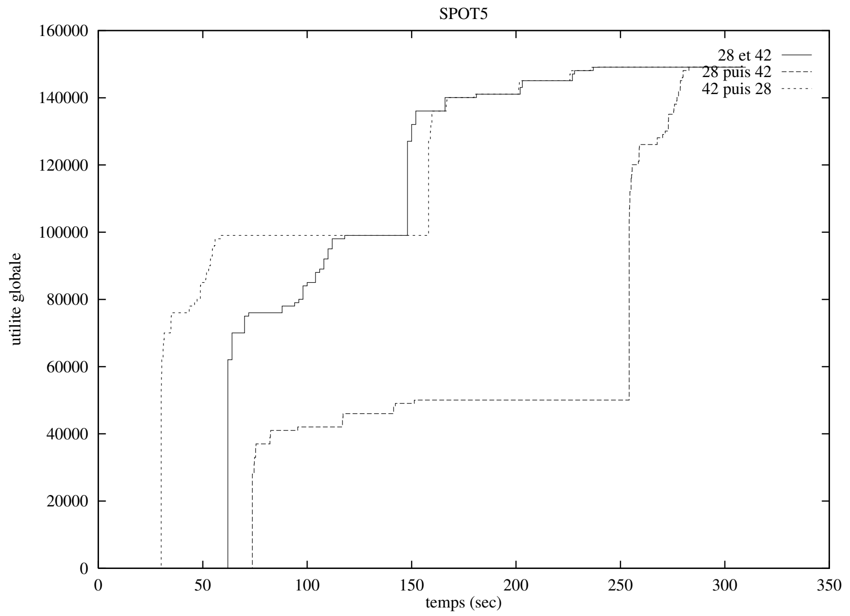


Fig. 3.12 - Evolution de l'utilité globale pour deux problèmes SPOT5 ; résolution séquentielle ou entrelacée par un algorithme de recuit simulé

- le calcul des minorants est coûteux en temps CPU, c'est pourquoi des politiques d'ordonnancement plus simples donnent aussi de bons résultats ; il est donc crucial de pouvoir produire efficacement de bons minorants ;
- sur les problèmes SPOT5, la variance sur la qualité des majorants fournis par un algorithme de recuit simulé ne s'avère pas assez forte pour mettre en valeur une plus grande robustesse d'une approche délibérative ; il serait intéressant de procéder à d'autres expérimentations sur des problèmes différents, pour mesurer la fiabilité d'un ordonnanceur avec calcul des minorants, par rapport aux autres approches ;
- le calcul des minorants offre l'avantage de produire dynamiquement une information sur la qualité globale atteinte par le système à tout instant ; dans le cas d'un système d'aide à la décision, c'est une information précieuse pour l'utilisateur ; cela permet de garantir des seuils d'acceptation, hors contraintes temporelles, bien avant que des méthodes complètes appliquées sur les problèmes cibles terminent.

Par ailleurs, le choix d'une modélisation des problèmes dans le formalisme VCSP offre un grand pouvoir d'expression et permet d'obtenir un système générique, indépendant d'un domaine d'application particulier. En revanche, cela impose des méthodes de résolution dites "faibles" [AS94] car elles effectuent une recherche aveugle et ont des performances qui peuvent varier fortement d'un problème à l'autre. De plus, le cadre VCSP impose une connaissance statique et précise d'un problème pour pouvoir le résoudre. Des extensions à ce formalisme, telles que les VCSP dynamiques [VS95, DV96] ou les VCSP probabilistes [Sch94, SFV97], permettraient d'étendre la capacité d'adaptation des systèmes temps réel intelligents.

Une approche délibérative fondée sur un encadrement de l'optimum n'est efficace que si nous avons les moyens de calculer suffisamment rapidement de bons minorants. Nous avons donc étudié plusieurs approches pour obtenir des minorants, soit en simplifiant les problèmes, soit un simplifiant les objectifs des méthodes complètes. C'est l'objet de la seconde partie de la thèse.

Partie II

Calcul de minorants à partir de simplifications génériques

Introduction

Cette seconde partie s'intéresse à déterminer dynamiquement la qualité normée d'une solution. Nous avons défini, en optimisation, la qualité normée d'une solution à partir de la distance entre la valuation de la solution courante et la valuation optimale. Puisque l'optimum est inconnu à l'exécution, nous essayons de l'approcher par un minorant et un majorant. A tout instant, l'encadrement de la valuation optimale par un minorant et un majorant définit un minorant de la qualité normée de la solution courante.

Les méthodes classiques de génération de solutions ne fournissent à tout instant que des majorants de l'optimum et seules les méthodes complètes terminent et prouvent l'optimum, la qualité de la solution est alors maximale. Notre principal objectif revient donc à calculer des minorants de l'optimum plus rapidement qu'en résolvant tout le problème avec une méthode complète. L'autre objectif important est que nos techniques soient génériques, indépendantes des domaines d'application. La seule restriction porte sur l'opérateur d'agrégation, choisi additif. En effet, les problèmes d'optimisation additive sous contraintes sont les plus courants. L'objectif de généricité vient du fait que le formalisme VCSP se veut minimal, donc pouvant accepter une large classe de problèmes. En contrepartie, l'effort est surtout mis sur les algorithmes de résolution et, dans notre cas, sur les techniques capables de produire des minorants. Nous avons établi une classification des techniques potentielles pour produire un minorant (voir figure 3.13). Principalement, nous avons étudié deux classes complémentaires de simplification.

La première classe de simplifications porte sur la structure du problème. Principalement, trois types de simplifications sont analysées : le changement de la structure de valuation, la suppression de certaines contraintes et l'abstraction sur les domaines des variables. Pour la plupart de ces techniques, l'objectif revient à réduire la taille du problème et par conséquent sa complexité, quel que soit l'algorithme complet utilisé pour la résolution. Une technique un peu à part, car elle nécessite une conversion du modèle Σ-VCSP dans le modèle *PL01*, consiste à relaxer la contrainte d'intégrité pour passer de domaines discrets à des domaines continus.

La suppression des contraintes peut être différemment orientée. Par exemple, il s'agit de séparer les variables en grappes distinctes, ou de décomposer les contraintes en sous-ensembles faciles à résoudre (arbres couvrants, ensemble *coupe-cycle*). Ou bien, la suppression consiste à se focaliser sur les contraintes les plus importantes. Ou encore, pour limiter les retours-arrières, nous pouvons supprimer des contraintes de manière à minimiser la largeur de bande du graphe des contraintes.

L'abstraction sur les domaines revient à rajouter des n-uplets permis dans les contraintes (en extension), pour utiliser ensuite la notion de substituabilité entre valeurs, de façon à regrouper en une seule les valeurs qui ont un comportement similaire vis à vis des contraintes. Cette simplification consiste donc à relaxer des contraintes de manière plus fine que ne le fait la sup-

- Simplifications structurelles :
  - relaxation des contraintes (cf. chapitre 5) :
    * suppression de contraintes ;
    * ajout de n-uplets autorisés à l'intérieur des contraintes ;
  - relaxation de la contrainte d'intégrité sur les domaines (cf. chapitre 6) :
    * transformation d'un problème Σ-VCSP en un problème *PL01*
    * utilisation des techniques de résolution de la *Programmation Linéaire*
  - changement de la structure de valuation (cf. chapitre 4) :
    * passage d'une structure additive à une structure lexicographique, ou possibiliste, ou classique ;
- Simplifications de l'objectif (cf. chapitre 6)
  - relaxations localement cohérentes :
    * utilisation de méthodes de renforcement de cohérence étendues à l'optimisation ;
    * relaxation arc-cohérente optimale en *PL01* ;
  - méthodes complètes adaptées au temps limité :
    * utilisation de l'algorithme *Best First Branch and Bound* ;
    * extension de l'algorithme *Depth First Branch and Bound* ;
    * extension de l'algorithme *Russian Doll Search.*

Fig. 3.13 - Vue d'ensemble des techniques potentielles pour produire un minorant

pression des contraintes. Elle peut aussi avoir pour objectif de faire apparaître des symétries ou de faciliter la transformation en contraintes linéaires.

Le changement de la structure de valuation est envisageable grâce au fait d'avoir choisi le formalisme des VCSP qui englobe plusieurs paradigmes d'optimisation. Cette simplification novatrice est étudiée de manière théorique et pratique. Nous observons expérimentalement et vérifions formellement, en partie, un ordre de complexité croissant des CSP classiques, aux VCSP additifs, en passant par les VCSP possibilistes, et les VCSP lexicographiques.

L'autre classe de simplifications porte sur la simplification de l'objectif. Au lieu de résoudre de façon optimale un problème, même simplifié, nous cherchons juste à le résoudre partiellement pour obtenir un minorant de sa solution optimale.

Une première approche consiste à atteindre un certain niveau de cohérence locale, soit par renforcement de cohérence étendu à l'optimisation, soit par la recherche d'une relaxation localement cohérente optimale.

L'autre approche consiste à adapter le principe du *Branch and Bound* des méthodes complètes, de façon à produire des minorants. Le parcours "meilleur en premier" est une solution. Cela peut se faire aussi en bridant l'algorithme *Depth First Branch and Bound*. Une méthode bien connue est celle appelée *Iterative Deepening*. Elle consiste à rechercher une solution de valuation strictement inférieure à une borne supérieure fixée au début de la résolution. S'il n'y en a pas, cette borne supérieure initiale est un minorant de l'optimum du problème. Le même raisonnement est appliqué successivement pour des bornes supérieures croissantes, jusqu'à trouver une solution, qui est optimale.

Enfin, toutes les simplifications présentées peuvent être combinées. Nous avons examiné certaines combinaisons qui semblent offrir les meilleurs résultats. Rappelons que l'objectif d'avoir une évolution des minorants croissante au cours du temps implique de résoudre successivement des simplifications ou combinaisons de simplifications de moins en moins grossières qui donnent des minorants de plus en plus proches de l'optimum, moyennant des temps de calcul de plus en plus longs. Parmi les exemples prometteurs de combinaisons de simplifications, nous pouvons noter la relaxation des contraintes par niveau combinée avec un changement de la structure de valuation (cf. section 4.3), ou la décomposition d'un problème en grappes combiné à une réduction des domaines (cf. section 5.4), ou enfin une combinaison de l'algorithme *Russian Doll Search* avec le principe de l'*Iterative Deepening* (cf. section 6.2.3).

Dans la suite, le chapitre 4 étudie le changement de structure de valuation. Le chapitre 5 décrit des techniques de relaxation par suppression de contraintes, ou par ajout de n-uplets. Enfin, le chapitre 6 aborde les techniques liées à la simplification de l'objectif.

Chapitre 4

Simplifications par changement de la structure de valuation et relaxation des contraintes par niveau

Ce chapitre présente une technique de simplification originale fondée sur la modification de la structure de valuation des VCSP. C'est donc une simplification structurelle, différente des classiques relaxations de contraintes, qui est d'abord décrite de façon théorique, puis amplement expérimentée et enfin étudiée dans un contexte interruptible. Les principaux résultats exposés dans ce chapitre ont été publiés dans [dGV97c] et [dGV97a].

4.1 Raisonnement sur des simplifications

4.1.1 Simplifications considérées

4.1.1.1 Fonction de simplification

Le type de changement effectué sur la structure de valuation auquel nous nous intéressons dans ce chapitre est restreint de manière à obtenir une résolution plus simple, tout en étant capable de déduire un minorant. Pour cela, nous définissons la notion de fonction de simplification :

Définition 25 ———

Fonction de simplification *:*
soit deux structures de valuation $S = (E, \succ, \top, \bot, \otimes)$ et $S' = (E', \succ', \top', \bot', \otimes')$;
une fonction de simplification Φ est une fonction de E vers E', telle que :

$$\Phi(\bot) = \bot', \Phi(\top) = \top',$$

$$\forall e_1, e_2 \in E, e_1 \succeq e_2 \Rightarrow \Phi(e_1) \succeq' \Phi(e_2)$$

Une fonction de simplification correspond à un morphisme d'ordre [Xuo92] ; c'est une fonction croissante. La deuxième condition de la définition 25 peut se réécrire :

$$\begin{aligned} \forall e_1, e_2 \in E, \quad e_1 = e_2 \quad &\Rightarrow \quad \Phi(e_1) =' \Phi(e_2) \\ e_1 \succ e_2 \quad &\Rightarrow \quad \Phi(e_1) \succeq' \Phi(e_2) \end{aligned}$$

Ainsi, la fonction de simplification peut supprimer certaines différences entre les valuations ($e1, e2 \in E / e1 \succ e2 \wedge \Phi(e1) = \Phi(e2)$), mais préserve l'ordre et les éléments maximum et minimum. Dans l'exemple de la figure 4.1, les fonctions de simplification sont surjectives, mais pas injectives.

La fonction de simplification Φ entre S et S' transforme un problème VCSP $P = (V, D, C, S, \varphi)$ en un problème simplifié $P' = (V, D, C, S', \Phi \circ \varphi)$, noté abusivement $\Phi(P)$.

4.1.1.2 Simplifications retenues

Les simplifications suivantes sont l'objet de notre étude :

- Φ_e, relaxation par niveau des contraintes : quelle que soit la structure de valuation S et e un élément quelconque de E, Φ_e est une simplification de S vers S qui fait correspondre à $\perp$ toute valuation $e1$ dans E strictement inférieure à e et qui laisse inchangées toute les autres valuations ; ainsi, le problème simplifié ignore toutes les contraintes inférieures à e, c'est un cas simple de relaxation, au sens de la *Programmation Linéaire* [NW88] ; voir l'exemple de la figure 4.1 ;

$$\forall e \in E, \quad \Phi_e : \begin{array}{ccll} E & \rightarrow & E & \\ \top & \mapsto & \top & \\ \perp & \mapsto & \perp & \\ f & \mapsto & f & \text{si } f \succeq e \\ f & \mapsto & \perp & \text{si } f \prec e \end{array} \tag{4.1}$$

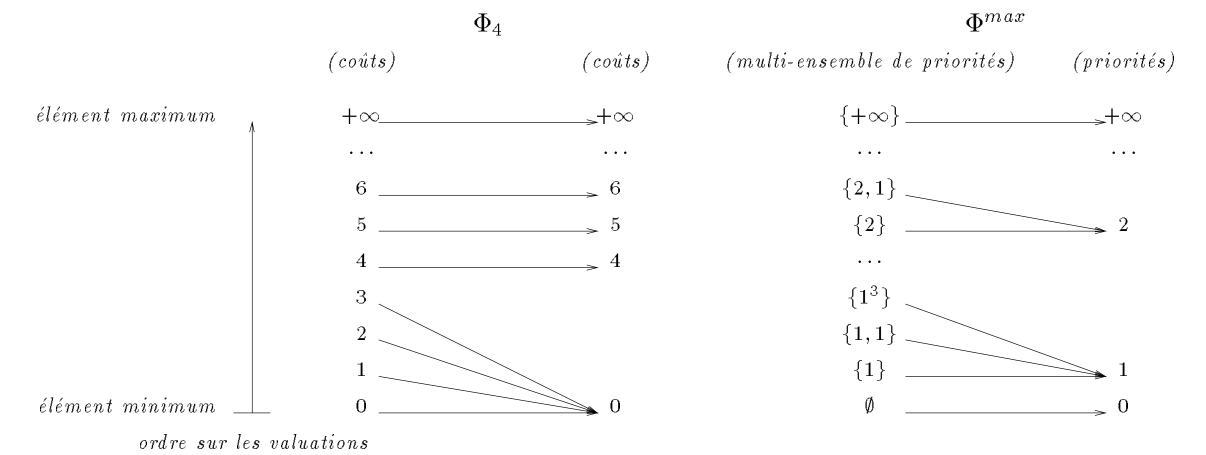


Fig. 4.1 - Exemples de fonctions de simplification : Φ_4 appliqué à un ensemble de coûts (relaxation des contraintes de coût inférieur à 4), et Φ^{max} appliqué à un multi-ensemble de priorités (lexicographique vers possibiliste)

- $\Phi^{\wedge}$, changement d'une structure quelconque S en une structure $\wedge$-VCSP : $\Phi^{\wedge}$ est une simplification telle que $\forall e, e \neq \perp$ alors $\Phi^{\wedge}(e) =$ faux ; ainsi, le problème simplifié est un CSP classique, toutes les contraintes molles sont devenus des contraintes dures ;

$$\begin{array}{llll} \Phi^{\wedge}: & E & \rightarrow & \{\text{vrai}, \text{faux}\} \\ & \top & \mapsto & \text{faux} \\ & \bot & \mapsto & \text{vrai} \\ & f & \mapsto & \text{faux} \qquad\qquad \text{si } f \succ \bot \end{array} \tag{4.2}$$

- Φ^{max}, changement d'une structure additive ou lexicographique en une structure possibiliste max-VCSP : depuis une structure Σ-VCSP, Φ^{max} est simplement la fonction identité ; depuis une structure lex-VCSP, Φ^{max} est une simplification telle que $\forall e, e \neq \emptyset$ alors $\Phi^{max}(e)$ est l'élément maximum dans le multi-ensemble e ; voir l'exemple de la figure 4.1 ; voir aussi la figure 2.2 pour les notations des différentes classes de VCSP ;

$$\begin{array}{llll|llll} \Phi^{max}: & \overline{\mathsf{N}} & \rightarrow & \overline{\mathsf{N}} & \Phi^{max}: & \overline{\mathsf{N}}^* & \rightarrow & \overline{\mathsf{N}} \\ & +\infty & \mapsto & +\infty & & \{+\infty\} & \mapsto & +\infty \\ & 0 & \mapsto & 0 & & \emptyset & \mapsto & 0 \\ & f & \mapsto & f & & f & \mapsto & m \\ & & & & & \multicolumn{3}{l}{\text{avec } m \in \overline{\mathsf{N}}, \text{l'élément maximum du multi-ensemble } f} \end{array} \tag{4.3}$$

- Φ^{lex}, changement d'une structure additive en une structure lexicographique lex-VCSP : Φ^{lex} est une simplification telle que $\forall e, e \neq \emptyset, e \neq +\infty$ alors $\Phi^{lex} = \{e\}$;

$$\begin{array}{llll} \Phi^{lex}: & \overline{\mathsf{N}} & \rightarrow & \overline{\mathsf{N}}^* \\ & +\infty & \mapsto & \{+\infty\} \\ & 0 & \mapsto & \emptyset \\ & f & \mapsto & \{f\} \end{array} \tag{4.4}$$

Nous pouvons facilement vérifier que ces fonctions de simplification respectent bien la définition 25. Les fonctions de simplification Φ^{lex} et Φ^{max} appliquées à une structure additive sont de simples bijections, toutes les autres simplifications sont surjectives.

Notons qu'une composition de fonctions de simplification reste une fonction de simplification. Nous utiliserons dans la section 4.3 des compositions entre la relaxation par niveau, simplification notée Φ_e, et un changement de structure de valuation, simplification notée $\Phi^{\otimes}$, avec $\otimes \in \{\wedge, max, lex\}$.

4.1.2 Résolution plus simple

4.1.2.1 Propriété de compatibilité par agrégation

Pour certaines simplifications, il est possible de confirmer de façon théorique qu'elles sont effectivement plus simples à résoudre. Pour cela, nous avons besoin que la fonction de simplification respecte une propriété supplémentaire de compatibilité par rapport aux opérateurs d'agrégation :

Propriété 1 ————————————————

Compatibilité par agrégation *:*
une fonction de simplification Φ entre deux structures de valuation S et S' est **agrégation-compatible** *ssi $\forall E_1, E_2$, des sous-ensembles de E :*

$$\Phi(\bigotimes_{e \in E_1} e) \succeq' \Phi(\bigotimes_{e \in E_2} e) \Rightarrow \bigotimes'_{e \in E_1} \Phi(e) \succeq' \bigotimes'_{e \in E_2} \Phi(e)$$

————————————————

Remarque 24 La notion de raffinement fort définie dans [SFV95] correspond au terme inverse pour signifier qu'un problème $\Phi(P)$ est une simplification agrégation-compatible de P. En effet, si Φ est une fonction de simplification agrégation-compatible, alors P est un raffinement fort de $\Phi(P)$.

$\diamond$

Nous pouvons alors établir le théorème suivant, illustré par la figure 4.2 :

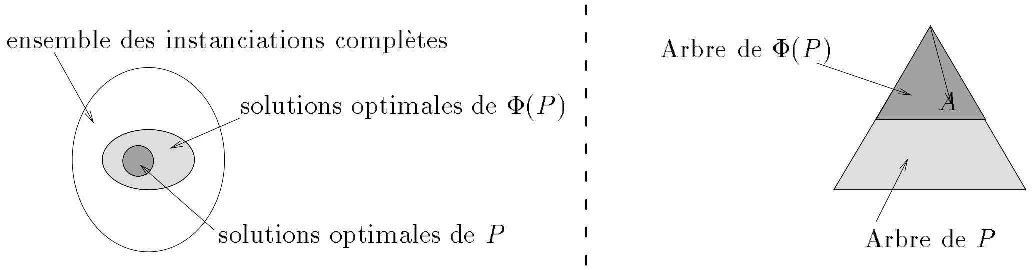


Fig. 4.2 - Propriétés sur la résolution de $\Phi(P)$ et P lorsque Φ est agrégation-compatible

Théorème 9

Si Φ est une fonction de simplification agrégation-compatible, *alors l'ensemble des solutions optimales du problème simplifié $\Phi(P)$ est un sur-ensemble de l'ensemble des solutions optimales du problème initial P.*
De plus, l'arbre de recherche de $\Phi(P)$ est inclus dans l'arbre de recherche de P, lorsque les arbres sont parcourus par un algorithme Depth-First Branch and Bound avec le même ordre statique sur les variables et sur les valeurs, ainsi que la même façon d'évaluer un minorant de la valuation globale d'une instanciation partielle.

Preuve :

si Φ est une fonction de simplification agrégation-compatible, alors nous pouvons en déduire de la définition 25 et de la propriété 1 :

$$\bigotimes_{e\in E_1} e \succeq \bigotimes_{e\in E_2} e \stackrel{\text{D. }25}{\Longrightarrow} \Phi(\bigotimes_{e\in E_1} e) \succeq' \Phi(\bigotimes_{e\in E_2} e) \stackrel{\text{P. }1}{\Longrightarrow} \bigotimes'_{e\in E_1} \Phi(e) \succeq' \bigotimes'_{e\in E_2} \Phi(e) \tag{4.5}$$

Soit $lb(A,P)$, n'importe quel minorant de la valuation globale de A dans P, qui est calculé **par agrégation des valuations de certaines contraintes**, par exemple par *Backward Checking* ou par *Forward Checking* (cf. section 2.3.1.2). De l'équation 4.5, nous en déduisons l'équation 4.6 et sa contraposée 4.7 :

$\forall A, A'$ des instanciations partielles :

$$lb(A,P) \succeq lb(A',P) \quad \Rightarrow \quad lb(A,\Phi(P)) \succeq' lb(A',\Phi(P)) \tag{4.6}$$
$$lb(A',\Phi(P)) \succ' lb(A,\Phi(P)) \quad \Rightarrow \quad lb(A',P) \succ lb(A,P) \tag{4.7}$$

Si A' est une instanciation complète optimale pour P, alors $\forall A$, une instanciation complète, nous avons du fait de l'optimalité de A', $\varphi(A,P) \succeq \varphi(A',P)$. à partir de l'équation 4.6, nous en déduisons que $\varphi(A,\Phi(P)) \succeq \varphi(A',\Phi(P))$ pour toute instanciation complète A, et donc A' est aussi une solution optimale de $\Phi(P)$. Ainsi, l'ensemble des solutions optimales de P est inclus dans l'ensemble des solutions optimales de $\Phi(P)$.

Examinons la propriété pour qu'un noeud soit parcouru par un algorithme *Depth-First Branch and Bound*.
Si A est une instanciation partielle, associée à un noeud de l'arbre de recherche de $\Phi(P)$, et que ce noeud est parcouru par un algorithme *DFBB* avec un ordre statique sur les variables et les valeurs, utilisant un minorant $lb(A, \Phi(P))$ pour évaluer ce noeud, alors $\forall A'$, une instanciation complète, parcourue précédemment dans l'ordre d'exploration de l'arbre, la propriété $\varphi(A', \Phi(P)) \succ' lb(A, \Phi(P))$ est toujours vérifiée. Sinon le noeud A n'aurait pas été parcouru. De l'équation 4.7, nous avons que $\forall A'$, une instanciation complète précédent A dans l'ordre d'exploration, alors $\varphi(A', P) \succ lb(A, P)$ et donc le noeud A est aussi développé dans P, sous les conditions que l'algorithme de parcours soit le même, ainsi que l'ordre sur les variables et les valeurs, ainsi que la façon d'évaluer un minorant de la valuation globale d'une instanciation partielle. Sous ces conditions, l'arbre de recherche de $\Phi(P)$ est inclus dans l'arbre de recherche de P. $\square$

Remarque 25 Notons que l'utilisation d'ordres dynamiques, comme le choix de la variable de plus petit domaine d'abord, ou le choix de l'instanciation partielle ayant le plus petit minorant (stratégie du *Best First Branch and Bound*), invalide le théorème 9 à cause des différentes instanciations complètes qui seront parcourues lors de la résolution de P et $\Phi(P)$. $\diamond$

4.1.2.2 Relation entre les simplifications

Le théorème suivant décrit pour les simplifications considérées, lesquelles sont agrégation-compatibles et lesquelles ne le sont pas :

Théorème 10

- Φ_e *est agrégation-compatible pour* max-VCSP *et* lex-VCSP *sous condition que* e *soit un singleton et que les agrégations portent sur des singletons*[1] *; mais* Φ_e *n'est pas agrégation-compatible pour* Σ-VCSP *;*
- $\Phi^{\wedge}$ *est agrégation-compatible quelle que soit la structure de valuation ;*
- Φ^{max} *est agrégation-compatible pour* lex-VCSP *mais pas pour* Σ-VCSP *;*
- Φ^{lex} *n'est pas agrégation-compatible pour* Σ-VCSP.

Preuve :

Nous établissons d'abord un lemme valable quelle que soit la fonction de simplification :

Lemme 1
Soit $\Phi : E \rightarrow E'$*, une fonction de simplification ; quel que soit l'ensemble fini de valuations* $E_1 \subset E$*, l'élément maximum de* E_1 *est aussi l'élément maximum des éléments de* E_1 *simplifiés par* Φ *:*

$$\forall E_1 \subset E,\ \Phi(\max_{e \in E_1} e) = \max_{e \in E_1} \Phi(e)$$

---

[1] Ces restrictions correspondent à l'utilisation concrète que nous faisons des simplifications Φ_e. En effet, dans le cas de problèmes lex-VCSP, e est égal à la valuation d'une contrainte qui est toujours un singleton. De même, l'agrégation des valuations, utilisée pour calculer le minorant de la valuation globale d'une instanciation partielle, est toujours une agrégation de valuations qui sont des singletons associés aux contraintes.

En effet, soit e_1, l'élément le plus grand de l'ensemble E_1 : $e_1 \in E_1$ tel que $e_1 = \max_{e \in E_1} e$; cet élément maximum e_1 a comme propriété de rester maximum dans l'ensemble des valuations simplifiées, quelle que soit la simplification employée :

$$\forall e \in E_1, e_1 \geq e \overset{\text{Déf. 25}}{\Longrightarrow} \Phi(e_1) \geq \Phi(e)$$

d'où,

$$\Phi(e_1) = \max_{e \in E_1} \Phi(e)$$

Nous établissons maintenant les preuves pour chaque fonction de simplification référencée dans le théorème 10 :

- Φ_l : *max*-VCSP $\rightarrow$ *max*-VCSP est agrégation-compatible :

$$\Phi_l(\max_{e \in E_1} e) \geq \Phi_l(\max_{e \in E_2} e) \overset{\text{Lem. 1}}{\Longrightarrow} \max_{e \in E_1} \Phi_l(e) \geq \max_{e \in E_2} \Phi_l(e)$$

- Φ_l : *lex*-VCSP $\rightarrow$ *lex*-VCSP est agrégation-compatible avec la restriction que l est un singleton, noté $\{k\}$ avec $k \in \mathsf{N}$ et qu'il n'y a que des agrégations de singletons[2] :

  soit $i \in \mathsf{N}$, l'élément maximum du multi-ensemble $\bigcup_{e \in E_1} e$ et j l'équivalent pour $\bigcup_{e \in E_2} e$;

  - si $i \geq k$ et $j \geq k$ alors $\bigcup_{e \in E_1} e \geq^* l$ et $\bigcup_{e \in E_2} e \geq^* l$,

    ainsi $\Phi_l(\bigcup_{e \in E_1} e) = \bigcup_{e \in E_1} e$ et $\Phi_l(\bigcup_{e \in E_2} e) = \bigcup_{e \in E_2} e$;

    par ailleurs, tout singleton e inférieur à l est supprimé par la simplification Φ_l ; donc l'agrégation $\bigcup_{e \in E_1} \Phi_l(e)$ ne contient que les éléments provenant des singletons supérieurs ou égaux à l ; d'après la définition de l'ordre lexicographique $>^*$, nous obtenons que :

$$\bigcup_{e \in E_1} e \geq^* \bigcup_{e \in E_2} e \Rightarrow \bigcup_{e \in E_1} \Phi_l(e) \geq^* \bigcup_{e \in E_2} \Phi_l(e)$$

    finalement, nous avons bien :

$$\begin{aligned} \Phi_l(\bigcup_{e \in E_1} e) \geq^* \Phi_l(\bigcup_{e \in E_2} e) &\Rightarrow \bigcup_{e \in E_1} e \geq^* \bigcup_{e \in E_2} e \\ &\Rightarrow \bigcup_{e \in E_1} \Phi_l(e) \geq^* \bigcup_{e \in E_2} \Phi_l(e) \end{aligned}$$

  - si $j < k$ alors $\forall e \in E_2 \Rightarrow \Phi_l(e) = \emptyset$; dans ce cas trivial,

$$\Phi_l(\bigcup_{e \in E_1} e) \geq^* \emptyset \Rightarrow \bigcup_{e \in E_1} \Phi(e) \geq^* \emptyset$$

  - si $i < k$ alors $\forall e \in E_2 \Rightarrow \Phi_l(e) = \emptyset$; soit $j < k$, dans ce cas prémisse et conclusion de l'implication d'agrégation-compatible sont nulles ; soit $j \geq k$, il est impossible d'établir la prémisse ;

- Φ_l : Σ-VCSP $\rightarrow$ Σ-VCSP n'est pas agrégation-compatible :

  voici un contre-exemple,

$$\Phi_2(1+1+1) \geq \Phi_2(2) \not\Rightarrow 0+0+0 \geq 2$$

[2] Sinon, voici un contre-exemple : $\Phi_{\{10^3\}}(\{10^2\} \cup \{10^2\}) \geq^* \Phi_{\{10^3\}}(\{10^3\}) \not\Rightarrow \emptyset \geq^* \{10^3\}$.

- $\Phi^{\wedge}$ est agrégation-compatible quelle que soit la structure de valuation :

  si $\Phi^{\wedge}(\underset{e\in E_1}{\otimes} e) \succeq \Phi^{\wedge}(\underset{e\in E_2}{\otimes} e)$ est vrai alors :

  - soit $\underset{e\in E_1}{\otimes} e \succ \bot$ et $\underset{e\in E_2}{\otimes} e \succ \bot$;

    ainsi, $\exists e_1 \in E_1 \,/\, e_1 \succ \bot$ et $\exists e_2 \in E_2 \,/\, e_2 \succ \bot$; donc $\Phi^{\wedge}(e_1) =$ faux et $\Phi^{\wedge}(e_2) =$ faux ;

    puisque la valuation faux est absorbante dans $\wedge$-VCSP, $\underset{e\in E_1}{\wedge} \Phi^{\wedge}(e) =$ faux, ainsi que $\underset{e\in E_1}{\wedge} \Phi^{\wedge}(e) =$ faux ;
  - soit $\underset{e\in E_1}{\otimes} e \succ \bot$ et $\underset{e\in E_2}{\otimes} e = \bot$;

    ainsi $\exists e_1 \in E_1 \,/\, \Phi^{\wedge}(e_1) =$ faux et $\forall e \in E_2, e = \bot \Rightarrow \Phi^{\wedge}(e) = \Phi^{\wedge}(\bot) =$ vrai ;

    donc $\underset{e\in E_1}{\wedge} \Phi^{\wedge}(e) =$ faux et $\underset{e\in E_2}{\wedge} \Phi^{\wedge}(e) =$ vrai ;
  - soit $\underset{e\in E_1}{\otimes} e = \bot$ et $\underset{e\in E_2}{\otimes} e = \bot$;

    donc $\underset{e\in E_1}{\wedge} \Phi^{\wedge}(e) =$ vrai et $\underset{e\in E_2}{\wedge} \Phi^{\wedge}(e) =$ vrai ;

  finalement, quels que soient les cas précédents, l'implication est bien vérifiée :

$$\Phi^{\wedge}(\underset{e\in E_1}{\otimes} e) \succeq \Phi^{\wedge}(\underset{e\in E_2}{\otimes} e) \Rightarrow \underset{e\in E_1}{\wedge} \Phi^{\wedge}(e) \succeq \underset{e\in E_2}{\wedge} \Phi^{\wedge}(e)$$

- Φ^{max} : lex-VCSP $\rightarrow$ max-VCSP est agrégation-compatible :

  soit $i \in \mathsf{N}$, l'élément maximum du multi-ensemble $\underset{e\in E_1}{\cup} e$; j l'équivalent pour $\underset{e\in E_2}{\cup} e$;

  par définition, $\Phi^{max}(\underset{e\in E_1}{\cup} e) = i$ et $\Phi^{max}(\underset{e\in E_2}{\cup} e) = j$;

  de plus, $\Phi^{max}(\max_{e\in E_1} e) \overset{\text{Lem. 1}}{=} \max_{e\in E_1} \Phi^{max}(e) = i$;

  de même, $\Phi^{max}(\max_{e\in E_2} e) \overset{\text{Lem. 1}}{=} \max_{e\in E_2} \Phi^{max}(e) = j$;

  finalement,

$$\begin{aligned}\Phi^{max}(\underset{e\in E_1}{\cup} e) \geq \Phi^{max}(\underset{e\in E_2}{\cup} e) \quad &\Rightarrow \quad i \geq j \\ &\Rightarrow \quad \max_{e\in E_1} \Phi^{max}(e) \geq \max_{e\in E_2} \Phi^{max}(e)\end{aligned}$$

- Φ^{max} : Σ-VCSP $\rightarrow$ max-VCSP n'est pas agrégation-compatible :

  voici un contre-exemple,

$$\Phi^{max}(1+1+1) \geq \Phi^{max}(2) \not\Rightarrow 1 \geq 2$$

- Φ^{lex} : Σ-VCSP $\rightarrow$ lex-VCSP n'est pas agrégation-compatible :

  voici un contre-exemple,

$$\Phi^{lex}(1+1+1) \geq^{*} \Phi^{lex}(2) \not\Rightarrow \{1^3\} \geq^{*} \{2\}$$

$\square$

Composition de fonctions de simplification

Nous pouvons noter que du fait de la propriété de transitivité de l'inclusion, la composition de deux fonctions de simplification agrégation-compatibles respecte les propriétés d'inclusion de l'arbre de recherche et de l'ensemble des solutions du théorème 9. Soit deux fonctions de simplification agrégation-compatibles Φ^1 et Φ^2 ; alors $\Phi^1 \circ \Phi^2(P)$ est plus simple à résoudre que P (et aussi plus simple que $\Phi^2(P)$).

La composition de deux fonctions de simplification n'est pas toujours commutative. Par exemple $\Phi^{\wedge} \circ \Phi_e \neq \Phi_e \circ \Phi^{\wedge}$; en revanche, $\Phi^{max} \circ \Phi_e = \Phi_e \circ \Phi^{max}$ et $\Phi^{lex} \circ \Phi_e = \Phi_e \circ \Phi^{lex}$.

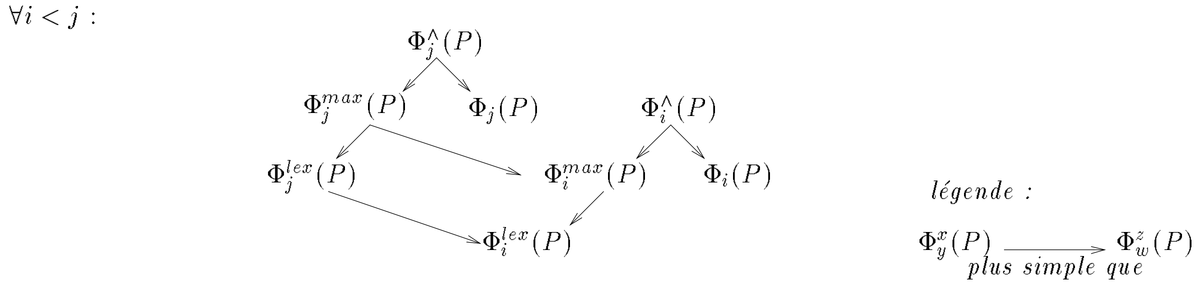


Fig. 4.3 - Ordre partiel sur les simplifications

Soit P, un problème Σ-VCSP, la figure 4.3 résume l'ordre partiel obtenu sur des simplifications de P, ainsi que sur des combinaisons entre relaxation par niveau et changement de la structure de valuation. La simplification du problème P, notée $\Phi_e^{lex}(P)$, correspond au problème P d'abord simplifié par Φ_e puis par Φ^{lex}, soit $\Phi^{lex} \circ \Phi_e(P)$. Une flèche entre $\Phi_y^x(P)$ et $\Phi_w^z(P)$ indique que le problème P simplifié par $\Phi^x \circ \Phi_y$ est plus simple à résoudre, suivant les conditions du théorème 9, que le problème P simplifié par $\Phi^z \circ \Phi_w$. Nous verrons dans la section 4.2 comment peuvent s'ordonner de façon plus complète, en terme de temps moyen de résolution sur des VCSP aléatoires, l'ensemble des simplifications.

4.1.3 Construction d'un minorant

A partir de l'optimum d'un problème simplifié, ou plus généralement d'un minorant de son optimum, il est possible de construire un minorant de l'optimum du problème initial. Pour effectuer ce calcul, nous avons besoin de définir la notion de fonction de transfert.

4.1.3.1 Fonction de transfert

Définition 26
Fonction de transfert :
soit une fonction de simplification Φ entre deux structures de valuation S et S' ;
une fonction de transfert Ψ est une fonction de E' vers E telle que :

$$\forall E_1,\ \textit{multi-ensemble fini d'éléments choisis dans } E,\ \forall e' \in E',\ \bigotimes_{e \in E_1}' \Phi(e) \succeq' e' \Rightarrow \bigotimes_{e \in E_1} e \succeq \Psi(e')$$

Ainsi, quel que soit le minorant e' de la valuation dans $\Phi(P)$ d'une instanciation complète A, la fonction de transfert Ψ fournie un minorant $\Psi(e')$ de la valuation dans P de A.

Une fonction de transfert est donc associée à une fonction de simplification. Elle va nous permettre de convertir un minorant, dans E', d'un problème simplifié, en un minorant, dans E, du problème initial :

Théorème 11 ———

Si Ψ est une fonction de transfert pour la fonction de simplification Φ entre S et S', et si e' est un minorant de l'optimum de $\Phi(P)$, noté abusivement $\varphi(\Phi(P))$, alors $\Psi(e')$ est un minorant de l'optimum de P, noté $\varphi(P)$.
En particulier :

$$\forall P\,,\ \varphi(P) \succeq \Psi(\varphi(\Phi(P)))$$

Preuve :

Soit e', un minorant de l'optimum de $\Phi(P)$:

$$\begin{array}{ll} & \forall A,\ \text{une instanciation complète},\ \varphi(A, \Phi(P)) \succeq' e' \\ \overset{\text{Déf. 26}}{\Longrightarrow} & \forall A,\ \text{une instanciation complète},\ \varphi(A, P) \succeq \Psi(e') \\ \Longrightarrow & \varphi(P) \succeq \Psi(e') \end{array}$$

□

Notons qu'une fonction de transfert triviale existe quelle que soit la fonction de simplification associée : $\forall e' \in E'$, $\Psi(e') = \bot$. Par ailleurs, quelle que soit la fonction de transfert Ψ, nous avons toujours $\Psi(\bot) = \bot$. Heureusement, nous pouvons exhiber des fonctions de transfert plus utiles pour les fonctions de simplification retenues.

Optimalité d'une fonction de transfert

Nous définissons la notion d'optimalité d'une fonction de transfert en précisant les informations dont elle dispose. Nous supposons qu'une fonction de transfert élabore son résultat à partir d'une valuation, notée e', qui peut être l'optimum du problème simplifié ou seulement un minorant, et d'une connaissance de la distribution des valuations associées aux contraintes. Cette distribution est un multi-ensemble de valuations, noté E, tel que $E = \varphi(C) \equiv$ [3] $\{\varphi(c), \forall\, c \in C\}$. Nous étendons donc la définition 26 d'une fonction de transfert pour ajouter le paramètre E : $\Psi(e', E)$.

Définition 27 ———
Fonction de transfert optimale *:*
soit Φ, une fonction de simplification entre $\otimes$-VCSP et $\otimes'$-VCSP ;
soit Ψ, une fonction de transfert associée à Φ ;
une fonction de transfert Ψ est dite optimale ssi :

$$\begin{array}{lll} \forall\, E & & \\ \forall e' = \displaystyle\bigotimes'_{e \in E_1\, /\, E_1 \subseteq E} \Phi(e) & & \\ \exists P = (V, D, C, S, \varphi) & \textit{tel que} & \varphi(\Phi(P)) = e' \\ & & \varphi(C) = E \\ & & \Psi(e', E) = \varphi(P) \end{array} \tag{4.8}$$

[3] Nous notons abusivement $\varphi(C)$, le multi-ensemble des valuations associées aux contraintes.

L'optimalité d'une fonction de transfert est établie en exhibant, quel que soit le multi-ensemble de valuation E et quelle que soit la valuation e', égale à l'agrégation dans Φ de certaines valuations de E, un problème P tel que l'optimum de sa simplification soit égal à e', telle que la distribution des valuations associées à ses contraintes soit égale à E et tel que le minorant retourné par la fonction de transfert Ψ, avec e' et E comme paramètres, soit égal à l'optimum du problème P.

Le théorème suivant affirme que si Ψ est une fonction de transfert optimale, alors il n'existe pas d'autres fonctions de transfert qui retournent des résultats meilleurs, quels que soient e' et E :

Théorème 12 ---

Soit Ψ, une fonction de transfert optimale associée à Φ ;
alors,

$$\begin{array}{ll} \forall\, \Psi' & \textit{une fonction de transfert associée à } \Phi \\ \forall\, E & \\ \forall e' = \displaystyle\bigotimes_{e \in E_1 \,/\, E_1 \subseteq E}{}' \Phi(e) & \\ & \Psi'(e', E) \preceq \Psi(e', E) \end{array} \tag{4.9}$$

Preuve :

Nous établissons une preuve par l'absurde. Supposons que Ψ est une fonction de transfert optimale associée à Φ et que :

$$\begin{array}{ll} \exists\, \Psi' & \text{une fonction de transfert associée à } \Phi \\ \exists\, E & \\ \exists e' = \displaystyle\bigotimes_{e \in E_1 \,/\, E_1 \subseteq E}{}' \Phi(e) & \\ & \Psi'(e', E) \succ \Psi(e', E) \end{array}$$

Puisque Ψ est optimale, d'après la propriété 4.8, nous savons qu'il existe un problème $P = (V, D, C, S, \varphi)$ tel que :

$$\begin{array}{l} \varphi(\Phi(P)) = e' \\ \varphi(C) = E \\ \Psi(e', E) = \varphi(P) \end{array}$$

De plus, comme Ψ' est aussi une fonction de transfert, nous pouvons appliquer le théorème 11 qui nous donne l'inéquation suivante :

$$\Psi'(e', E) \preceq \varphi(P)$$

De cette déduction, ainsi que la précédente, $\Psi(e', E) = \varphi(P)$, nous obtenons immédiatement une contradiction avec notre hypothèse que $\Psi'(e', E) \succ \Psi(e', E)$. $\square$

4.1.3.2 Application aux simplifications retenues

Nous définissons les notations suivantes qui seront employées dans la définition des fonctions de transfert :

soit P, un problème VCSP ;

soit $(e_1, \ldots, e_k)$, la liste des différentes valuations qui sont associées aux contraintes de P par φ ; les e_i sont ordonnées par valuation décroissante ;

soit n_i, le nombre de contraintes dans P dont la valuation est égale à e_i ;

soit m_i, $m_i \leq n_i$, le nombre de contraintes dans P qui sont agrégées dans le calcul d'un minorant de la valuation globale d'une instanciation partielle A et dont les valuations sont égales à e_i ;

soit $e' = \{e_1^{m_1}, \ldots, e_k^{m_k}\}$, la notation utilisée pour représenter la valuation lexicographique d'un tel minorant ; en revanche une valuation lexicographique composée d'un seul élément, normalement notée $\{e\}$, peut être aussi notée par e de manière à regrouper plusieurs définitions de fonctions de transfert en une seule.

Nous décrivons pour chaque fonction de simplification retenue, la fonction de transfert optimale associée :

Théorème 13

Les fonctions de transfert suivantes sont optimales :

- Φ_e : $\otimes$-VCSP $\rightarrow$ $\otimes$-VCSP *($\otimes \in \{\wedge, max, lex, \Sigma\}$) a pour fonction de transfert l'identité :*

$$\forall e' \in E',\ \Psi_e(e') = e'$$

- $\Phi^{\wedge}$: $\otimes$-VCSP $\rightarrow$ $\wedge$-VCSP *($\otimes \in \{max, lex, \Sigma\}$) a pour fonction de transfert, dépendante des données de P, $\Psi^{\wedge}$: $\wedge$-VCSP $\rightarrow$ $\otimes$-VCSP telle que :*

$$\begin{cases} \Psi^{\wedge}(\text{vrai}) = \perp \\ \Psi^{\wedge}(\text{faux}) = e_k \end{cases}$$

- Φ^{max} : $\otimes$-VCSP $\rightarrow$ max-VCSP *($\otimes \in \{lex, \Sigma\}$) a pour fonction de transfert l'identité :*

$$\forall e' \in E^{max},\ \Psi^{max}(e') = e'$$

- Φ^{lex} : Σ-VCSP $\rightarrow$ lex-VCSP *a pour fonction de transfert, dépendante des données de P, Ψ^{lex} : lex-VCSP $\rightarrow$ Σ-VCSP telle que :*

$$\begin{cases} \Psi^{lex}(\{+\infty\}) = +\infty \\ \Psi^{lex}(\{e_1^{m_1}, \ldots, e_k^{m_k}\}) = lex(\{e_1^{m_1}, \ldots, e_k^{m_k}\}) \end{cases}$$

$lex(\emptyset) = 0$
$lex(\{e_i^{m_i}, \ldots, e_k^{m_k}\}) =$
  `si` $m_i = n_i$ `alors`
    $(n_i.e_i) + lex(\{e_{i+1}^{m_{i+1}}, \ldots, e_k^{m_k}\})$
  `sinon`
    $\min[\,(m_i+1).e_i,\ (m_i.e_i) + lex(\{e_{i+1}^{m_{i+1}}, \ldots, e_k^{m_k}\})\,]$
  `finsi`

Preuve :

Nous démontrons que ce sont toutes des fonctions de transfert et quelles sont optimales :

- Ψ_l : $\otimes$-VCSP $\rightarrow$ $\otimes$-VCSP ($\otimes \in \{\wedge, max, lex, \Sigma\}$) est une fonction de transfert optimale :
  $\forall E_1$, multi-ensemble fini d'éléments choisis dans E, $\forall e' \in E$:

$$\begin{aligned}\bigotimes_{e\in E_1} \Phi_l(e) \succeq e' \quad &\Rightarrow \quad \bigotimes_{e\in E_1 \,/\, e\succeq l} e = \bigotimes_{e\in E_1} \Phi_l(e) \succeq e'\\ &\Rightarrow \quad \bigotimes_{e\in E_1} e \succeq \bigotimes_{e\in E_1 \,/\, e\succeq l} e \succeq e'\end{aligned}$$

  Pour vérifier l'optimalité de Ψ à l'aide de la propriété 4.8, nous construisons un problème P ayant une solution optimale qui satisfait toutes les contraintes de valuation inférieure à l et qui minimise la violation des autres contraintes ; dans ce cas $\varphi(P) = \varphi(\Phi_l(P)) = \Psi_l(\varphi(\Phi_l(P)))$;

- $\Psi^{\wedge}$: $\wedge$-VCSP $\rightarrow$ $\otimes$-VCSP ($\otimes \in \{max, lex, \Sigma\}$) est une fonction de transfert optimale :
  $\forall E_1$ inclus dans le multi-ensemble des valuations des contraintes de P :
  - soit $\bigwedge_{e\in E_1} \Phi^{\wedge}(e) \succeq$ vrai :

    la conclusion $\bigotimes_{e\in E_1} e \succeq \perp$ est toujours vraie ;

    soit P, un problème cohérent ; alors $\varphi(P) = \perp$; donc il n'existe pas de fonction de transfert $\Psi^{\wedge\prime}$ meilleure que $\Psi^{\wedge}$;
  - soit $\bigwedge_{e\in E_1} \Phi^{\wedge}(e) =$ faux :

    il existe au moins une contrainte violée dans E_1 ; donc $\bigotimes_{e\in E_1} e \succ \perp \Rightarrow \bigotimes_{e\in E_1} e \succeq \min_{e\in E_1} e$; dans P, la plus petite valuation non nulle est égale à e_k ;

    soit P, un problème ayant une solution optimale qui satisfait toutes les contraintes sauf une de valuation e_k ; alors la propriété 4.8 est vérifiée : $\varphi(P) = e_k = \Psi^{\wedge}(\text{faux})$;

- $\Psi^{\max}$: max-VCSP $\rightarrow$ $\otimes$-VCSP ($\otimes \in \{lex, \Sigma\}$) est une fonction de transfert optimale :
  $\forall E_1$, multi-ensemble fini d'éléments choisis dans E, $\forall e' \in E^{max}$:

$$\begin{aligned}\max_{e\in E_1} \Phi^{max}(e) \geq e' \quad &\overset{Lem.1}{\Longrightarrow} \quad \Phi^{max}(\max_{e\in E_1} e) \geq e'\\ &\Rightarrow \quad \max_{e\in E_1} e \succeq e'\\ &\Rightarrow \quad \bigotimes_{e\in E_1} e \succeq e'\end{aligned}$$

  soit P, un problème ayant une solution optimale qui satisfait toutes les contraintes sauf une de valuation e' ; alors le théorème 4.8 est vérifiée : $\varphi(P) = e' = \Psi^{\max}(e')$;

- Ψ^{lex} : lex-VCSP $\rightarrow$ Σ-VCSP est une fonction de transfert optimale :

  si le problème simplifié $\Phi^{lex}(P)$ est incohérent alors le problème P est aussi incohérent :

$$\Psi^{lex}(\{+\infty\}) = +\infty$$

  $\forall E_1$ inclus dans le multi-ensemble des valuations des contraintes de P, $\forall e' = \{e_1^{m_1}, \ldots, e_k^{m_k}\}$:
    supposons que :

$$\bigcup_{e\in E_1} \Phi^{lex}(e) \geq^* \{e_1^{m_1}, \ldots, e_k^{m_k}\}$$

    posons $\bigcup_{e\in E_1} \Phi^{lex}(e) = \{e_1^{m'_1}, \ldots, e_k^{m'_k}\}$;

    notons que $m'_i \leq n_i$ et $m_i \leq n_i$;

    chaque valuation e_i représente une valuation du multi-ensemble E_1 ; ainsi :

$$\sum_{e\in E_1} e = \sum_{i=1}^{k} m'_i.e_i$$

nous examinons alors les implications de l'hypothèse :

$$\{e_1^{m'_1}, \ldots, e_k^{m'_k}\} \geq^* \{e_1^{m_1}, \ldots, e_k^{m_k}\}$$

– soit $\{e_1^{m'_1}, \ldots, e_k^{m'_k}\} =^* \{e_1^{m_1}, \ldots, e_k^{m_k}\}$;
  dans ce cas, $\sum_{e \in E_1} e = \sum_{i=1}^{k} m_i.e_i$
– soit $\{e_1^{m'_1}, \ldots, e_k^{m'_k}\} >^* \{e_1^{m_1}, \ldots, e_k^{m_k}\}$:

  * soit $m'_1 > m_1$ et $m_1 < n_1$;
    dans ce cas, nous sommes sûr que $\sum_{e \in E_1} e \geq (m_1 + 1).e_1$
  * soit $m'_1 = m_1$ et $\{e_2^{m'_2}, \ldots, e_k^{m'_k}\} >^* \{e_2^{m_2}, \ldots, e_k^{m_k}\}$;
    dans ce cas, $\sum_{e \in E_1} e \geq m_1.e_1 + X_1$, avec $X_1 \in \mathsf{N}$ tel que :
    · soit $m'_2 > m_2$ et $m_2 < n_2$;
      dans ce cas, $X_1 \geq (m_2 + 1).e_2$
    · soit $m'_2 = m_2$ et $\{e_3^{m'_3}, \ldots, e_k^{m'_k}\} >^* \{e_3^{m_3}, \ldots, e_k^{m_k}\}$;
      dans ce cas, $X_1 \geq m_2.e_2 + X_2$; avec $X_2 \in \mathsf{N}$;

      de façon récursive, nous pouvons appliquer le même raisonnement pour les X_i suivants, jusqu'à X_{k-1} ; enfin, $X_k = 0$;

dans tous les cas, nous avons comme minorant de $\sum_{e \in E_1} e$:

$$\sum_{e \in E_1} e \geq \min\Big(\sum_{i=1}^{k} m_i.e_i \;,\; \texttt{si } (m_1 < n_1)\texttt{alors } (m_1+1).e_1 \;,\; m_1.e_1 + \min\big(\texttt{si } (m_2 < n_2)\texttt{alors } (m_2+1).e_2 \;,\; m_2.e_2 + \min(\ldots)\big)\Big)$$

ainsi,

$$\sum_{e \in E_1} e \geq \min\Big(\sum_{i=1}^{k} m_i.e_i \;,\; lex(\{e_1^{m_1}, \ldots, e_k^{m_k}\})\Big)$$

or, $\sum_{i=1}^{k} m_i.e_i \geq lex(\{e_1^{m_1}, \ldots, e_k^{m_k}\})$;

finalement, nous obtenons bien l'implication :

$$\bigcup_{e \in E_1} \Phi^{lex}(e) \geq^* \{e_1^{m_1}, \ldots, e_k^{m_k}\} \Rightarrow \sum_{e \in E_1} e \geq lex(\{e_1^{m_1}, \ldots, e_k^{m_k}\})$$

Quelles que soient les valeurs des e_i de l'ensemble de valuation E et quelle que soit la valuation $e' = \{e_1^{m_1}, \ldots, e_k^{m_k}\}$, il est possible de construire un problème, composé des contraintes valuées suivant E, dont l'optimum soit égal à $lex(\{e_1^{m_1}, \ldots, e_k^{m_k}\})$ et tel que l'optimum du problème simplifié soit égal à e'. Par exemple, dans le cas où les différentes valuations de E sont espacées de façon lexicographique (cf. section 4.2.1), alors tout problème P, avec $\varphi(C) = E$ possède le même optimum que sa simplification lexicographique : $\varphi(P) = \sum_{i=1}^{k} m_i.e_i = lex(\{e_1^{m_1}, \ldots, e_k^{m_k}\}) = \Psi^{lex}(\varphi(\Phi^{lex}(P)))$. Dans le cas où les valuations de E sont telles que $lex(\{e_1^{m_1}, \ldots, e_k^{m_k}\}) = (m_1 + 1).e_1$, avec la valuation initiale $e' = \{e_1^{m_1}, \ldots, e_k^{m_k}\}$, alors le problème ayant une solution optimale satisfaisant toutes les contraintes sauf $m_1 + 1$ contraintes de coût e_1, et ayant une solution sous-optimale satisfaisant m_1 contraintes de coût e_1, vérifie également la propriété 4.8. De la même façon, il est possible de faire ce raisonnement pour les autres résultats de la fonction lex.

□

Etude de la fonction de transfert Ψ^{lex}

La fonction de transfert Ψ^{lex} est la seule fonction, parmi les fonctions de transfert retenues, qui ne soit pas triviale.

Etudions le comportement de Ψ^{lex} dans le cas où nous résolvons de manière optimale un problème $\Phi^{lex}(P)$. Pour examiner l'efficacité de Ψ^{lex} pour déduire un minorant proche de l'optimum de P, nous étudions la différence qu'il peut y avoir entre $\varphi(P)$ et $\Psi^{lex}(\varphi(\Phi^{lex}(P)))$: une grande (resp. faible) différence correspond à un cas défavorable (resp. favorable).

Un cas défavorable serait d'avoir $\varphi(\Phi^{lex}(P)) = \{e_1^0, e_2^{n_2} \ldots, e_k^{n_k}\}$. En effet, $\Psi^{lex}(\{e_1^0, e_2^{n_2} \ldots, e_k^{n_k}\}) = e_1$ (en supposant que $e_1 < n_2.e_2$). En revanche, l'optimum $\varphi(P)$ peut être égal à $\sum_{i=2}^{k} n_i.e_i$. La différence entre l'optimum de P et son minorant est égale à $\sum_{i=2}^{k} n_i.e_i - e_1$.

Le cas le plus favorable arrive lorsque $\varphi(\Phi^{lex}(P)) = \{e_1^{m_1}, \ldots, e_k^{m_k}\}$ et $\Psi^{lex}(\{e_1^{m_1}, \ldots, e_k^{m_k}\}) = \sum_{i=1}^{k} m_i.e_i$. Dans ce cas, obligatoirement, $\varphi(P) = \sum_{i=1}^{k} m_i.e_i$. La différence est nulle.

En résumé, les problèmes Σ-VCSP favorables à l'obtention d'un bon minorant, c-à-d proche de l'optimum, ont comme caractéristiques :

- les différents coûts des contraintes sont largement espacés, au mieux de façon lexicographique (cf. section 4.2.1) ;
- les contraintes de coût maximum e_1 sont fortement incohérentes ;
- les contraintes sont presque toutes satisfaites ; à l'exception des contraintes de plus faible coût.

Fonction de transfert d'une composition de simplifications

Soit P, un problème VCSP ; nous employons les mêmes notations que celles définies au début de la section 4.1.3.2.

Nous étudions la composition de deux fonctions de simplification : une relaxation des contraintes par niveau Φ_{e_l}, suivie d'un changement de la structure de valuation $\Phi^{\otimes}$ ($\otimes \in \{\wedge, max, lex\}$).

Les fonctions de transfert résultantes, notées $\Psi_{e_l}^{\otimes}$, sont obtenues en appliquant les fonctions de transfert $\Psi^{\otimes}$, définies précédemment, à un problème simplifié $\Phi_{e_l}(P)$ qui ne contient plus qu'une liste restreinte de valuations différentes, $(e_1, \ldots, e_l)$, associées à ses contraintes :

- $\Psi_{e_l}^{\wedge}$: $\wedge$-VCSP $\rightarrow$ $\otimes$-VCSP ($\otimes \in \{max, lex, \Sigma\}$) :

$$\begin{cases} \Psi_{e_l}^{\wedge}(\text{vrai}) = \perp \\ \Psi_{e_l}^{\wedge}(\text{faux}) = e_l \end{cases}$$

- $\Psi^{max}_{e_l} : max\text{-VCSP} \rightarrow \otimes\text{-VCSP}\ (\otimes \in \{lex, \Sigma\})$:

$$\forall e' \in \{e_1, \ldots, e_l, 0\}\ \Psi^{max}_{e_l}(e') = e'$$

- $\Psi^{lex}_{e_l} : lex\text{-VCSP} \rightarrow \Sigma\text{-VCSP}$:

$$\begin{cases} \Psi^{lex}_{e_l}(\{+\infty\}) = +\infty \\ \Psi^{lex}_{e_l}(\{e_1^{m_1}, \ldots, e_l^{m_l}\}) = lex(\{e_1^{m_1}, \ldots, e_l^{m_l}\}) \end{cases}$$

Utilisation des fonctions de transfert

Le calcul d'un minorant à l'aide d'une fonction de transfert peut intervenir principalement dans les situations suivantes :

- la résolution complète d'un problème simplifié $\Phi(P')$ fournit l'optimum de $\Phi(P')$; cet optimum permet de déduire un minorant de l'optimum de P' (cf. théorème 11) ;
- au cours de la résolution d'un problème simplifié $\Phi(P')$, certaines méthodes complètes, par exemple, l'algorithme *Best-First Branch and Bound*, fournissent une succession de minorants croissants de l'optimum de $\Phi(P')$, jusqu'à produire l'optimum ; ces minorants permettent de déduire des minorants croissants[4] de l'optimum de P' (cf. théorème 11) ;
- au cours de la résolution d'un problème simplifié $\Phi(P')$, les méthodes complètes déterminent un minorant de la valuation globale de l'instanciation partielle courante pour tester si elle doit être étendue ; l'application d'une fonction de transfert sur ce minorant permet d'obtenir un minorant dans P' de cette même instanciation partielle[5] ; ainsi, la production d'instanciations partielles globalement incohérentes (cf. *Nogoods* valués dans la section 2.3.1.4) au niveau du problème simplifié $\Phi(P')$ est réutilisable au niveau du problème cible P'.

Remarque 26 Le problème P', pour lequel un minorant est calculé, peut être lui même une simplification d'un problème initial P. Une utilisation des fonctions de transfert entre simplifications sera faite dans la section 4.3. ◇

4.1.4 Construction d'un majorant

Construction d'un majorant à partir d'une solution

Toute instanciation complète dans $\Phi(P)$ fournit simplement un majorant de l'optimum de P. Il suffit d'évaluer cette instanciation dans P. Ainsi, les solutions produites lors de la résolution d'un problème simplifié $\Phi(P)$ fournissent aussi des majorants de l'optimum du problème P.

Construction d'un majorant à partir d'une valuation

Il est aussi possible de déduire un majorant de P en connaissant uniquement la valuation d'une solution de $\Phi(P)$. Tandis que les fonctions de transfert de la section 4.1.3 concernent la production de minorants, les implications suivantes permettent de produire des majorants :

[4]Toutes les fonctions de transfert du théorème 13 sont des fonctions croissantes : $lb(\Phi(P)) \succeq lb'(\Phi(P)) \Rightarrow \Psi(lb(\Phi(P))) \succeq \Psi(lb'(\Phi(P)))$.

[5]Cela nécessite de résoudre $\Phi(P')$ et P' avec les mêmes heuristiques statiques.

soit les notations employées au début de la section 4.1.3.2, à l'exception que P est un problème Σ-VCSP ; soit $\varphi(A, \Phi^{\otimes}_{e_l}(P))$, la valuation d'une solution A dans le problème simplifié $\Phi^{\otimes}_{e_l}(P)$:

$$
\begin{array}{rclcrcl}
\varphi(A, \Phi^{\wedge}_{e_l}(P)) & = & \text{vrai} & \Rightarrow & \varphi(\Phi^{max}(P)) & \leq & e_{l+1} \\
 & & & \Rightarrow & \varphi(\Phi^{lex}(P)) & \leq & \{e_{l+1}^{n_{l+1}}, \ldots, e_k^{n_k}\} \\
 & & & \Rightarrow & \varphi(P) & \leq & \sum_{i=l+1}^{k} n_i.e_i \\
\varphi(A, \Phi^{\wedge}_{e_l}(P)) & = & \text{faux} & \Rightarrow & \varphi(\Phi^{max}(P)) & \leq & e_1 \\
 & & & \Rightarrow & \varphi(\Phi^{lex}(P)) & \leq & \{e_1^{n_1}, \ldots, e_k^{n_k}\} \\
 & & & \Rightarrow & \varphi(P) & \leq & \sum_{i=1}^{k} n_i.e_i \\
\varphi(A, \Phi^{max}_{e_l}(P)) & = & e_j & \Rightarrow & \varphi(\Phi^{lex}(P)) & \leq & \{e_j^{n_j}, \ldots, e_k^{n_k}\} \\
 & & & \Rightarrow & \varphi(P) & \leq & \sum_{i=j}^{k} n_i.e_i \\
\varphi(A, \Phi^{lex}_{e_l}(P)) & = & \{e_1^{m_1}, \ldots, e_l^{m_l}\} & \Rightarrow & \varphi(P) & \leq & \sum_{i=1}^{l} m_i.e_i + \sum_{i=l+1}^{k} n_i.e_i
\end{array}
$$

Compatibilité par agrégation et majorants

Soit A et A', deux instanciations complètes. Si Φ est une fonction de simplification agrégation-compatible, et si la valuation de A dans $\Phi(P)$ est inférieure à la valuation de A' dans $\Phi(P)$ alors le majorant obtenu en évaluant A dans P est inférieur au majorant obtenu en évaluant A' dans P :

$$\Phi \text{ est agrégation-compatible, } \varphi(A, \Phi(P)) \prec' \varphi(A', \Phi(P)) \Rightarrow \varphi(A, P) \prec \varphi(A', P)$$

(preuve : cf. la contraposée 4.7 de l'équation 4.6 dans la preuve du théorème 9)

Pour une fonction de simplification non compatible par agrégation, il est intéressant d'évaluer dans le problème cible toutes les solutions produites lors de la résolution du problème simplifié.

Encadrement de l'optimum

La figure 4.4 résume les possibilités d'encadrer l'optimum d'un problème P, en ayant résolu une simplification $\Phi(P)$ de ce problème.

4.2 Etude expérimentale des simplifications

4.2.1 Etude de la transformation du paysage énergétique par simplification

La figure 4.5 a pour objectif de montrer les transformations opérées sur les valuations des instanciations complètes lors d'une simplification. Elle met en valeur les propriétés d'inclusion des solutions optimales pour une simplification agrégation-compatible et de non-inclusion, dans cet exemple, pour une simplification non agrégation-compatible. De plus, nous pouvons voir quelles

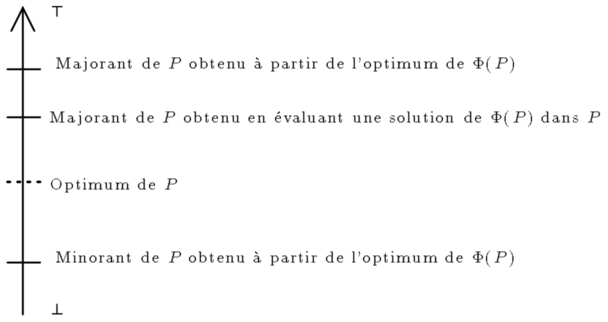


Fig. 4.4 - Encadrement de l'optimum de P par résolution d'une simplification $\Phi(P)$

sont les zones qui contiennent de bonnes instanciations complètes, suivant la simplification employée.

Cette figure 4.5 représente trois **paysages énergétiques** [6] différents correspondant aux valuations de toutes les instanciations complètes (3^{10}) d'un unique problème Σ-VCSP P simplifié en possibiliste $\Phi^{max}(P)$, en lexicographique $\Phi^{lex}(P)$, et sans simplification (laissé en additif). Le problème initial P est un VCSP additif aléatoire : $n = 10, d = 3, c = 80\%, t = 50\%$. La valuation de ses contraintes est choisie aléatoirement suivant une distribution uniforme parmi l'ensemble de valuations $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

En abscisse figure la séquence numérotée des instanciations complètes produites par un parcours en profondeur d'abord sans heuristique, ni coupe. C'est le même ordre qui est utilisé pour évaluer les instanciations dans les trois simplifications. En ordonnée, les valuations sont indiquées sur une unique échelle logarithmique. En possibiliste, les priorités des instanciations complètes vont de 5 à 10 ; en additif, les coûts vont de 16 à 144 ; et en lexicographique, les valuations vont de 896 à 2870850[7]. Sur la figure, les valuations sont marquées par un simple point ; à l'exception des valuations optimales qui sont marquées par un symbole ; par exemple, un losange pour les optima du problème $\Phi^{max}(P)$.

Premièrement, nous remarquons que les optima du problème $\Phi^{max}(P)$ englobent bien les optima de $\Phi^{lex}(P)$, mais aussi ceux de P. En revanche, les optima de $\Phi^{lex}(P)$ n'englobent pas les optima de P. Deuxièmement, nous observons que le problème $\Phi^{lex}(P)$ différencie effectivement mieux les instanciations que $\Phi^{max}(P)$. La classe des lex-VCSP est bien un raffinement de la classe des max-VCSP. Troisièmement, les zones ayant plusieurs bonnes instanciations successives sont semblables pour $\Phi^{lex}(P)$ et P : les bonnes solutions du problème P simplifié en lexicographique sont généralement aussi de bonnes solutions pour le problème P non simplifié.

Nous analysons la répartition des instanciations complètes des problèmes $\Phi^{max}(P)$, $\Phi^{lex}(P)$ et P suivant leurs qualités. La qualité d'une solution de coût ub pour un problème dont l'optimum

[6] Un paysage énergétique est une représentation graphique d'une fonction objectif. En VCSP, cela peut correspondre à visualiser l'ensemble des valuations des affectations complètes triées suivant un ordre donné.

[7] La même conversion que celle présentée dans [SFV95] a été utilisée pour valuer un multi-ensemble de priorité dans l'ensemble des entiers naturels $\mathbb{N}$. A chaque priorité i est associée une valeur de conversion $c_i \in \mathbb{N}$, dépendante de P, telle que $c_i = 1 + \sum_{j=1}^{i-1} \sum_{c \in C / \Phi^{lex} \circ \varphi(c) = \{j\}} c_j$ et $c_1 = 1$. Pour une valuation lexicographique égale au multi-ensemble $\{5, 4, 2, 1^4\}$, la valeur de conversion est égale à $c_5 + c_4 + c_2 + 4 * c_1$.

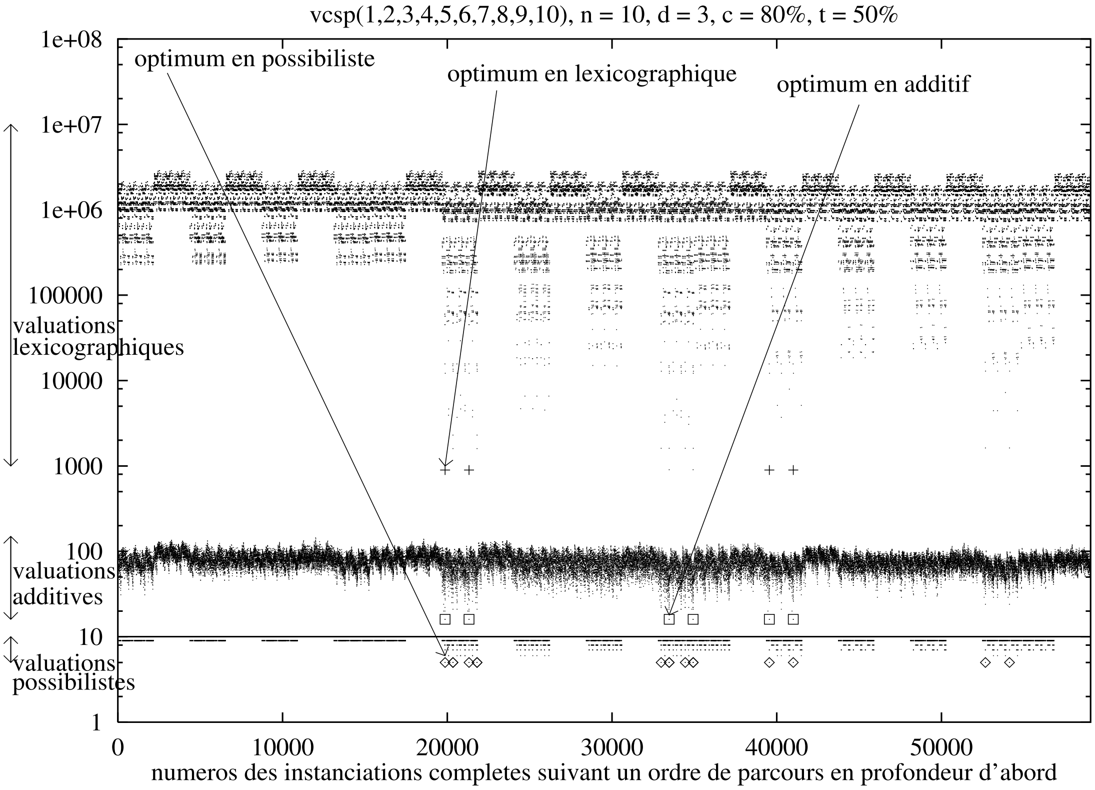


Fig. 4.5 - Transformation des valuations des instanciations complètes par simplification d'un Σ-VCSP aléatoire (10 variables)

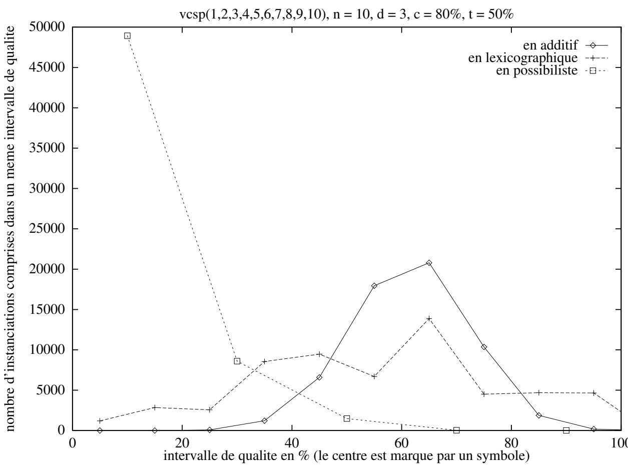


Fig. 4.6 - Histogramme des instanciations complètes suivant leur qualité

vaut *opt* et dont le coût au pire vaut ub_{max} est égale à $\frac{ub_{max}-ub}{ub_{max}-opt}$. Ainsi, la qualité $\rho(A,\Phi(P))$ d'une instanciation complète A dans un problème $\Phi(P)$ est donnée par la formule :

$$\rho(A,\Phi(P)) = \frac{\bigotimes'_{c \in C(\Phi(P))} \Phi \circ \varphi(c) - \varphi(A,\Phi(P))}{\bigotimes'_{c \in C(\Phi(P))} \Phi \circ \varphi(c) - \varphi(\Phi(P))}$$

Cette répartition est donnée sous forme d'un histogramme dans la figure 4.6. L'histogramme est construit en comptant le nombre d'instanciations appartenant à un intervalle de qualité de largeur 10% (sauf pour le problème $\Phi^{max}(P)$ où la largeur est de 20%). Nous voyons sur cet exemple que :

- en possibiliste, les instanciations sont surtout regroupées dans l'intervalle $[0\%, 40\%]$, avec une décroissance exponentielle, les bonnes instanciations étant très rares ;

  cela peut se traduire par une résolution efficace : la majorité des instanciations complètes ont une valuation possibiliste largement supérieure à l'optimum ; donc, beaucoup d'instanciations partielles ne conduisent pas à de bonnes solutions ; ainsi, même un minorant pessimiste de la valuation globale d'une mauvaise instanciation partielle a des chances d'être supérieur à l'optimum, ce qui entraîne une coupe dans l'arbre de recherche si l'algorithme du *Branch and Bound* dispose aussi d'un majorant proche de l'optimum ;

  de plus, dans cet exemple, puisque l'ensemble des solutions optimales en possibiliste est assez réduit et que ces solutions valuées en additif ont aussi un coût faible, l'examen d'une ou de plusieurs solutions optimales du problème simplifié en possibiliste devrait permettre de trouver de bonnes solutions dans le cadre additif ;

- en lexicographique, les instanciations sont réparties assez uniformément sur tous les intervalles de qualité ; cela permet a priori de mieux différencier les valuations globales des instanciations partielles ; cela a pour effet d'accélérer la résolution par rapport à l'additif ;

- en additif, les instanciations sont réparties en cloche, centrée autour de la qualité 60% ; c-à-d que la plupart des instanciations partielles ont une valuation globale en moyenne identique et une distance relative à l'optimum peu élevée ; il faut alors de bons minorants pour arriver à couper suffisamment tôt dans l'arbre de recherche.

Bilan

L'analyse précédente ne vaut que sur un exemple bien précis, de petite taille et dont la valuation des contraintes est choisie parmi dix valuations distinctes. Nous observons des paysages énergétiques assez différents. L'efficacité d'une résolution semble dépendre de deux critères décrivant ces paysages : une bonne différenciation des valuations des instanciations et/ou une majorité d'instanciations de mauvaises qualités qui permettent de réduire l'arbre de recherche. Par ailleurs, bien que les paysages changent, les zones contenant de bonnes solutions sont quasiment les mêmes quelle que soit la simplification.

Nous allons vérifier ces interprétations par une étude plus systématique sur un ensemble de Σ-VCSP aléatoires.

4.2.2 Etude systématique des simplifications sur des Σ-VCSP binaires aléatoires

Nous allons examiner d'une part l'efficacité à couper l'espace de recherche suivant les différentes simplifications ; et d'autre part, la qualité des majorants et des minorants produits par la résolution des simplifications.

Les paramètres qui contrôlent la génération aléatoire des Σ-VCSP binaires sont n, le nombre de variables, d, la taille des domaines identique pour toutes les variables, c, la connectivité du graphe de contraintes et t, la dureté identique pour toutes les contraintes. Enfin, un coût est associé à chaque contrainte de façon aléatoire et uniforme parmi un ensemble Ω de valuations. Ainsi, à chaque valuation de cet ensemble est associé un nombre égal de contraintes (à l'exception de la plus petite valuation qui récupère aussi le reste des contraintes modulo la taille de l'ensemble Ω).

Les expérimentations sont réalisées sur deux séries de Σ-VCSP, notées Pm et Pa, qui diffèrent principalement par la distribution espacée ou resserrée des coûts de Ω :

1. Pm : $n = 15$, $d = 10$, $c \in [10 \cdots 90]$ variant par pas de 10, $t \in [10 \cdots 90]$ variant par pas de 10 et $\Omega = \{1, 10, 100, 1000\}$
2. Pa : $n = 15$, $d = 5$, $c \in [10 \cdots 90]$ variant par pas de 10, $t \in [10 \cdots 90]$ variant par pas de 10 et $\Omega = \{1, 2, 3, 4\}$

Pour chaque combinaison des paramètres c et t, 100 problèmes sont générés. Ainsi, les résultats qui suivent correspondent à des moyennes sur cent problèmes. De façon à équilibrer les temps de résolution entre Pm et Pa, nous avons choisi une taille de domaine plus petite pour Pa, que pour Pm.

4.2.2.1 Résolution accélérée par les simplifications

L'efficacité à couper l'espace de recherche est exprimée en relatif (resp. dans l'absolu) par le gain (resp. la différence) en terme de nombre de noeuds développés pour la résolution des problèmes simplifiés par Φ^{lex} et Φ_e par rapport à la résolution dans le cadre additif. L'étude n'est pas réalisée pour les simplifications Φ^{max} et $\Phi^{\wedge}$. Le lecteur peut trouver dans [SFV95] une étude expérimentale montrant la difficulté croissante de résolution d'un ensemble de problèmes traités en CSP, puis en *max*-VCSP, puis en *lex*-VCSP. La section 2.4.1 présente aussi une comparaison entre une résolution en satisfaction et en optimisation additive.

L'algorithme de résolution employé est un *Branch and Bound* avec *forward checking* valué et des heuristiques dynamiques simples de choix des variables et des valeurs (cf. 2.3.1). Sur ces problèmes, le gain en terme de temps CPU mis pour la résolution est équivalent au gain en terme de nombre de noeuds. Le fait d'avoir plus ou moins de contraintes à gérer ne change pas beaucoup les performances de l'algorithme (seulement d'un facteur constant).

Gains observés sur les relaxations par niveau

La figure 4.7 montre le nombre moyen de noeuds développés en fonction de la dureté des contraintes et suivant le niveau de relaxation, pour les deux séries de problèmes Pm et Pa. Par exemple, le niveau 100 pour Pm signifie que seules les contraintes de coût 100 et 1000 sont considérés lors de la résolution. De même, le niveau 1 pour Pm ou Pa signifie que toutes les contraintes sont prises en compte (aucune simplification n'est réalisée). Nous voyons que la

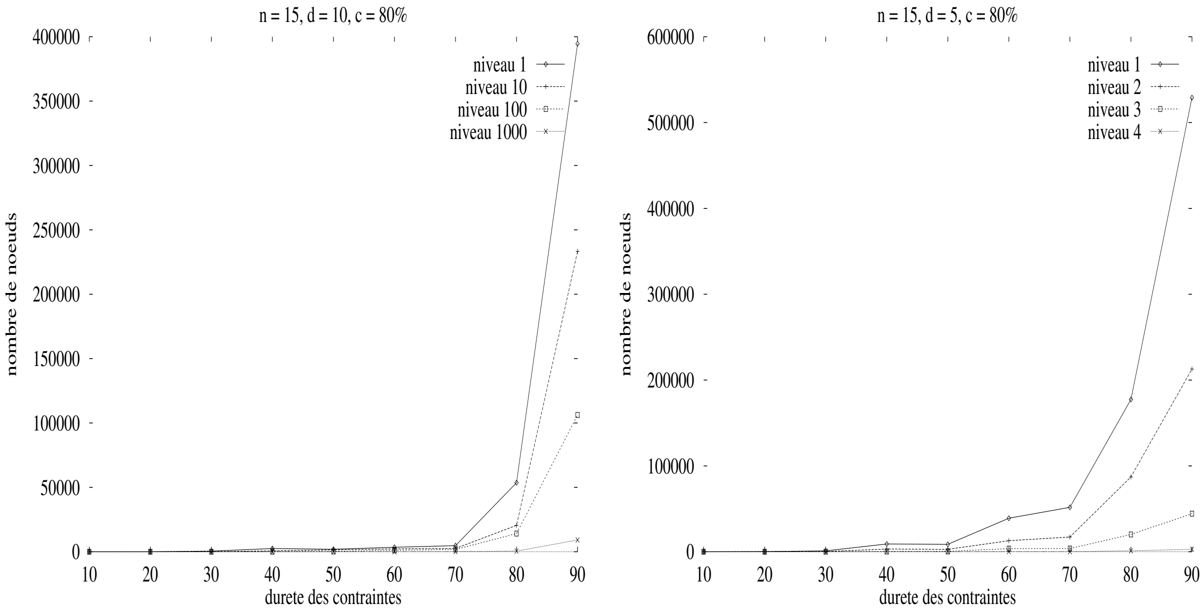


Fig. 4.7 - Simplification de la résolution (en terme de nombre moyen de noeuds) par relaxation des contraintes par niveau pour Pm et Pa (connectivité de 80%)

réduction du nombre de contraintes entraîne une simplification importante de la résolution. Si nous examinons plus en détail ces résultats, nous pouvons noter que la relaxation par niveau engendre une diminution de la connectivité du graphe des contraintes. Pour les courbes de la figure 4.7, la connectivité des problèmes Pm (resp. Pa), au départ égale à 80% au niveau 1, passe à 60% pour le niveau 10 (resp. 2), puis à 40% pour le niveau 100 (resp. 3) et enfin à 20% pour le niveau 1000 (resp. 4).

La figure 4.8 généralise ces observations. Elle montre l'évolution de la complexité (en nombre moyen de noeuds) en fonction de la dureté et de la connectivité pour résoudre les problèmes Pa sans simplifications. L'évolution est similaire pour les problèmes Pm. En optimisation sous contraintes, la complexité des résolutions (trouver l'optimum et prouver l'optimalité) augmente avec la dureté des contraintes et aussi avec la connectivité , du moins pour un algorithme *Branch and Bound* avec simple *forward checking*.

Pour mieux comprendre quel gain offre les relaxations par niveau, nous étudions l'évolution du gain moyen $\frac{1}{N}\sum_{p \in Px} \frac{\text{nombre de noeuds pour résoudre } p}{\text{nombre de noeuds pour résoudre } \Phi_e(p)}$ en fonction de la dureté et de la connectivité. Le gain moyen est la moyenne des gains observés sur chaque instance p d'une série de N problèmes Px (Pm ou Pa). Les expérimentations montrent une très forte variation sur le nombre de noeuds développés pour des problèmes appartenant à une même série. Cette variance conduit à une moyenne peu fiable, même si elle est calculée sur 100 problèmes. C'est pourquoi nous préférons utiliser la **médiane** [8] pour représenter le gain, dans les figures 4.9 et 4.10.

Les courbes de la figure 4.9 montrent l'évolution du gain médian sur des problèmes Pm, en fonc-

[8] La médiane d'une liste triée de N nombres correspond au nombre à la position $N/2$ dans la liste.

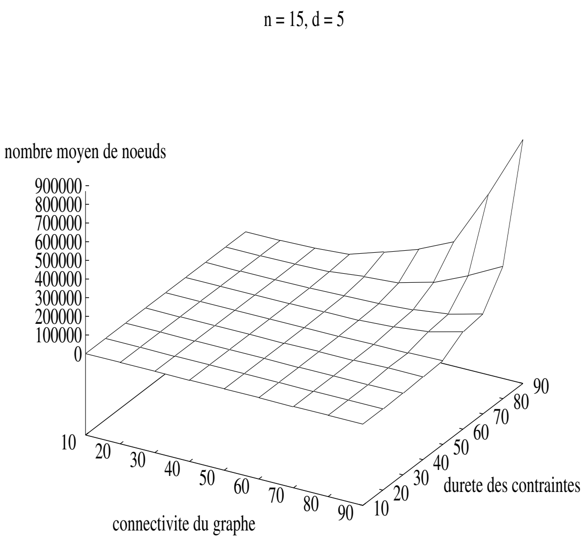
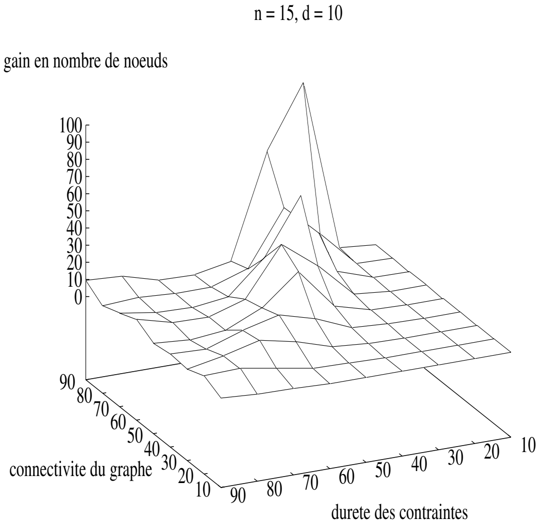


Fig. 4.8 - Complexité de la résolution des problèmes Pa (en nombre moyen de noeuds)



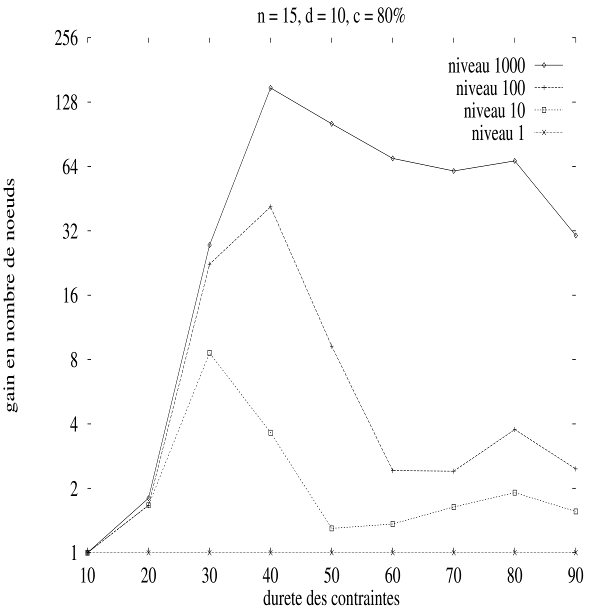


Fig. 4.9 - Gain en nombre de noeuds entre $\Phi_e(Pm)$ et Pm ($e = 100$ pour la figure de gauche)

tion du niveau de relaxation ainsi que de la dureté et de la connectivité. La courbe de gauche fixe le niveau de relaxation à 100 et fait varier la dureté et la connectivité. Les courbes de droite fixent la connectivité à 80% et font varier la dureté et le niveau de relaxation. La courbe de gauche montre une zone de transition ("arête dorsale"), en relation avec la frontière entre les problèmes cohérents (de coût nul) et les problèmes incohérents (de coût supérieur à zéro). En effet, lorsque les problèmes sont fortement cohérents, il est très facile de trouver une solution optimale, d'où un gain égal à 1 pour les duretés proches de 0%. La relaxation par niveau a pour effet de diminuer la connectivité et donc de décaler la frontière entre cohérence et incohérence vers des duretés plus fortes. Par exemple, pour une connectivité fixée à 80%, en appliquant la formule donnée dans la section 2.4.1, nous déterminons une frontière à la dureté 33% ; au niveau 10, cela se passe à la dureté 42%, puis 56% au niveau 100 et 80% au niveau 1000. C'est pourquoi le gain augmente fortement dès que le problème devient proche de la frontière et même au delà, tandis que le problème simplifié est encore cohérent. Ensuite, le gain se stabilise ; il reste relativement uniforme pour une dureté ou une connectivité croissante. Il est intéressant de noter que ce gain peut être très élevé si la simplification fait diminuer fortement la connectivité : gain supérieur à 64 pour $\Phi_{1000}(Pm)$ (l'échelle des gains est logarithmique). En revanche, la simplification $\Phi_{10}(Pm)$ ne procure qu'un gain inférieur à 2 pour la moitié des problèmes Pm testés.

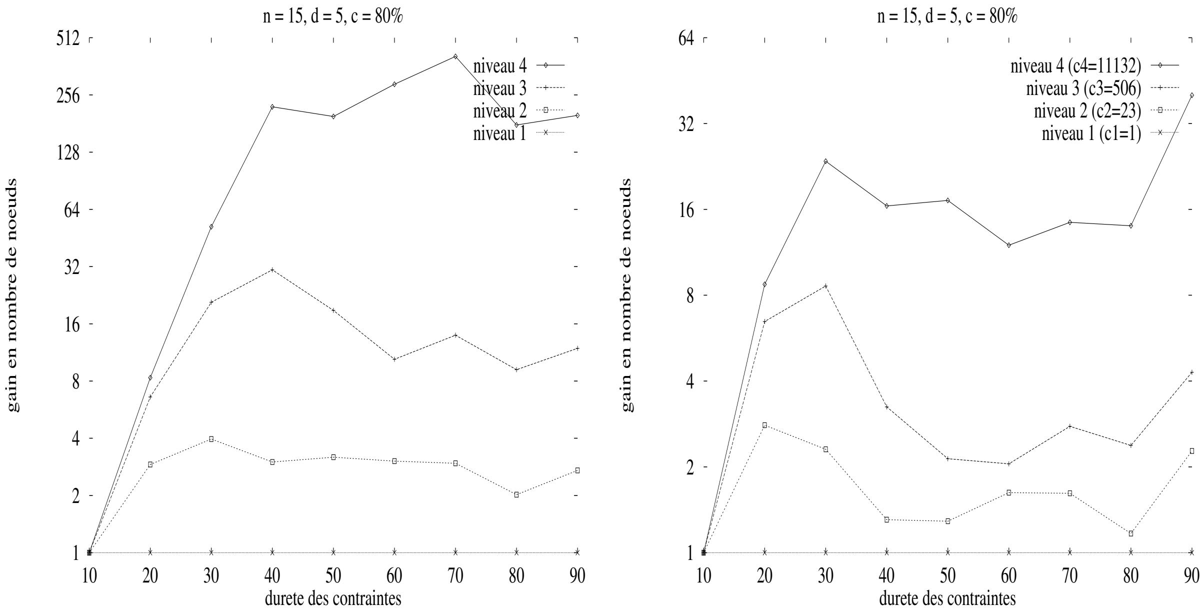


Fig. 4.10 - Gain en nombre de noeuds entre $\Phi_e(Pa)$ et Pa ; gain en nombre de noeuds entre $\Phi_e^{lex}(Pa)$ et $\Phi^{lex}(Pa)$

La figure 4.10 (de gauche) montre que pour les problèmes Pa, nous observons le même type de courbes que celles de la figure 4.9 ; à l'exception du gain qui ne rediminue pas autant après la frontière entre cohérence et incohérence. Il est aussi intéressant d'étudier l'efficacité de la relaxation par niveau lorsque le problème est traité en lexicographique. L'application de la simplification lexicographique a pour effet d'espacer les poids des contraintes (cf. la section 4.2.1 pour le calcul des nouveaux poids c_i). Les courbes de droite permettent de comparer, par rapport aux courbes de gauche, l'évolution du gain lorsque la résolution est faite en lexicographique plutôt qu'en additif. Nous remarquons que les gains sont moins importants lorsque les poids sont plus espacés.

Remarque 27 La relaxation des contraintes peut entraîner une séparation des variables en plusieurs **composantes connexes** [9] disjointes aux quelles est associé un sous-graphe distinct des contraintes. Une détection des composantes connexes, après relaxation par niveau, permet d'accélérer la résolution globale en la remplaçant par une succession de résolutions séparées plus courtes. Ceci n'est effectif sur les problèmes aléatoires que lorsque la connectivité initiale est faible. ◇

Gain observé sur la simplification en lexicographique

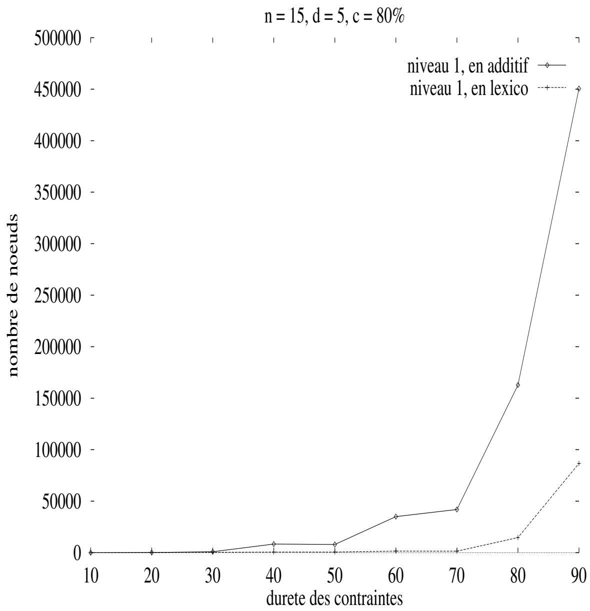


Fig. 4.11 - Simplification de la résolution (en terme de nombre médian de noeuds) par résolution des problèmes Pa en additif et en lexicographique (connectivité de 80%)

Le changement de la structure de valuation additive vers une structure lexicographique simplifie de façon importante la résolution lorsque les coûts des contraintes sont nombreux et peu espacés. Ainsi, pour les problèmes Pm, il n'y a en moyenne pas de gain en nombre de noeuds entre une résolution additive et lexicographique, quelle que soit la valeur des paramètres *connectivité* et *dureté* (gain inférieur à 1.18 pour $c = 80\%$ et $t = 80\%$). En revanche, pour les problèmes Pa, le gain est important. La figure 4.11 montre le nombre médian de noeuds développés pour résoudre Pa sans relaxation des contraintes (équivalent au niveau 1) suivant les deux types de structure de valuation.

La figure 4.12 de gauche montre l'évolution du gain médian en fonction de la dureté et de la connectivité pour les problèmes Pa complets (pris au niveau 1). L'allure générale de la surface est

[9] Une composante connexe est un groupe de variables, dont chaque variable est reliée à toutes les autres par un chemin dans le graphe des contraintes associées à ces variables. Les composantes sont disjointes deux à deux, au sens où aucune variable d'un groupe n'est en relation de contrainte avec une variable de l'autre groupe. Il existe un algorithme polynomial, en $O(n)$, pour la détection des composantes connexes, voir par exemple [Tsa93].

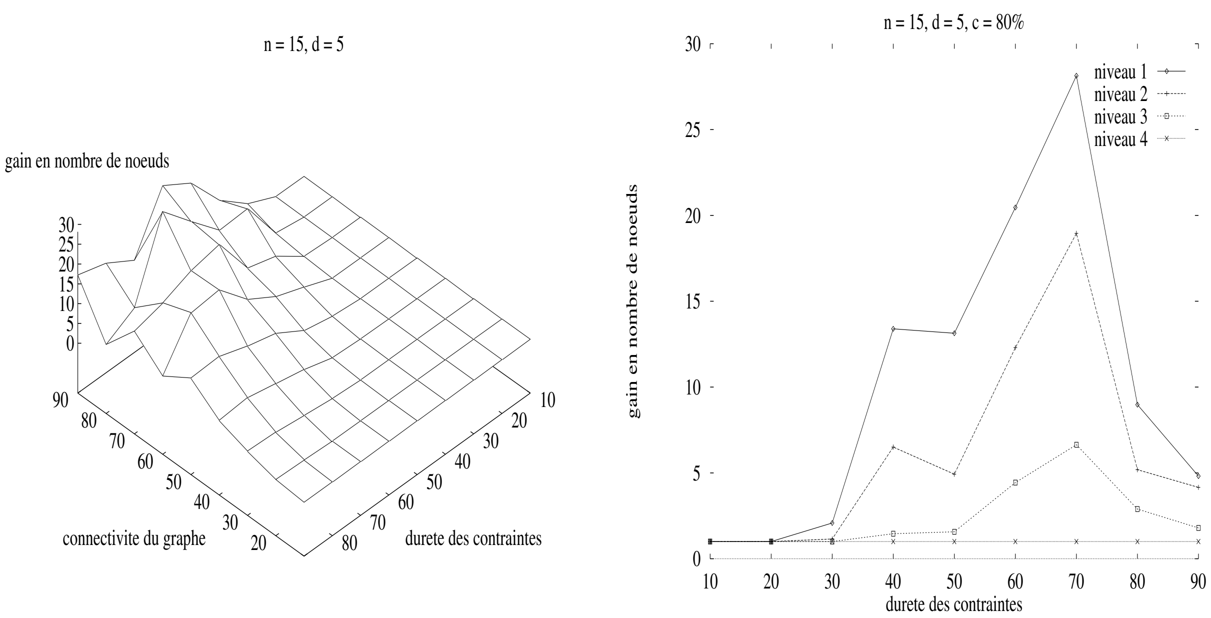


Fig. 4.12 - Gain en nombre de noeuds entre $\Phi_e^{lex}(Pa)$ et $\Phi_e(Pa)$ ($e = 1$ pour la figure de gauche)

grossièrement une cloche : à partir de la frontière où les problèmes deviennent incohérents, le gain croît lorsque les paramètres c et t augmentent ; puis rediminue lorsque la dureté dépasse un seuil. En fait la surface est assez accidentée, d'une part à cause de l'écart-type important sur les résultats et d'autre part du fait de l'existence de plusieurs seuils. Ces seuils de dureté semblent reliés aux frontières de l'incohérence des différentes hiérarchies de contraintes. En effet, pour les problèmes Pa à la connectivité 80%, les contraintes dont le coût est égal à 4 (resp. supérieur à 3, supérieur à 2 et supérieur à 1) sont toutes satisfiables en dessous de la dureté 68% (resp. 44%, 32% et 25%). L'algorithme du *Branch and Bound* peut couper plus efficacement lorsqu'un sous-ensemble des contraintes est assimilé à des contraintes dures, en cours de recherche. C'est le cas en lexicographique, à $c = 80\%, t < 68\%$, une solution violant aucune contrainte de priorité 4 peut être trouvée. Dans ce cas, le majorant servant de coupe interdit toute nouvelle instanciation partielle qui violerait une ou plusieurs contraintes de priorité 4. Ce n'est pas le cas en additif.

Puisque nous voulons combiner le changement de structure de valuation avec la relaxation des contraintes par niveau, nous étudions le gain du passage en lexicographique lorsque des contraintes ont déjà été relaxées (le gain de la relaxation par niveau n'est pas pris en compte). La figure 4.12 de droite montre que le gain médian en nombre de noeuds pour résoudre $\Phi_e^{lex}(Pa)$ par rapport à $\Phi_e(Pa)$ diminue lorsque le niveau de relaxation des contraintes augmente[10]. En revanche, nous observons les mêmes seuils de dureté qui influent sur l'évolution du gain.

Bilan

Les simplifications par relaxation des contraintes par niveau et par changement de la structure de valuation accélèrent de façon notable la résolution quelle que soit la valeur des paramètres de connectivité et de dureté. Les principales observations sont :

- accélération plus importante si la relaxation par niveau fait chuter fortement la connectivité

[10] Au niveau 4, le gain est nul car la résolution en lexicographique est identique à celle en additif.

(gains supérieurs à 100 sur $\Phi_{1000}(Pm)$ et $\Phi_4(Pa)$) ou rend un problème satisfiable ("arête dorsale" de la figure 4.9), ou si le passage en lexicographique revient à écarter fortement les coûts des contraintes (gains supérieurs à 10 sur $\Phi^{lex}(Pa)$) ;

- à l'inverse des relaxations par niveau, le gain du passage en lexicographique diminue lorsque le nombre de hiérarchies de contraintes prises en compte dans la résolution diminue ; ces deux techniques de simplification sont donc bien complémentaires ;
- la combinaison des deux techniques de simplification produit des simplifications encore plus efficaces (cf. figures de droite 4.10 et 4.12).

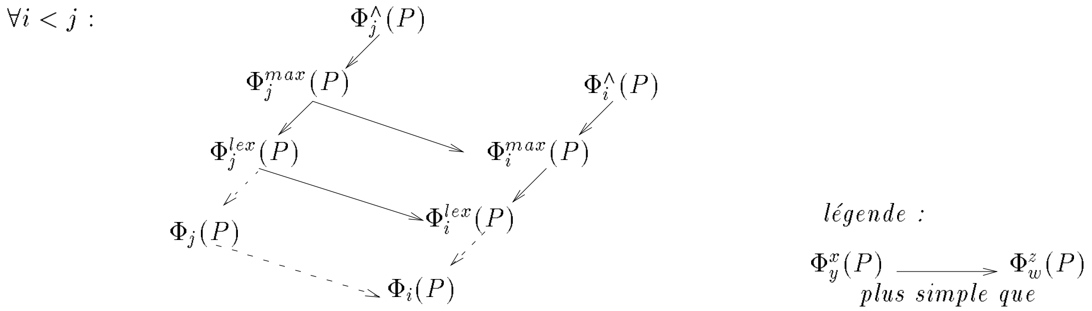


Fig. 4.13 - Ordre de complexité moyenne observé expérimentalement sur les simplifications

Ces expérimentations sur le comportement moyen des simplifications appliquées à des problèmes aléatoires permettent d'étendre l'ordre partiel prouvé dans la section 4.1.2.2 (cf. figure 4.3). Les extensions sont : l'opérateur additif est généralement plus dur que les autres opérateurs d'agrégation ; le fait de retirer des contraintes par une relaxation par niveau en additif accélère généralement la résolution. Le nouvel ordre partiel est résumé par la figure 4.13.

Remarque 28 La relaxation des contraintes par niveau accélère généralement la résolution en optimisation. En revanche, ce n'est pas le cas en satisfaction. En effet, les problèmes de satisfaction de contraintes sont plus durs à résoudre lorsqu'ils sont à la frontière entre cohérence et incohérence (cf. section 2.4.1). Le fait de retirer des contraintes à un problème facile peut le rapprocher de cette frontière des problèmes difficiles.

◇

4.2.2.2 Qualité des majorants et des minorants obtenus à partir des solutions optimales des simplifications

Qualité des majorants

L'évaluation dans un problème initial d'une solution optimale pour une simplification de ce problème fournit un majorant. La qualité de ce majorant peut être mesurée et normée par rapport à l'optimum du problème initial. Soit $P = (V, D, C, S, \varphi)$, un problème cible additif et $\omega = \sum_{c \in C} \varphi(c)$, la valuation associée au pire cas dans P. Soit A, une solution d'une simplification $\Phi(P)$. La qualité normée exacte (en pourcentage) (cf. équation 3.2) du majorant, déduite de A dans P, est notée $\rho_{ub}(A, P)$:

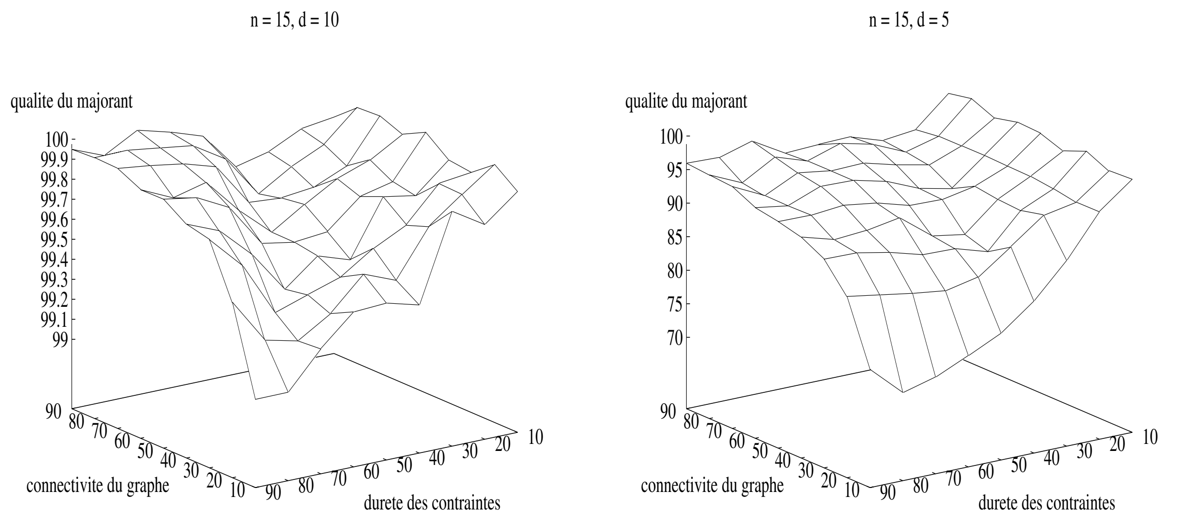


Fig. 4.14 - Qualité des majorants obtenus par résolution de $\Phi_{100}(Pm)$ et $\Phi_3(Pa)$

$$\rho_{ub}(A,P) = 100 \cdot \rho(A,P) = 100 \cdot \frac{\omega - \varphi(A,P)}{\omega - \varphi(P)} \tag{4.10}$$

Les majorants déduits des simplifications par relaxation des contraintes sont de bonne qualité pour Pm (par ex., $\rho_{ub} > 99\%$ pour $\Phi_{100}(Pm)$) et Pa (par ex., $\rho_{ub} \approx 92\%$ pour $\Phi_3(Pa)$). La figure 4.14 montre l'évolution de la qualité moyenne $\bar{\rho}_{ub}$ pour des majorants obtenus à partir des résolutions de $\Phi_{100}(Pm)$ et de $\Phi_3(Pa)$. $\bar{\rho}_{ub}$ correspond à la moyenne des qualités ρ_{ub} de 100 instances.

En moyenne sur des problèmes aléatoires, les contraintes relaxées (niveaux inférieurs à e) seront violées aléatoirement suivant leur degré de dureté t : $\varphi(A,P) \approx \Psi_e(\varphi(A,\Phi_e(P))) + t\sum_{c\in C/\varphi(c)<e}\varphi(c)$. La qualité $\rho_{ub}(A,P)$ diminue lorsque ce terme augmente par rapport au coût optimal $\varphi(P)$. Cela semble se produire dans la zone des problèmes simplifiés à la frontière entre cohérence et incohérence. Cette zone correspond à l'emplacement de l'"arête dorsale" de la figure 4.9. De plus la qualité des majorants a aussi tendance à diminuer lorsque la connectivité diminue. Une explication repose sur l'hypothèse que les problèmes aléatoires faciles ont plus de solutions optimales que les problèmes durs. Sur les problèmes relativement faciles, la relaxation des contraintes produit des problèmes simplifiés quasi-satisfiables (dont l'optimum est proche de zéro). Ces problèmes simplifiés ont a priori beaucoup de solutions optimales. Plus ils contiennent de solutions optimales, plus la probabilité que ces solutions soient aussi optimales pour les problèmes non simplifiés est faible[11].

Les majorants déduits des simplifications en lexicographique sont de bonne qualité si la distribution des coûts des contraintes est espacée ou si les problèmes sont presque satisfiables. Pour les 8100 instances de la classe de problème Pm, résolus en lexicographique, seulement deux instances ($c = 90\%, t = 60\%$ et $c = 90\%, t = 70\%$) ont une solution optimale dans $\Phi^{lex}(Pm)$ qui

[11] à l'exception des problèmes très simples, lorsque la dureté des contraintes est proche de zéro (à $t = 10\%$ sur la figure 4.14).

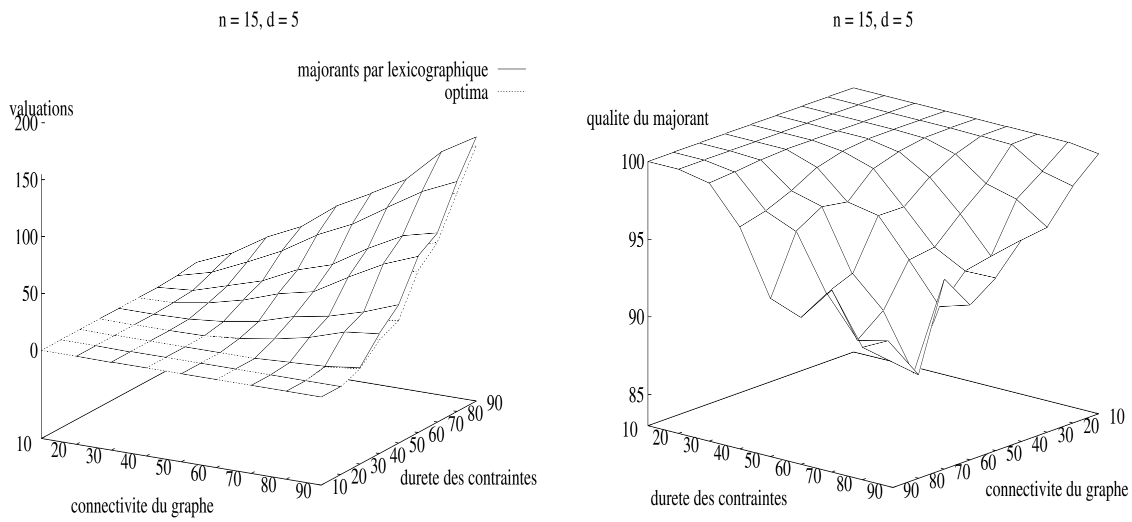


Fig. 4.15 - Comparaison, en absolu et en relatif, des majorants obtenus par la résolution des simplifications $\Phi^{lex}(Pa)$ par rapport aux optima des Pa

n'est pas optimale dans Pm. Ainsi, la qualité $\rho_{ub}(A, Pm)$ est égale à 100%.

En revanche, pour les problèmes Pa, la distance entre les majorants déduits des simplifications et les optima de Pa augmente avec la dureté et la connectivité. La figure 4.15 de gauche montre l'évolution des valuations moyennes des majorants et des optima suivant t et c. L'écart entre ces deux surfaces ne dépasse pas en moyenne une valuation de 15. Cela se traduit par une qualité $\rho_{ub}(A, Pa)$ variant entre 90% et 100%, comme le montre la figure 4.15 de droite. Celle-ci représente une surface ayant une légère forme de cuvette : la qualité ré-augmente lorsque la dureté ou la connectivité s'approchent de 100%.

Qualité des minorants

Soit $P = (V, D, C, S, \varphi)$, un problème cible additif. Soit $\varphi(\Phi(P))$, l'optimum d'une simplification $\Phi(P)$. Soit Ψ, la fonction de transfert associée à Φ. La qualité normée exacte (en pourcentage) du minorant déduite de A dans P est notée $\rho_{lb}(\varphi(\Phi(P)))$:

$$\rho_{lb}(\varphi(\Phi(P))) = 100 \cdot \frac{\Psi(\varphi(\Phi(P)))}{\varphi(P)} \tag{4.11}$$

La qualité des minorants diminue lorsque le niveau de relaxation augmente ou que la distribution des coûts des contraintes se resserre. La figure 4.16 montre l'évolution de la qualité des minorants obtenus par la résolution des problèmes relaxés $\Phi_{100}(Pm)$ et $\Phi_3(Pa)$ en fonction de la dureté et de la connectivité. Pour les problèmes satisfiables, la qualité vaut 100% par défaut. Pour les problèmes proches de la satisfaction (optimum > 0) tels que leurs simplifications soient satisfiables, la qualité est nulle d'après la formule 4.11, d'où la transition brutale observée. Ensuite la qualité augmente régulièrement jusqu'à une certaine limite (environ 95% pour $\Phi_{100}(Pm)$ et 60% pour $\Phi_3(Pa)$).

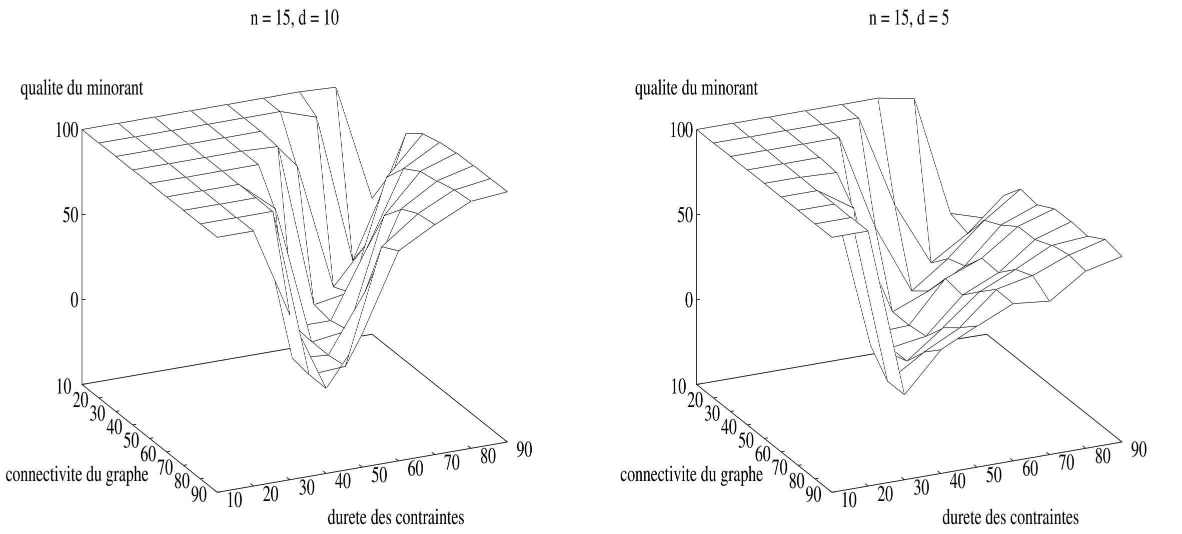


Fig. 4.16 - Qualité des minorants obtenus par résolution de $\Phi_{100}(Pm)$ et $\Phi_3(Pa)$

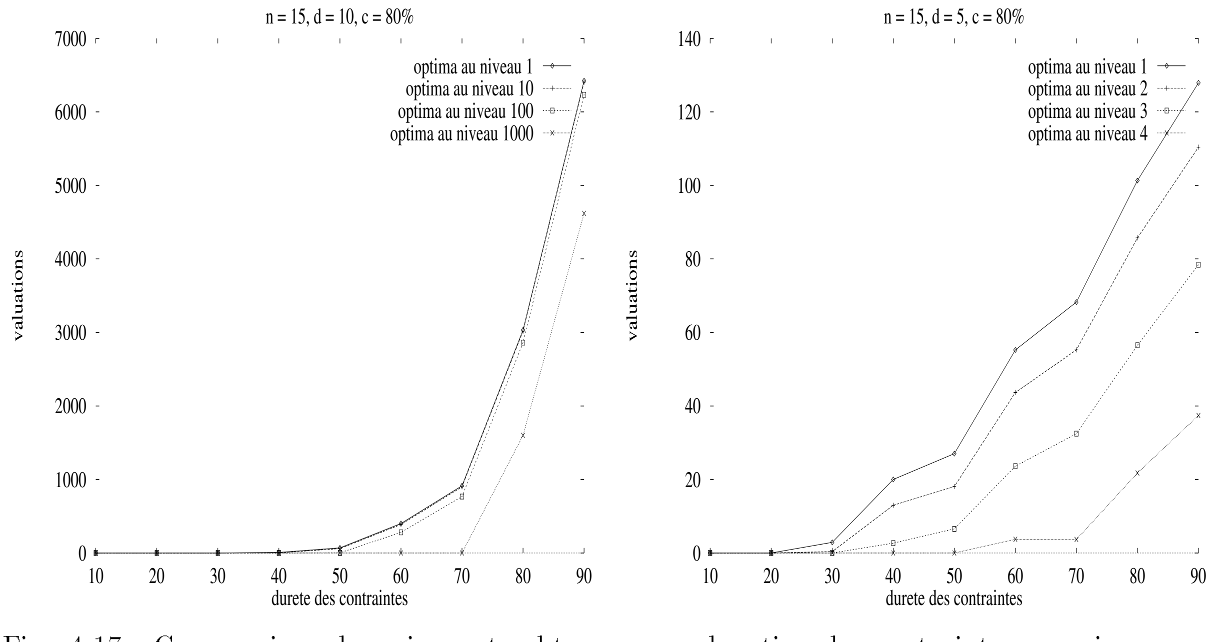


Fig. 4.17 - Comparaison des minorants obtenus par relaxation des contraintes par niveau sur les problèmes Pm et Pa à la connectivité $c = 80\%$

La figure 4.17 montre la valuation moyenne des minorants obtenus par résolution des relaxations par niveau des problèmes Pm et Pa à la connectivité $c = 80\%$. Nous observons que les simplifications $\Phi_{10}(Pm)$ et $\Phi_{100}(Pm)$ procurent des minorants très proches des optima de Pm. En revanche $\Phi_{1000}(Pm)$ n'offre pas de bons minorants, sauf lorsque la dureté est élevée : à $t = 100\%$, la qualité limite $\rho_{lb}(\varphi(\Phi_e(Pm))) \approx 90\%$. Pour les problèmes Pa, la qualité des minorants se dégrade plus vite que pour les problèmes Pm, comme le montre aussi la figure 4.16.

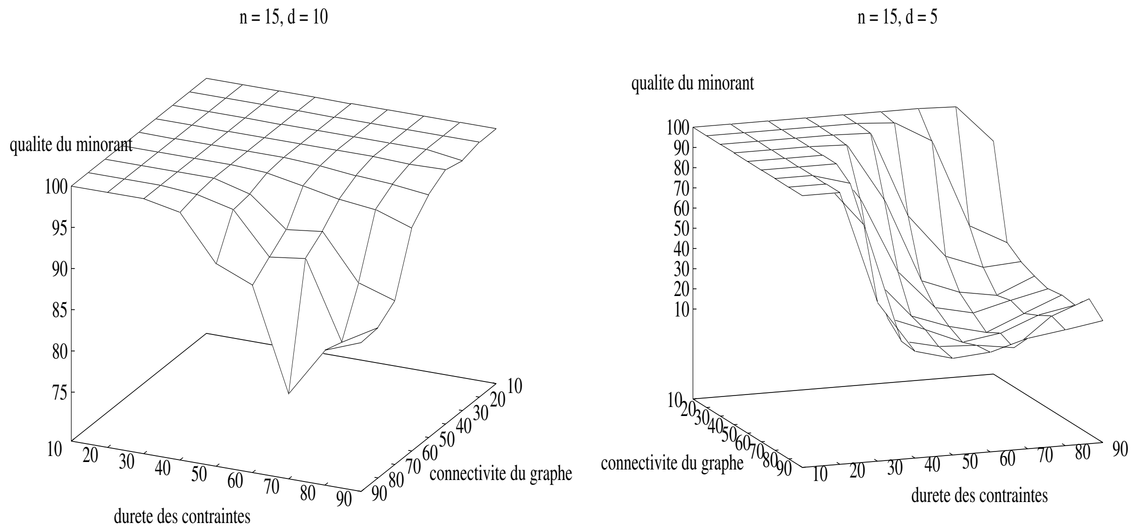


Fig. 4.18 - Qualité des minorants de Pm (resp. Pa) obtenus à partir des optima de $\Phi^{lex}(Pm)$ (resp. $\Phi^{lex}(Pa)$)

La figure 4.18 montre l'évolution de la qualité moyenne $\bar{\rho}_{lb}$ pour des minorants obtenus à partir des résolutions en lexicographique de Pm et de Pa. $\bar{\rho}_{lb}$ correspond à la moyenne des qualités ρ_{lb} de 100 instances. Pour les problèmes Pm, cette qualité vaut 100% tant que les problèmes ont des solutions satisfaisant toutes les contraintes de coût supérieur à 1. Puis elle diminue avec des problèmes plus durs, en restant supérieure à 85% (sauf pour $c = 90\%, t = 70\%$ où $\rho_{lb} = 79\%$). Pour les problèmes Pa, la qualité chute brutalement dès que les problèmes deviennent insatisfiables[12]. La surface décrit alors une large cuvette ; la qualité varie dedans entre 10% et 30%.

Bilan

La conclusion de cette étude de la qualité des majorants et des minorants est séparée en deux, suivant la distribution de coût des contraintes :

- problèmes Pm : la qualité des majorants et minorants est bonne quel que soit le type de simplification (à l'exception des relaxations au niveau 1000) ;
- problèmes Pa : la qualité des majorants diminue sensiblement surtout pour les simplifications par relaxation, mais reste correcte pour la simplification en lexicographique ; en

[12] Par défaut, lorsque l'optimum est égal à zéro, la qualité ρ_{lb} vaut 100%.

revanche la qualité des minorants fournis par la simplification en lexicographique est très médiocre, de même que des simplifications avec un niveau de relaxation élevé.

De façon générale, la qualité des majorants et des minorants semble meilleure pour des problèmes très faciles ($t \mapsto 0\%$, $c \mapsto 0\%$) ou très durs ($t \mapsto 100\%$, $c \mapsto 100\%$) que pour des problèmes dans la zone de difficulté intermédiaire[13].

Finalement, nous apercevons que la production d'un bon encadrement de l'optimum dans un temps plus court que celui pris par une méthode complète appliquée directement sur le problème cible est un problème délicat. Le gain en temps de résolution, grâce à une simplification se paye par une perte en terme de qualité de l'encadrement. Ce compromis est à la base de l'approche Anytime. Il n'a de sens que si la résolution globale d'un problème fait évoluer progressivement cet encadrement au cours du temps. C'est l'objet de la section suivante. Par ailleurs, nous constatons sur ces résultats expérimentaux qu'il est plus facile d'encadrer par le haut (production de bons majorants) que par le bas (production de bons minorants).

4.3 Enchaînement des simplifications

Dans un contexte interruptible où le temps accordé à la résolution est limité et variable, l'état d'avancement de la résolution est donné par l'encadrement de l'optimum du problème à résoudre. Cette information progresse en partie par une production de minorants croissants de l'optimum. L'amélioration progressive des minorants est obtenue en enchaînant des simplifications de moins en moins grossières mais de plus en plus dures à résoudre. L'efficacité de cet enchaînement est amélioré en récupérant le maximum d'informations des premières simplifications résolues pour accélérer les suivantes. Nous rappelons que l'objectif est d'encadrer de façon progressive et interruptible l'optimum des **problèmes additifs** (cf. section 3.2.2.1).

4.3.1 Description générale de l'enchaînement des simplifications

L'ensemble des simplifications utilisées est constitué à partir d'une combinaison des simplifications par relaxation des contraintes suivant un seuil e (e est un élément de l'ensemble des valuations de P), notées Φ_e , et des simplifications par changement de la structure de valuation, notées $\Phi^{\otimes}$ ($\otimes \in \{\wedge, max, lex, \sum\}$). La combinaison de ces deux simplifications, notée $\Phi_e^{\otimes}$, permet d'enrichir le nombre potentiel de résolutions qui offrent différents compromis entre la qualité du minorant produit et le temps de résolution.

L'enchaînement des simplifications suit l'ordre de complexité globalement croissant, partiellement prouvé en section 4.1.2 et observé expérimentalement en section 4.2 (cf. figure 4.13).

Nous reprenons les mêmes notations que dans la section 4.1.3.2 pour caractériser un problème cible, noté P : soit $(e_1, \ldots, e_k)$, la liste triée des valuations décroissantes associées aux contraintes de P. L'enchaînement général des simplifications est alors :

- changements de structure de valuation suivant l'ordre $\wedge$-VCSP $\prec$ max-VCSP $\prec$ lex-VCSP $\prec$ Σ-VCSP ;

[13] Cette observation se vérifie également pour la complexité moyenne de certains algorithmes de résolution. Le calcul du minorant d'une instanciation partielle à l'aide des Directed Arc-consistency Counts permet de faire rediminuer le temps de résolution lorsque la dureté s'approche de 100% (cf. section 2.4.1, figure 2.20).

- pour chaque structure de valuation, résolution des relaxations au niveau e_1 (contenant uniquement les contraintes de plus forte valuation) jusqu'au niveau e_k (incluant toutes les contraintes) ; à l'exception de la structure de valuation en satisfaction ($\otimes = \wedge$) qui est directement résolue sans relaxation (au niveau e_k).

Chaque résolution d'une simplification par une méthode complète produit des majorants à partir des solutions complètes rencontrées (cf. section 4.1.4) et un minorant à partir de l'optimum du problème simplifié[14]. Le théorème 13 et son application à la composition de simplifications (cf. fin de la section 4.1.3.1) définissent les fonctions de transfert $\Psi_e^{\otimes}$ permettant de déduire les minorants. La valuation du minorant déduit de l'optimum d'un problème simplifié $\Phi_e^{\otimes}(P)$ vaut $\Psi_e^{\otimes}(\varphi(\Phi_e^{\otimes}(P)))$. À tout moment, l'encadrement courant du problème cible est donné par le meilleur majorant trouvé et par le dernier minorant produit. L'enchaînement assure une évolution croissante des minorants.

Plus précisément, lors de l'enchaînement des simplifications, les fonctions de transfert sont aussi utilisées pour déduire des minorants sur les simplifications à venir. Ainsi, la solution optimale de la dernière simplification complètement résolue produit un minorant et aussi un majorant sur les simplifications suivantes. Pour les simplifications où le minorant déduit est égal au majorant, nous pouvons en déduire que ces futures simplifications sont inutiles car nous connaissons déjà leur optima. La description suivante explique les déductions possibles suivant la structure de valuation de la simplification et précise l'enchaînement effectivement utilisé dans nos expérimentations :

- en $\wedge$-VCSP, résolution directement au niveau e_k le plus faible ; ainsi, si le problème est trivialement satisfiable, cela est détecté au plus tôt et l'enchaînement est immédiatement terminé ; sinon, la simplification $\Phi_{e_k}^{max}$ est détectée inutile car son optimum sera connu au plus tard une fois $\Phi_{e_{k-1}}^{max}(P)$ résolu ;

- en max-VCSP, résolution par niveau, à chaque fois sous la forme d'un CSP classique, du premier niveau e_1 à l'avant dernier niveau e_{k-1}, jusqu'à ce que l'incohérence soit rencontrée ; de cette façon, de bons majorants du problème cible (respectant toutes les contraintes de plus forte priorité) peuvent être rapidement trouvés pour les niveaux cohérents ; de même, tant que le niveau e_l est cohérent, $\varphi(\Phi_{e_l}^{max}(P)) = 0$, nous avons $\varphi(\Phi_{e_l}^{lex}(P)) = \emptyset$ et $\varphi(\Phi_{e_l}^{\Sigma}(P)) = 0$; donc $\Phi_{e_l}^{lex}$ et $\Phi_{e_l}^{\Sigma}$ deviennent inutiles ; sinon, dès que le niveau e_i est prouvé incohérent, alors $\forall j \geq i, j \leq k, \varphi(\Phi_{e_j}^{max}(P)) = e_i$; donc les résolutions aux niveaux suivants deviennent inutiles ;

- en lex-VCSP, résolution par niveau, du niveau où la résolution en max-VCSP s'est arrêtée jusqu'au niveau inférieur tel que l'on est certain que la résolution des niveaux suivants n'améliorera pas le minorant du problème cible ; c'est le cas lorsque la valuation minimale retournée par la fonction $lex()$ (cf. théorème 13) est inférieure à la valuation en additif de la solution lexicographique optimale, notée $\{e_1^{m_1}, \ldots, e_i^{m_i}\}$, du niveau e_i :

$$\Psi_{e_i}^{lex}(\varphi(\Phi_{e_i}^{lex}(P))) < \sum_{l=1}^{i} m_l e_l$$

  dans ce cas, les simplifications $\Phi_{e_j}^{lex}$, avec $j > i$, deviennent inutiles vis à vis de l'amélioration du minorant ; auparavant, tant que $\Psi_{e_l}^{lex}(\varphi(\Phi_{e_l}^{lex}(P))) = \sum_{u=1}^{l} m_u e_u$, nous pouvons en dé-

[14] Il peut y avoir éventuellement plusieurs minorants si la méthode n'est pas simplement un *Depth First Branch and Bound* et produit plusieurs minorants du problème simplifié durant la résolution (cf. section 6.2).

duire les optima des simplifications Φ_{e_l} égaux à $\sum_{u=1}^{l} m_u e_u$; ces simplifications ne seront pas résolues ;

- en Σ-VCSP, résolution par niveau, du niveau où la résolution en *lex*-VCSP s'est arrêtée jusqu'au dernier niveau e_k, c-à-d résolution du problème cible.

Remarque 29 En possibiliste, la résolution est particulière. Elle ne se fait pas dans la structure de valuation possibiliste mais dans celle de la satisfaction. En effet, le découpage en résolutions successives avec un niveau de relaxation croissant rend inutile l'utilisation de l'opérateur possibiliste.
Bien que dans le cadre de la satisfaction, le fait d'ajouter des contraintes ne rend pas forcément plus dur à résoudre un problème ; l'enchaînement proposé à l'étape possibiliste est généralement de complexité croissante. En effet, l'ordre croissant des niveaux de relaxation correspond à des problèmes simplifiés cohérents puis incohérents. La transition de phase entre cohérence et incohérence marque l'arrêt de l'étape possibiliste, le passage à l'étape lexicographique et correspond à un pic de complexité, comme celui observé sur la figure 2.19.
Enfin, les niveaux de relaxation qui sont cohérents ne produisent pas de minorant, mais par contre plusieurs majorants.

$\diamond$

Remarque 30 Les instanciations complètes générées par les résolutions successives sont mémorisées dans une liste. Ainsi, avant d'enchaîner une nouvelle résolution, chaque instanciation complète de la liste est évaluée dans la nouvelle simplification en vue de récupérer un bon majorant pour accélérer cette prochaine résolution. De même, un minorant est calculé pour la nouvelle simplification à partir de la résolution précédente. Il permet d'arrêter la résolution si une solution a pour valuation ce minorant.

$\diamond$

Remarque 31 L'enchaînement des simplifications se termine dès que le minorant et le majorant du problème cible se rejoignent.

$\diamond$

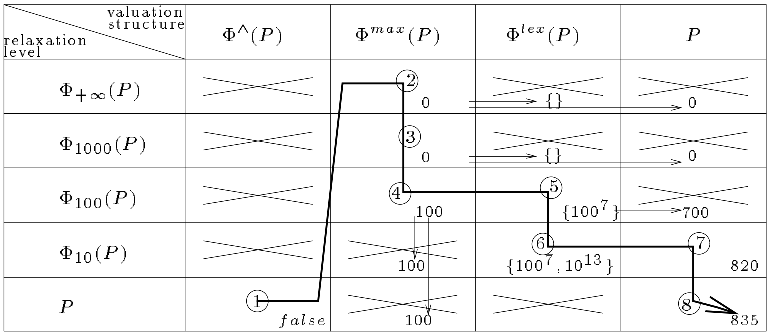
4.3.2 Application sur un exemple

Le problème cible est un Σ-VCSP contenant des contraintes impératives, de valuation $+\infty$, et des contraintes de préférence, de valuation 1000,100,10 et 1. L'optimum de ce problème vaut 835. La figure 4.19 représente le tableau des combinaisons potentielles de simplifications pour ce problème. Une ligne du tableau correspond à un niveau particulier de relaxation des contraintes. Une colonne correspond à une structure de valuation particulière. A chaque case est associée une combinaison de simplification. L'optimum, obtenu par résolution (dénoté par un numéro d'étape de résolution) ou par déduction (dénoté par une petite flèche), y est indiqué. La grande flèche indique les simplifications qui ont été effectivement résolues. Une croix dans une case signifie que la simplification a été court-circuitée.

Les différentes étapes de résolution des simplifications sont :

étape 1 la simplification $\Phi^{\wedge}(P)$ est résolue ; comme elle est incohérente, un premier minorant de l'optimum de P, égal à 1, peut être déduit en utilisant la fonction de transfert $\Psi^{\wedge}$;

étapes 2-3 les simplifications $\Phi^{max}_{+\infty}(P)$ et $\Phi^{max}_{1000}(P)$ sont résolues ; comme elles sont cohérentes, nous pouvons en déduire que les simplifications $\Phi^{lex}_{+\infty}(P)$, $\Phi^{lex}_{1000}(P)$, $\Phi^{\Sigma}_{+\infty}(P)$ et $\Phi^{\Sigma}_{1000}(P)$ sont aussi cohérentes ; elles ne seront pas résolues ;



step	0	1	4	5	6	7	8
lower bound	0	1	100	700	800	820	835

Fig. 4.19 - Exemple d'enchaînement de simplifications

étape 4 la simplification $\Phi_{100}^{max}(P)$ est résolue ; comme elle est incohérente, un second minorant, égal à 100, peut être déduit en utilisant la fonction de transfert Ψ_{100}^{max} ; nous pouvons aussi déduire que les simplifications $\Phi_{10}^{max}(P)$ et $\Phi^{max}(P)$ ont aussi le même optimum, égal à 100 ; il est inutile de les résoudre ;

étape 5 la simplification $\Phi_{100}^{lex}(P)$ est résolue ; comme son optimum vaut $\{100^7\}$, c-à-d n'importe quelle solution optimale viole 7 contraintes de priorité 100, un troisième minorant, égal à 700, peut être déduit en utilisant la fonction de transfert Ψ_{100}^{lex} ; nous pouvons aussi en déduire que l'optimum de la simplification $\Phi_{100}(P)$ vaut 700 ; elle ne sera pas résolue ;

étape 6 la simplification $\Phi_{10}^{lex}(P)$ est résolue ; comme son optimum vaut $\{100^7, 10^{13}\}$, un quatrième minorant, égal à 800, peut être déduit en utilisant la fonction de transfert Ψ_{10}^{lex} ; puisque la condition $\Psi_{10}^{lex} < 7*100+13*10$ est devenue vrai, il est inutile de poursuivre les résolutions en lexicographique car le minorant du problème cible n'augmentera pas ; la simplification $\Phi^{lex}(P)$ ne sera pas résolue ;

étape 7 la simplification $\Phi_{10}(P)$ est résolue ; comme son optimum vaut 820, un cinquième minorant, aussi égal à 820, peut être déduit en utilisant la fonction de transfert Ψ_{10} ;

étape 8 finalement, nous résolvons le problème cible P, dont l'optimum est égal à 835.

En bas de la figure 4.19, nous avons résumé la progression des minorants en fonction du numéro d'étape de la simplification.

4.3.3 Résultats expérimentaux

Nous avons mené des expérimentations sur les mêmes problèmes additifs binaires aléatoires Pm et Pa que ceux de la section 4.2.2. La méthode complète utilisée est un algorithme *Depth First Branch and Bound* avec *forward checking* valué et ordre dynamique pour le choix des variables et des valeurs. A cela s'ajoute une détection des composantes connexes à chaque niveau de relaxation des contraintes. La méthode incomplète utilisée comme comparaison est un algorithme de recuit simulé éventuellement mixé avec une descente en gradient (cf. section 2.3.2).

Le minorant du problème cible est mis à jour chaque fois que la résolution d'une simplification est terminée. Le majorant est mis à jour chaque fois que la résolution de la simplification courante produit une meilleure solution pour le problème cible. Pour chaque problème cible, nous mémorisons les courbes d'évolution au cours du temps des minorants et des majorants produits par l'enchaînement des simplifications.

Le comportement de l'enchaînement des simplifications est comparé à celui d'une résolution unique, par une méthode complète, ou par une méthode incomplète. Pour cela, nous examinons deux exemples pris sur Pm et Pa, puis nous donnons une vue plus globale sur l'ensemble des problèmes traités.

4.3.3.1 Résultats pour un problème de Pm et un autre de Pa

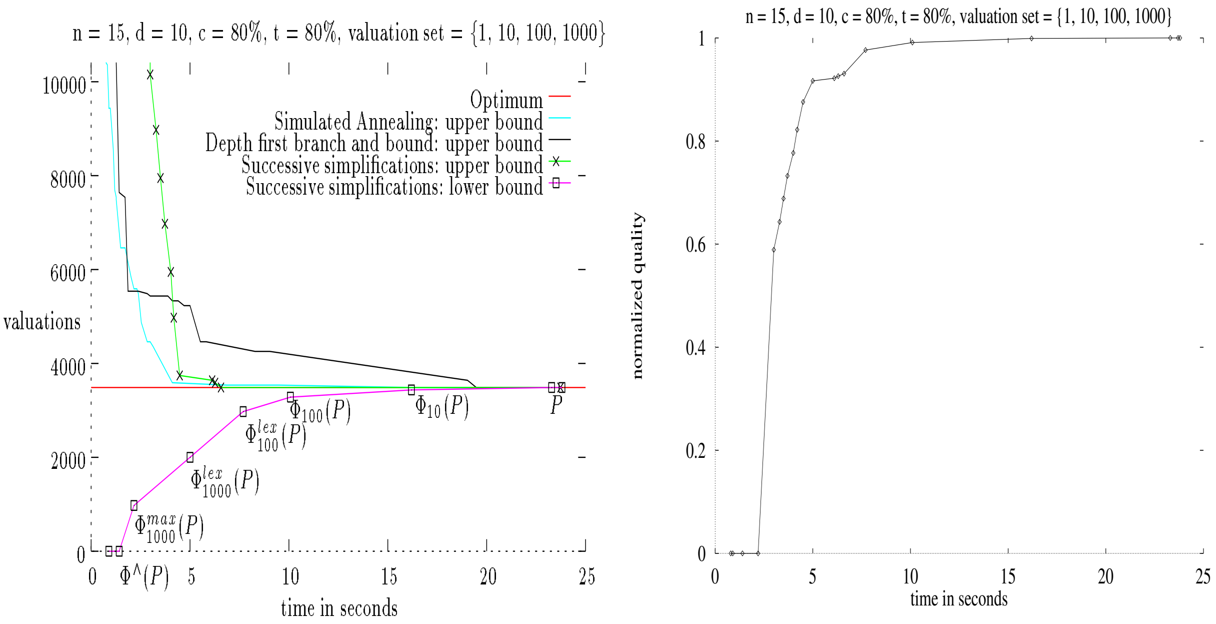


Fig. 4.20 - Résultat de l'enchaînement des simplifications pour un problème Pm

Nous étudions d'abord l'enchaînement des simplifications pour un problème de Pm dans la zone où la méthode est efficace : $c = 80\%, t = 80\%$. La figure 4.20 de gauche montre l'évolution de la valuation des majorants et minorants en fonction du temps (en secondes). Sur ce problème, l'optimum vaut 3477. L'algorithme de recuit simulé, réglé pour cette classe de problèmes[15], obtient le plus rapidement de bonnes solutions. L'algorithme *Depth First Branch and Bound* appliqué sur le problème cible, sans connaissance d'une borne initiale, met 23.8 secondes à terminer et prouver l'optimalité. Les deux dernières courbes, majorants et minorants, sont obtenues par l'enchaînement des simplifications. En dessous de chaque nouveau minorant est indiqué la simplification qui l'a produit. Les résolutions des simplifications en lexicographique aux niveaux de relaxation 1000 et 100 produisent de bonnes solutions. L'optimum est trouvé en 6.6 secondes. La progression des minorants, déduits des différentes simplifications, est : $\Psi(\Phi^\wedge(P)) = 1$, $\Psi(\Phi^{max}_{1000}(P)) = 1000$, $\Psi(\Phi^{lex}_{1000}(P)) = 2000$, $\Psi(\Phi^{lex}_{100}(P)) = 3000$, $\Psi(\Phi_{100}(P)) = 3300$, $\Psi(\Phi_{10}(P)) = 3460$ et $\Psi(P) = 3477$, prouvé après 23 secondes.

Remarque 32 Les temps de résolution prennent tout en compte, à l'exception du chargement du problème cible. Ainsi, le temps pour changer la structure de valuation et pour relaxer des

[15] décroissance géométrique de la température: 0.99, taille des paliers: 15, nombre d'étapes de gradient: 150.

contraintes par niveau est pris en compte dans la courbe des minorants.

$\diamond$

La figure 4.20 de droite montre l'évolution de la qualité normée approchée fournie par l'encadrement de l'optimum. La qualité normée approchée, notée $\varrho(A, P)$ (cf. équation 3.3), est similaire à $\rho_{ub}(A, P)$, excepté que l'optimum, inconnu pendant la résolution, est remplacé par le minorant courant, noté lb(P) :

$$\varrho(A, P) = \frac{\omega - \varphi(A, P)}{\omega - lb(P)} \tag{4.12}$$

Il suffit de 7 secondes pour que la qualité normée du résultat courant soit garantie supérieure à 95%.

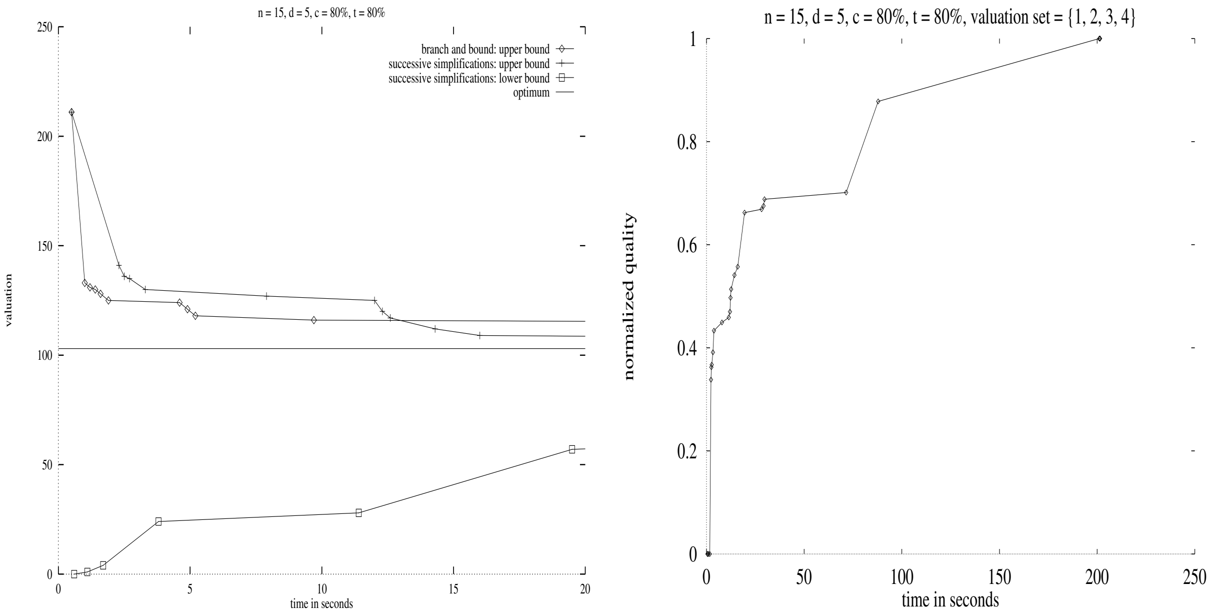


Fig. 4.21 - Résultat de l'enchaînement des simplifications pour un problème Pa

Le second exemple est un problème de Pa. Ici, le minorant évolue beaucoup plus lentement. Il faut environ 100 secondes pour avoir une garantie de qualité normée supérieure à 90%. Il a fallu 168 secondes à une méthode complète appliquée sur tout le problème pour terminer et prouver un optimum à 103.

4.3.3.2 Résultats pour un ensemble de problèmes

Les figures 4.22 et 4.23 montre l'évolution de la qualité ρ_{ub} des majorants (cf. équation 4.10) et de la qualité ρ_{lb} des minorants (cf. équation 4.11). Cette qualité est exprimée par un pourcentage allant de 0% (majorant égal à ω, minorant égal à 0) à 100% (majorant et minorant égaux à l'optimum). L'utilisation de ces qualités normées exactes permet de moyenner les résultats d'un ensemble de 100 problèmes cibles.

La figure 4.22 de gauche est une moyenne sur 100 problèmes Pm. Il faut environ 18 secondes à l'algorithme *Depth First Branch and Bound* pour atteindre une qualité supérieure à 95%, alors

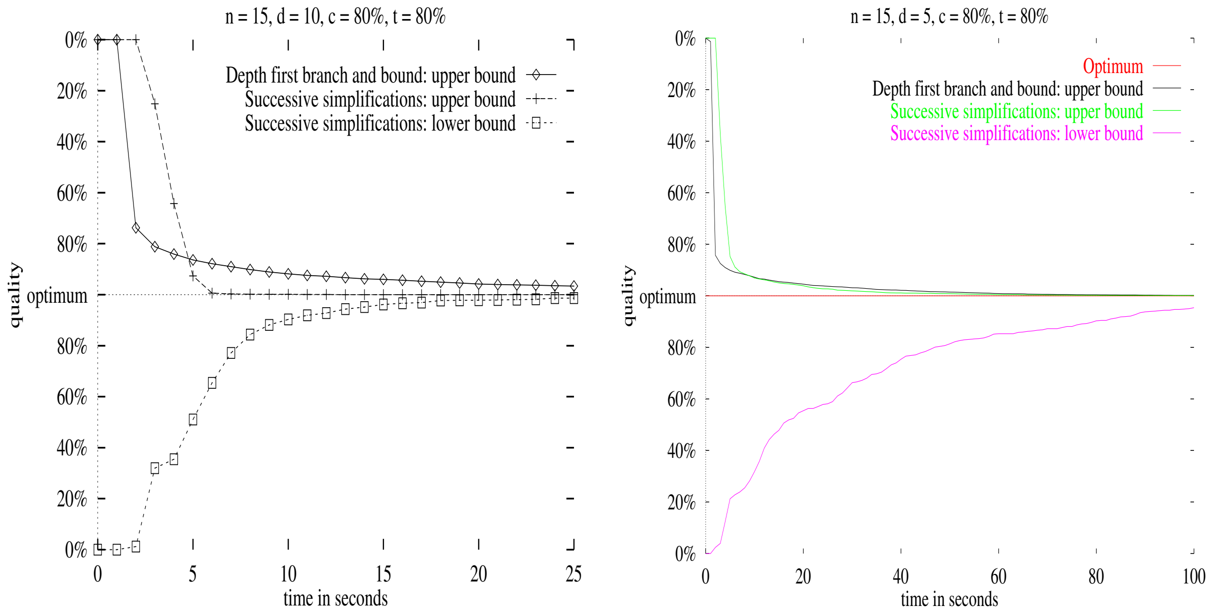


Fig. 4.22 - Qualité des majorants et minorants en fonction du temps : moyenne sur 100 problèmes pour Pm et Pa à connectivité et dureté fixées

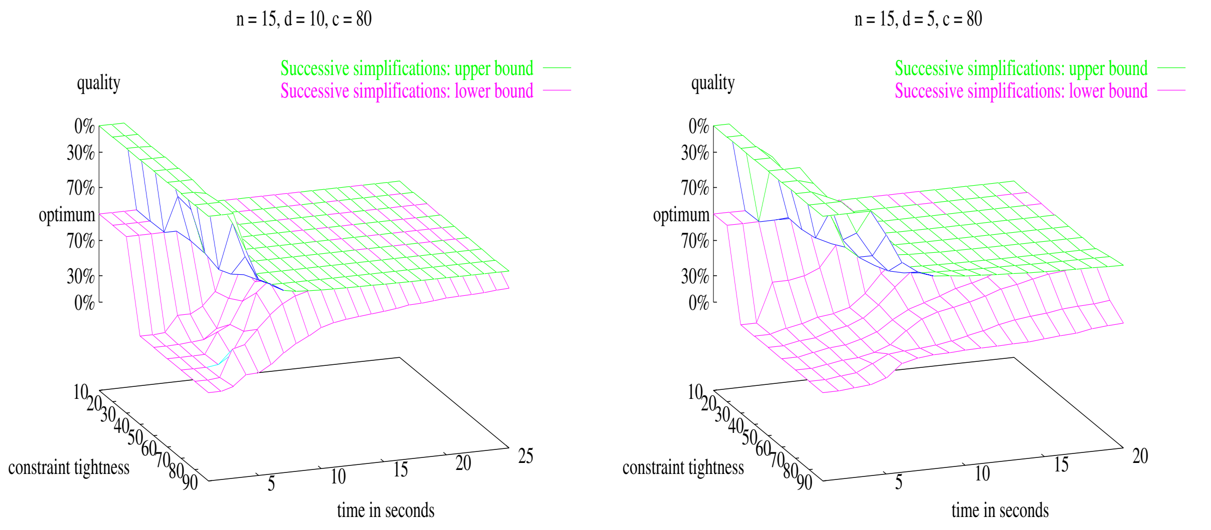


Fig. 4.23 - Qualité des majorants et minorants en fonction du temps : moyenne sur 100 problèmes pour Pm et Pa à connectivité fixée

que 6 secondes (resp. 14 secondes) suffisent pour atteindre ce même seuil par l'enchaînement des simplifications en majorant (resp. en minorant). Pour les problèmes Pa, figure 4.23, seuls les majorants obtenus par simplification restent compétitifs face à l'algorithme *Depth First Branch and Bound*.

La figure 4.23 montre l'évolution en trois dimensions de la qualité moyenne des majorants et minorants obtenus par simplifications successives. Seule la connectivité reste fixée, alors que la dureté varie de 10% à 90% par pas de 10%. Nous constatons que l'intervalle constitué par les deux bornes se resserre plus lentement lorsque la dureté augmente. Cela est dû au fait que les problèmes sont plus durs à résoudre quand la dureté est élevée.

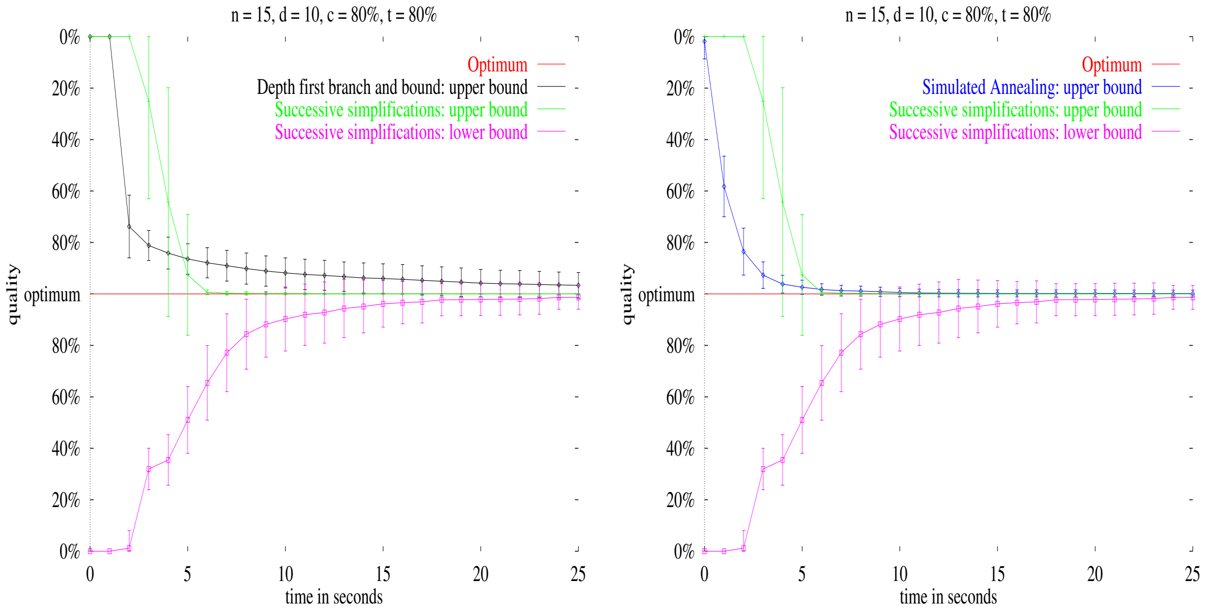


Fig. 4.24 - Variance sur la qualité des majorants et minorants en fonction du temps : moyenne sur 100 problèmes pour Pm à connectivité et dureté fixées ; comparaison avec l'algorithme *Depth First Branch and Bound* et l'algorithme du recuit simulé

La figure 4.24 reprend la figure 4.22 de gauche en y ajoutant l'information d'écart-type à la moyenne. L'écart-type sur la qualité des majorants produits par l'algorithme *Depth First Branch and Bound* décroît rapidement à 6% puis reste relativement stationnaire. A l'inverse, l'écart-type est beaucoup plus important pour les majorants et minorants produits par les simplifications successives (environ 44% pour les majorants et 15% pour les minorants). Mais ensuite, il diminue plus fortement au cours du temps (0.9% après 6 secondes pour les majorants et 4.6% après 25 secondes pour les minorants). La figure 4.24 de droite montre que l'écart-type diminue plus vite jusqu'à devenir très faible pour les majorants produits par l'algorithme du recuit simulé. En revanche, l'optimum n'est pas toujours atteint, donc l'écart-type reste non nul (1.3% au bout de 33 secondes).

L'analyse de l'écart-type montre que les enchaînements de simplifications offrent une robustesse, face à une série de problèmes, qui va bien croissante avec le temps. La qualité des majorants devient même supérieure en peu de temps à celle obtenue par une recherche locale, pourtant finement réglée. Cette robustesse est obtenue grâce à la diversité des simplifications résolues qui ont pour effet de fournir des bons majorants et grâce à l'évolution progressive de la complexité des résolutions[16].

4.3.3.3 Analyse de l'efficacité des enchaînements de simplifications

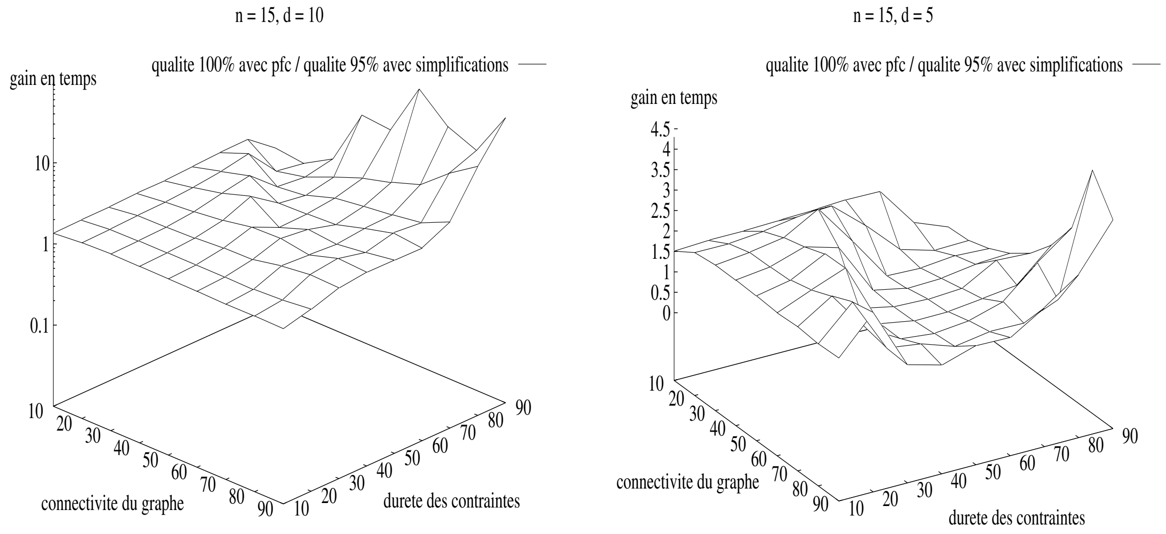


Fig. 4.25 - Gain en temps pour obtenir une qualité normée supérieure à 95% par rapport au temps mis par une méthode complète appliquée sur le problème cible Pm ou Pa

Nous analysons d'abord l'efficacité globale de l'enchaînement, c-à-d l'évolution de la qualité normée approchée (cf. équation 4.12 ou 3.3) par rapport au temps mis pour trouver et prouver l'optimalité par une méthode complète appliquée directement sur le problème cible. La figure 4.25 montre le gain en temps pour obtenir une qualité normée supérieure à 95% par rapport au temps mis par une méthode complète appliquée sur le problème cible Pm ou Pa. Ce gain est plus important lorsque les problèmes ont une forte dureté ou, dans une moindre mesure, une forte connectivité. Par exemple, pour Pm (resp. Pa), le gain est d'environ 7 (resp. 1.3) à connectivité et dureté égales à 80%. Il est aussi intéressant pour les problèmes satisfiables dû à la résolution en satisfaction faite en premier, surtout au niveau de la frontière entre cohérence et incohérence. En revanche, le gain est plus faible dans la zone intermédiaire des problèmes insatisfiables, surtout lorsque la connectivité est faible (le gain descend à 1.2 pour Pm et 0.4 pour Pa) . Dans l'ensemble, le gain est nettement plus faible, voir inférieur à 1, pour les problèmes Pa.

L'efficacité de l'enchaînement est surtout dû à la bonne qualité des majorants. La figure 4.26 montre le temps moyen pour obtenir des majorants à 5% de l'optimum, obtenus soit par les simplifications successives, soit par une méthode complète appliquée sur le problème cible. La figure 4.27 montre combien il est plus difficile de produire des minorants que de trouver des majorants, à une même distance de l'optimum (à 5%), en utilisant les simplifications successives.

[16] La variance sur les temps de résolution pour prouver des minorants semble plus faible que celle pour obtenir des majorants. Sur la figure 2.21, l'écart-type est en moyenne une fois et demi plus grand pour les majorants ($\alpha > 20$) que pour les minorants ($\alpha \leq 20$), à une même distance de l'optimum.

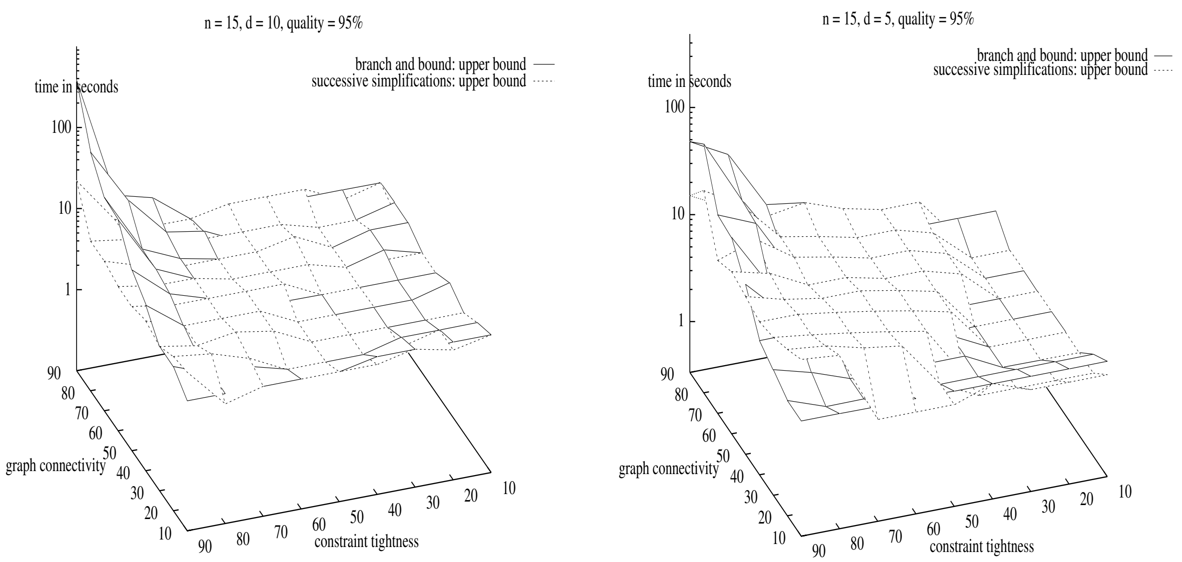


Fig. 4.26 - Temps moyen pour obtenir des majorants à 5% de l'optimum par une méthode complète appliquée sur le problème cible et par l'enchaînement des simplifications

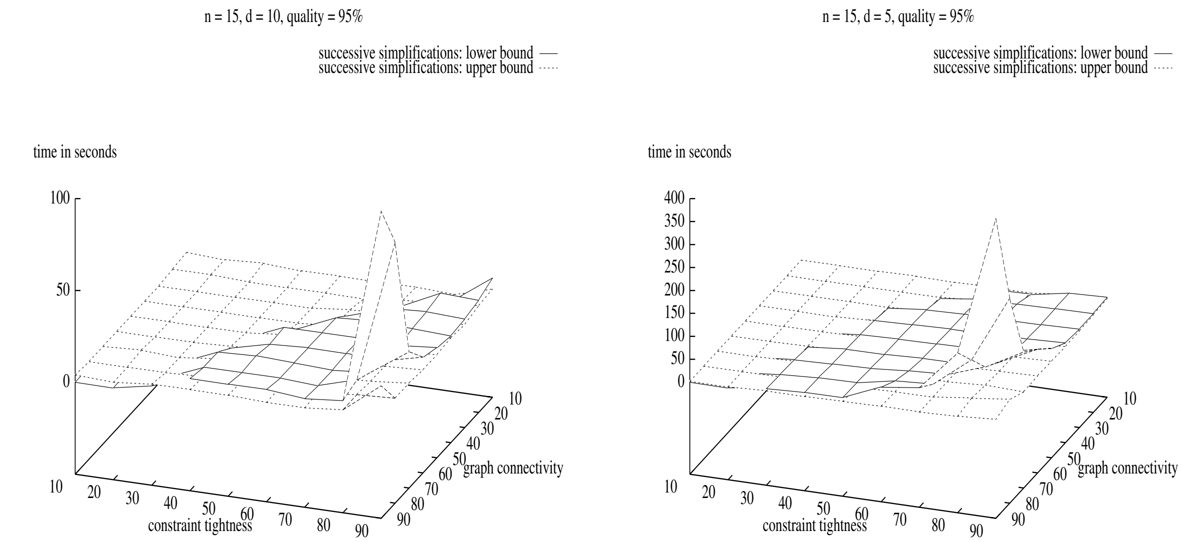


Fig. 4.27 - Temps moyen pour obtenir des majorants et des minorants à 5% de l'optimum

4.3.3.4 Bilan

Au niveau des performances observées sur les problèmes Pm et Pa, nous constatons qu'elles sont bonnes sur une large gamme de valeurs des paramètres connectivité et dureté. C'est le cas sur les problèmes sous-contraints, du fait de la première étape de l'enchaînement faite en satisfaction. C'est aussi le cas sur les problèmes faiblement incohérents car des solutions sont rapidement trouvées pour les relaxations par niveau et car la fonction de transfert Ψ_{lex} s'avère efficace. C'est enfin le cas sur les problèmes très durs (forte connectivité et surtout forte dureté, supérieure à 80%) grâce aux bons majorants obtenus par des résolutions successives diversifiées et également, une légère reprise des minorants. Bien sûr, l'enchaînement bénéficie de niveaux de valuations espacés, ce qui favorise la fonction Ψ_{lex}, et du fait que ces différents niveaux soient nombreux, ce qui rend plus progressif l'évolution du minorant. L'obtention de bons majorants, comparables à ceux obtenus grâce à des méthodes de type recherche locale, peut s'expliquer d'une part par la diversité des espaces de recherche parcourus par les résolutions successives (du fait du changement de comportement des heuristiques sur les valeurs lorsque le niveau de relaxation et l'opérateur d'agrégation changent) et d'autre part par le fait que les zones contenant des bons majorants semblent similaires quel que soit l'opérateur d'agrégation (cf. figure 4.5).

Par ailleurs, l'enchaînement proposé est en adéquation avec une approche interruptible et délibérative. En effet, nous enchaînons bien des résolutions de complexité globalement croissante. Cela produit un encadrement progressif de l'optimum. De plus la robustesse des majorants et minorants (mesurée par leurs écart-types) est croissante avec le temps. Enfin l'enchaînement garantit l'optimalité à terme.

Néanmoins les résolutions subissent l'évolution de la complexité constatée pour les méthodes complètes de résolution lorsque la connectivité ou la dureté augmentent, même si c'est une simplification du problème qui est résolue. La même difficulté survient lorsque la taille du problème augmente. Cela se traduit par un blocage dans l'enchaînement au niveau où la résolution devient hors de portée : il y a alors absence d'informations pertinentes. Sinon, sur les problèmes Pm et surtout Pa, il reste des zones dans lesquelles les performances de l'enchaînement sont mauvaises, du fait des faiblesses des fonctions de transfert ou d'un nombre de résolutions successives trop faible.

Ce bilan n'est valable que pour une certaine catégorie de problèmes aléatoires. Il est nécessaire pour mieux valider notre approche de la comparer à d'autres approches et de la mesurer sur des problèmes de taille plus grande. Nous présenterons également des extensions à l'enchaînement décrit dans la section 4.3.1.

4.4 Validation et extensions

Premièrement, nous effectuons quelques validations de la technique d'enchaînement de simplifications. Il s'agit de voir comment réagit cette technique face à d'autres techniques pour produire des minorants et face à des problèmes plus gros, en changeant les méthodes de résolution.

Deuxièmement, nous présentons des extensions à la technique d'enchaînement de simplifications. Ces extensions visent à accélérer les résolutions successives des simplifications.

4.4.1 Validation de l'enchaînement des simplifications

Comparaison avec d'autres méthodes produisant des minorants

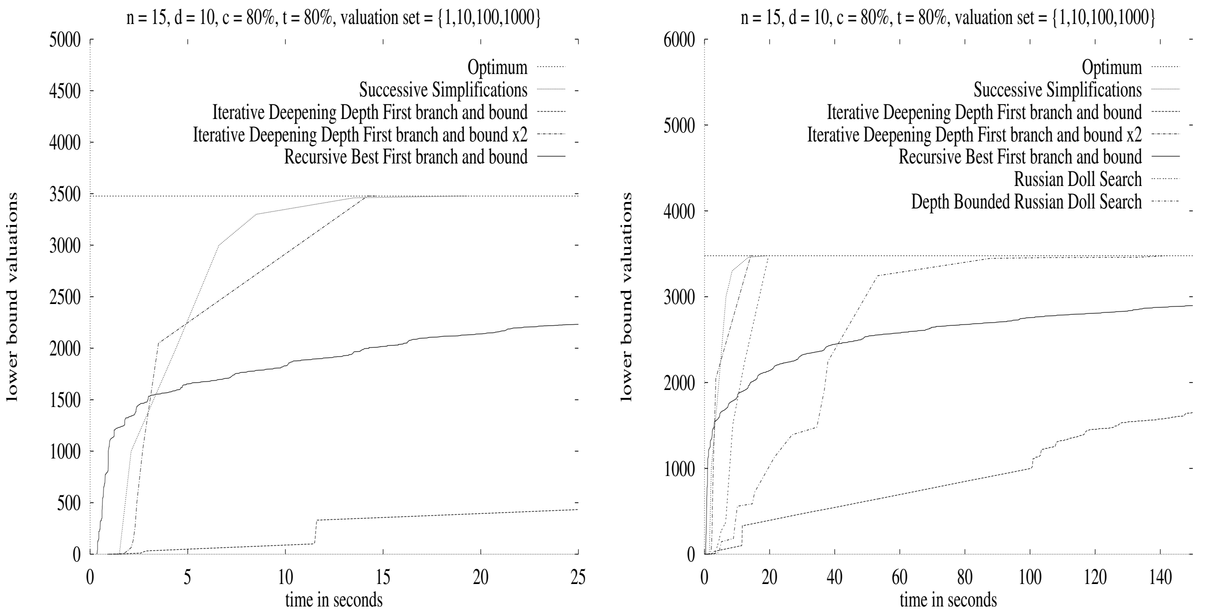


Fig. 4.28 - Evolution des minorants produits par 6 algorithmes différents sur un problème Pm

La comparaison n'est faite que sur un problème Pm où l'enchaînement donne de bons résultats (à connectivité = 80% et dureté = 80%). L'optimum est à 3477. Six algorithmes différents pour produire des minorants sont comparés en terme d'évolution de la valuation du minorant en fonction du temps (en secondes). La figure 4.28 de gauche[17] montre que les simplifications successives produisent, sur cet exemple, de bien meilleurs minorants que les deux algorithmes classiques, proposés dans [Kor85, Kor93], *Iterative Deepening Depth First Branch and Bound* (équivalent à *Iterative Objective Relaxing*) et *Recursive Best First Branch and Bound* (cf. section 6.2 pour une description de ces algorithmes). Il semble que ces deux algorithmes soient mal adaptés à de fortes variations sur les coûts associés aux contraintes.

Une modification de l'algorithme *Iterative Deepening Depth First Branch and Bound*, appelée *Iterative Deepening Depth First Branch and Bound* $*2$ améliore beaucoup son fonctionnement. Cela consiste à multiplier par 2 la borne initiale donnée au *Branch and Bound* à chaque itération. Cela assure une croissance géométrique du minorant et aussi de la complexité de la résolution. L'inconvénient de cette méthode réside dans son manque de progressivité lors des dernières étapes car la croissance de la borne initiale devient trop rapide. Cela se traduit par une courbe du minorant moins pleine que celle produite par les simplifications successives. Il faudrait diminuer le facteur multiplicatif au cours des itérations, à l'instar de l'algorithme *TCGD* [WC97, CW91]. Néanmoins, le bon réglage semble difficile à généraliser, et repose sur une analyse statistique des problèmes traités. C'est une approche que nous avons a priori écarté, à cause de la variance observée sur ce type de statistiques.

[17] Nous trions les méthodes suivant le minorant qu'elles ont produit au bout de 8 secondes. Cela donne par ordre décroissant, les simplifications successives en premier, puis *Iterative Deepening Depth First Branch and Bound* $*2$, puis *Recursive Best First Branch and Bound*, puis *Iterative Deepening Depth First Branch and Bound.*

La figure 4.28 de droite[18] ajoute deux autres algorithmes et augmente la durée de résolution à 150 secondes. Le premier algorithme rajouté est une méthode complète, appelée *Russian Doll Search* [LV96], qui résout des sous-problèmes imbriqués jusqu'à résoudre le problème complet (cf. section 2.3.1.4). Ces résolutions successives ont pour principal intérêt de fournir des informations plus précises sur la valuation globale d'une instanciation partielle. Ces informations sont surtout utilisées par les derniers niveaux (cf. section 2.3.1.2). En pratique, l'évolution du minorant est importante uniquement lors des dernières étapes, ce qui n'est pas favorable à une utilisation interruptible. Le second algorithme rajouté essaie d'améliorer l'évolution du minorant lors des premières étapes moyennant une augmentation du temps globale de résolution. Il combine les principes de l'algorithme *Russian Doll Search* avec ceux de l'*Iterative Deepening Search* (profondeur limitée croissante) (cf. section 6.2.3.2). Cette combinaison est plutôt efficace sur des problèmes très contraints et impossibles à résoudre complètement en un temps raisonnable, ce qui n'est pas le cas de l'exemple choisi pour la comparaison.

Utilisation d'autres méthodes de résolution pour affronter des problèmes de plus grande taille

Pour tester l'enchaînement sur des problèmes de plus grande taille, il est nécessaire d'employer des méthodes de résolution plus adaptées. Premièrement, cela consiste à utiliser des méthodes complètes comportant une meilleure évaluation de la valuation globale des instanciations partielles. Deuxièmement, l'enchaînement des simplifications peut être combiné avec une méthode incomplète pour accélérer la production de bons majorants.

Utilisation d'autres méthodes complètes

Plutôt que d'utiliser un simple *Branch and Bound* avec *forward checking*, les simplifications peuvent être résolues en utilisant une combinaison *forward checking* et *directed arc consistency counts* [Wal94, LM96a], notée *PFC+DAC*, ou bien en utilisant l'algorithme *Russian Doll Search* [LV96].

L'algorithme *PFC+DAC* impose un ordre statique sur les variables. D'après [Wal94], les valeurs des compteurs *DAC* ne sont calculées qu'une fois, avant la résolution. Nous avons étudié une mise à jour dynamique de ces compteurs, à l'instar d'une des extensions de *PFC+DAC* proposée dans [LMSV98]. En effet, au cours de la recherche, la suppression des valeurs des variables non affectées qui impliquent un coût supérieur à la borne courante peut conduire à une mise à jour d'un ou de plusieurs compteurs *DAC*. Nous avons implémenté une version, appelée *PFC+DAC complet*, en reprenant les structures de donnée de l'algorithme *AC4* et en effectuant à chaque noeud de l'arbre une mise à jour exhaustive des compteurs, de façon cyclique. La figure 4.29 montre le gain moyen en nombre de noeuds de *PFC+DAC complet* par rapport à l'ancien algorithme *PFC+DAC* sur l'ensemble des problèmes aléatoires binaires Pm et Pa. Le gain en temps est plus faible et reste proportionnel au gain en nombre de noeuds. Le gain en temps moyenné sur l'ensemble des problèmes Pm (resp. Pa) est de 2.17 (resp. 1.15). La figure 4.30 montre l'encadrement de l'optimum d'un problème à 30 variables, en utilisant l'enchaînement des simplifications résolues par *PFC+DAC complet*. Sur cet exemple, l'optimum est à 26037 et $\omega = \sum_{c \in C} \varphi(c) = 59997$ (cf. équation 4.12). Il a fallu 3112 secondes pour que l'enchaînement

[18] Nous trions les méthodes suivant le minorant qu'elles ont produit au bout de 8 secondes. Cela donne par ordre décroissant, les simplifications successives en premier, puis *Iterative Deepening Depth First Branch and Bound *2*, puis *Recursive Best First Branch and Bound*, puis *Russian Doll Search*, puis *Depth Bounded Russian Doll Search* (équivalent à *Iterative Deepening + Russian Doll Search*), puis *Iterative Deepening Depth First Branch and Bound*.

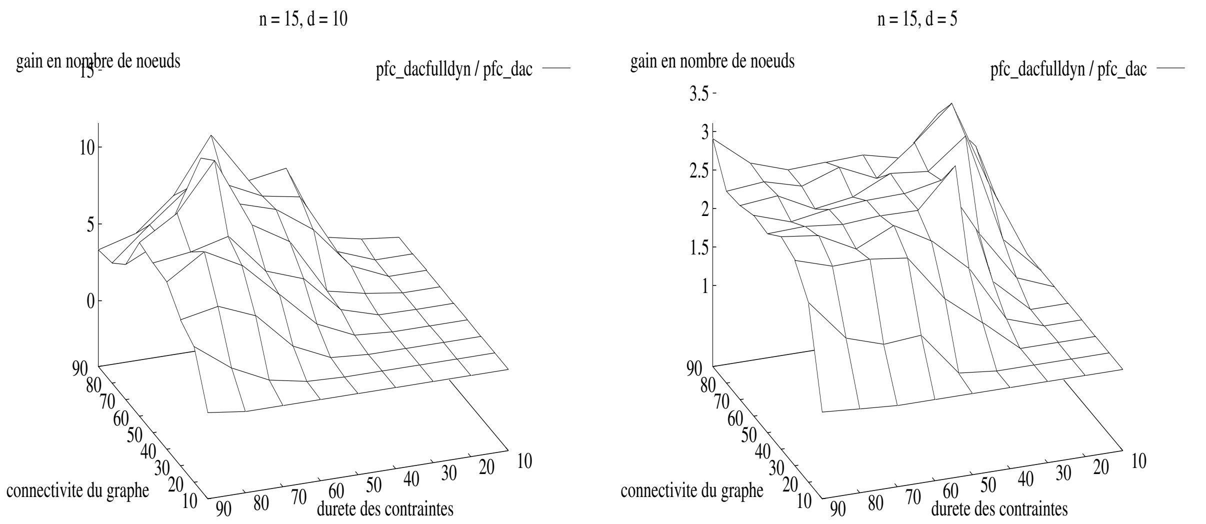


Fig. 4.29 - Gain en nombre de noeuds du nouvel algorithme *PFC+DAC complet* par rapport à l'ancien *PFC+DAC* sur les problèmes *Pm* et *Pa*

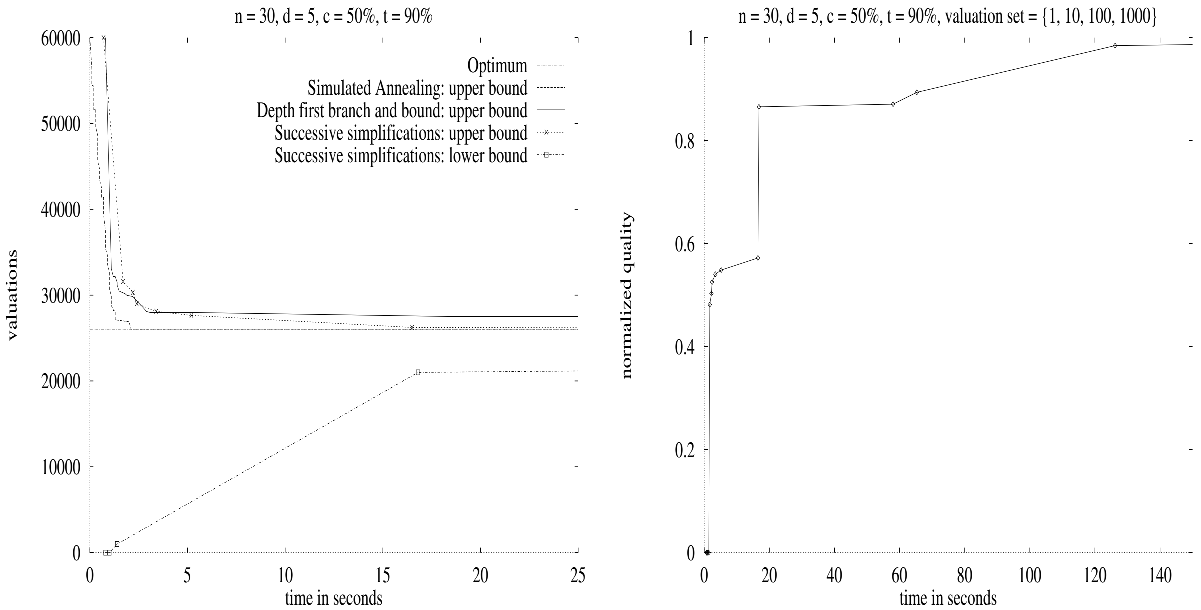


Fig. 4.30 - Résultat de l'enchaînement des simplifications, résolues par l'algorithme *PFC+DAC complet*, pour un problème à 30 variables

termine sur un *SUN UltraSparc* ; alors que le même algorithme *PFC+DAC complet*, exécuté sur le problème cible, a pris 3928 secondes. Pour comparaison, un algorithme de recuit simulé a trouvé l'optimum en moins de 30 secondes.

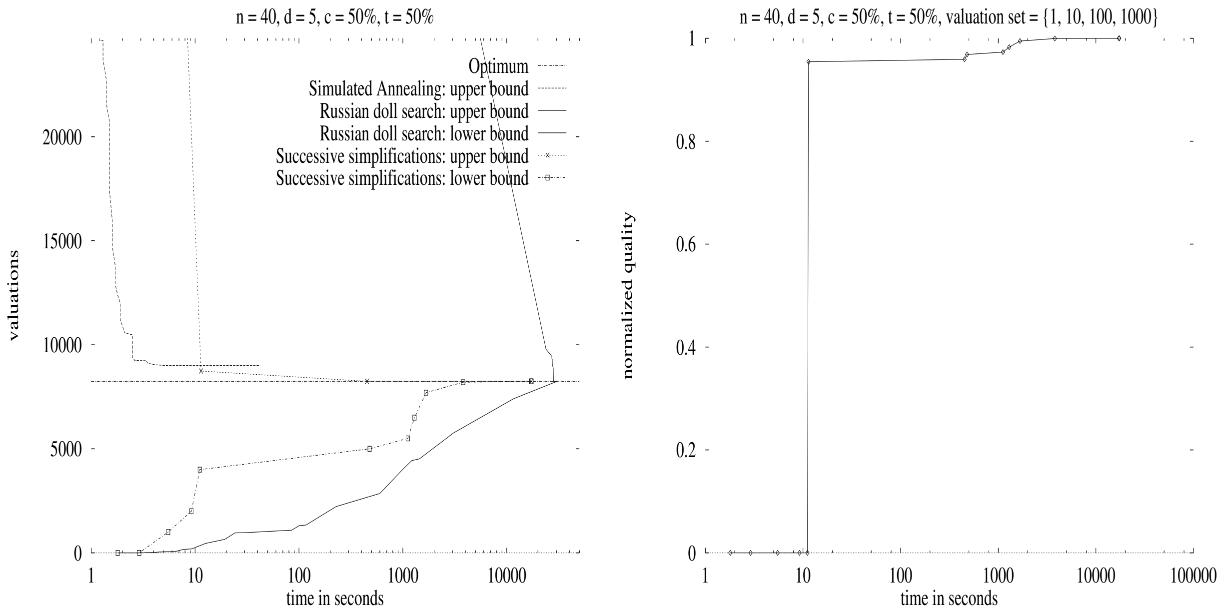


Fig. 4.31 - Résultat de l'enchaînement des simplifications, résolues par l'algorithme *Russian Doll Search*, pour un problème à 40 variables

La résolution en utilisant l'algorithme *Russian Doll Search* peut produire plusieurs minorants pour une seule simplification. Nous appliquons les fonctions de transfert à chaque fois qu'un sous-problème imbriqué est résolu. La figure 4.31 montre l'encadrement d'un problème à 40 variables, en utilisant l'enchaînement des simplifications résolues par l'algorithme *Russian Doll Search*. Ici, l'échelle de temps est logarithmique. Sur cet exemple, l'optimum est à 8244 et $\omega = 107770$. Il a fallu 17289 secondes (presque 5 heures) pour que l'enchaînement termine sur un *SUN UltraSparc* ; alors que le même algorithme *Russian Doll Search*, exécuté sur le problème cible, a pris 30500 secondes (environ 8 heures 30). Pour comparaison, un algorithme de recuit simulé a trouvé une valuation de 9004 en moins de 40 secondes.

Ajout de méthodes incomplètes

Pour des problèmes de grande taille, les résolutions complètes deviennent très longues. La production de majorants peut être nettement améliorée par l'utilisation de méthodes incomplètes, en parallèle aux méthodes produisant des minorants. Nous avons implémenté une exécution de l'enchaînement des simplifications coopérant avec des algorithmes de recuit simulé exécutés en parallèle.

Echange d'informations

La méthode inexacte met à jour le majorant du problème cible si une meilleure solution est trouvée. La méthode exacte agit de même. Une exclusion mutuelle sur la modification de la dernière meilleure solution évite tout problème de chevauchement des opérations de mise à jour.

Les solutions intermédiaires produites par la méthode inexacte sont aussi stockées dans la liste qui maintient les solutions intermédiaires produites par l'enchaînement de la méthode exacte. Elles augmentent les chances de trouver une bonne borne initiale pour accélérer l'algorithme *Branch and Bound* lors de la résolution des simplifications.

Enfin, lorsque la méthode exacte trouve un minorant égal au meilleur majorant trouvé soit par la méthode inexacte, soit par la méthode exacte, l'ensemble de l'application s'arrête.

Partage du temps de calcul

La technique du *multithreading* est employée pour obtenir une exécution simultanée des méthodes exactes et inexactes. Actuellement la priorité donnée aux deux méthodes est identique : elles se partagent le temps de calcul de manière équilibrée. Néanmoins, l'utilisateur peut choisir d'orienter l'effort sur la méthode inexacte en augmentant les paramètres de précision de cette méthode[19].

Une autre possibilité de privilégier et renforcer la méthode inexacte consiste à lancer plusieurs méthodes inexactes en parallèle. Chaque méthode correspondant à l'exécution d'un *thread* de même priorité, le *multithreading* conduit à un partage équilibré du temps de calcul entre tout les *threads*. Par exemple, si l'utilisateur choisit d'exécuter deux recuits simulés sur le problème cible, en parallèle à la résolution des simplifications, alors 66% du temps de calcul sera consacré à la méthode inexacte et 33% à la méthode exacte[20].

Exemple de coopération

La figure 4.32 montre l'encadrement de l'optimum par un enchaînement des simplifications avec ou sans recuit simulé. Le recuit simulé permet de produire des bons majorants plus rapidement mais ralenti d'un facteur 2 la production des minorants. Par ailleurs, le recuit simulé se termine avant la fin de l'enchaînement des simplifications, et donc le facteur de ralentissement disparaît.

Comportement face à des problèmes de plus grande taille

Que se passe-t-il lorsque la taille des problème augmente ? Forcément, le temps de résolution des simplifications augmente. Du coup, la décomposition en étapes successives risque d'être insuffisante pour faire évoluer régulièrement l'encadrement de l'optimum. C'est à dire que sur un temps limité, trop peu d'étapes seront résolues d'où des minorants très faibles. C'est déjà ce qui était observé sur la figure 4.21.

Tant que la résolution des dernières étapes se fait dans un temps raisonnable, l'enchaînement continue de produire un bon encadrement de l'optimum. C'est ce que montre les figures 4.30 et 4.31 sur des problèmes aléatoires à 30 variables et à 40 variables. Nous observons que la qualité normée dépasse les 95% en 126 secondes pour le problème en figure 4.30 et en 11 secondes pour celui de la figure 4.31. Le seuil des 99% est franchi en moins de 5 minutes pour 30 variables et

[19] Sinon le critère d'arrêt de la méthode inexacte risque d'être atteint bien avant la fin de l'enchaînement des méthodes exactes.

[20] Le 1% restant est consommé par le système d'exploitation assurant le *multithreading* !

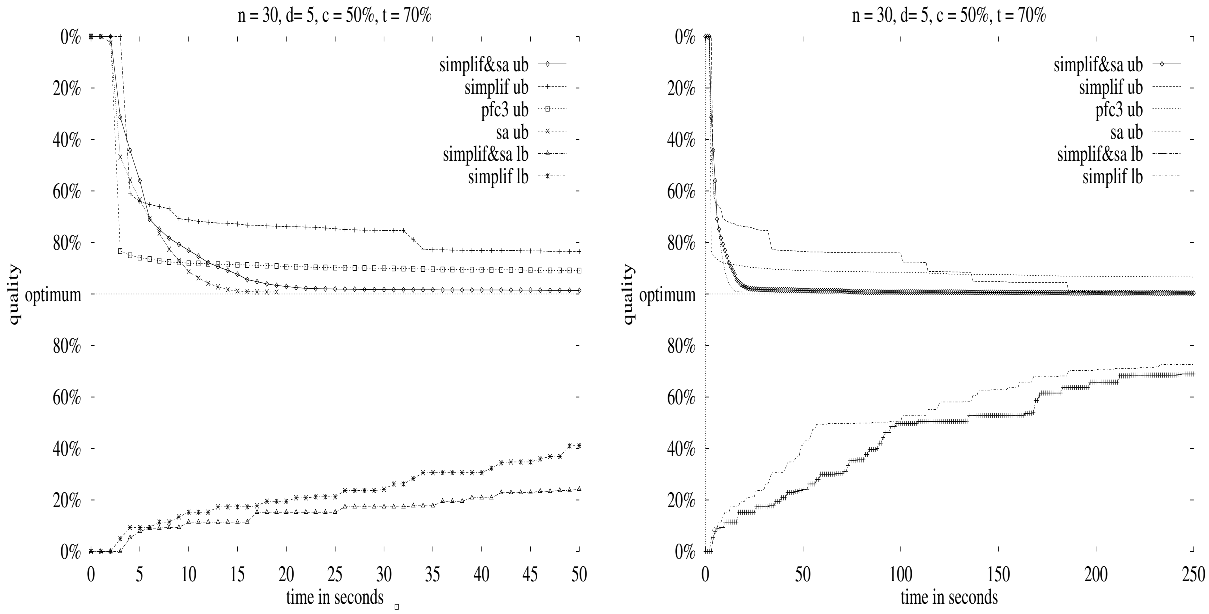


Fig. 4.32 - Coopération d'un recuit simulé avec l'enchaînement des simplifications

en moins de 30 min pour 40 variables.

Sur des problèmes réels, tels ceux de SPOT5 et du CELAR, l'enchaînement nécessite d'utiliser l'algorithme *Russian Doll Search* qui donne les meilleurs temps de résolution, parmi nos algorithmes de *Branch and Bound*.

Pour les problèmes SPOT5 que nous avons testés, le changement de structure de valuation n'apporte rien : la résolution en lexicographique est aussi longue que la résolution en additif car les coûts des contraintes sont souvent très espacés (choisis parmi $\{1, 2, 5, 1000, 2000\}$). En revanche, la relaxation des contraintes par niveau est efficace : la résolution développe 10 fois moins de noeuds en retirant seulement les contraintes de coût 1. La figure 4.33 montre l'enchaînement de ces relaxations sur une instance des problèmes SPOT5. C'est la première simplification, Φ_e, avec $e = 1000$, qui contribue le plus à l'augmentation du minorant. Les niveaux suivants sont nettement moins importants : $e = 5$, 2 puis 1. Il a fallu 156 secondes pour atteindre une qualité normée approchée de 99% sur une station *SUN UltraSparc*. L'algorithme *Russian Doll Search* a terminé en 232 secondes.

Nous avons expérimenté l'enchaînement des simplifications sur un sous-problème de l'instance 6 du CELAR. La figure 4.34 montre l'évolution de l'encadrement et de la qualité normée approchée sur ce sous-problème à 14 variables, appelé *CELAR6-SUB1* (cf. section 5.2.1.4). Pour comparaison, nous avons ajouté la courbe d'un algorithme de recuit simulé et aussi l'encadrement produit par l'algorithme *Russian Doll Search*. L'algorithme de recuit simulé trouve l'optimum, égal à 2669, en 15 secondes et termine en 400 secondes (paramètres $l = 50, g = 1000, m = 0.95$, cf. section 2.3.2). L'algorithme *Russian Doll Search* termine au bout de 450 minutes (environ 7 heures 30 min), alors que l'enchaînement termine en 1093 minutes (environ 18 heures). Il faut seulement 40 secondes pour atteindre une qualité normée approchée à 91%. Ceci est dû au faible coût de l'optimum par rapport à ω, la somme des coûts des contraintes. En revanche, il faut 5 heures pour être à moins de 1% de la qualité maximale.

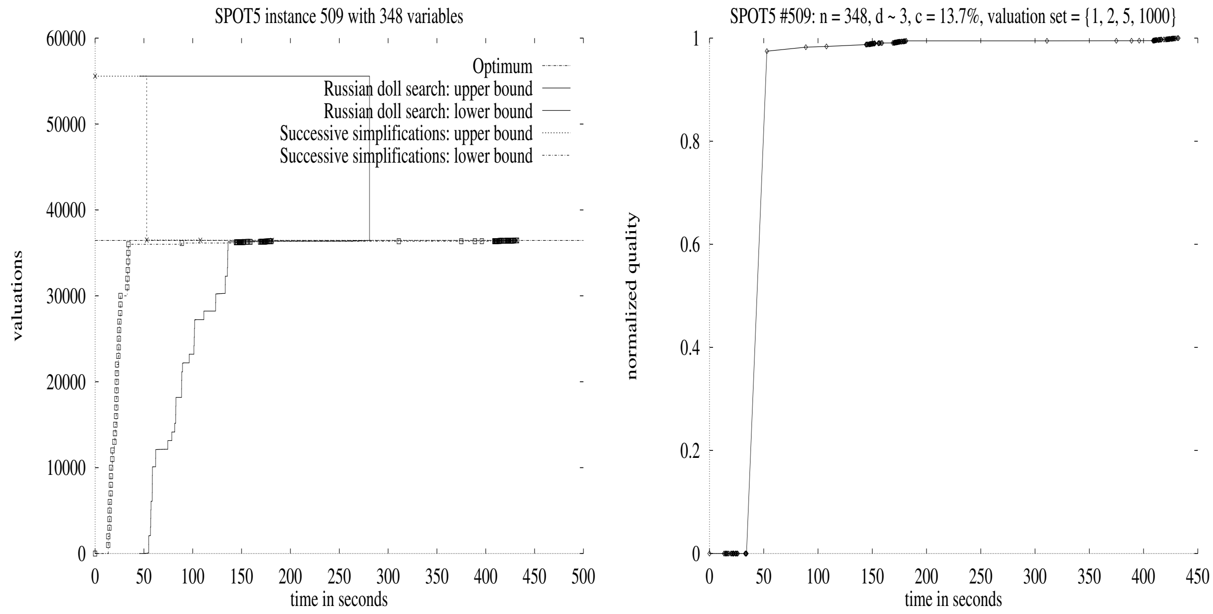


Fig. 4.33 - Résultat de l'enchaînement des simplifications, résolues par l'algorithme *RDS*, pour l'instance 509 des problèmes SPOT5

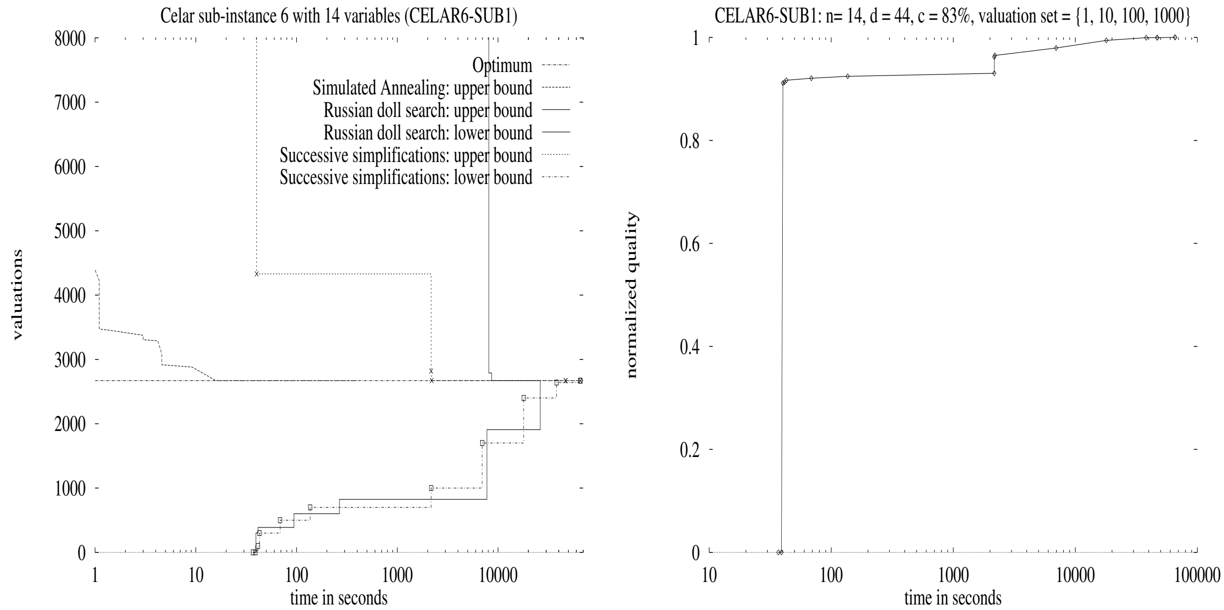


Fig. 4.34 - Résultat de l'enchaînement des simplifications, résolues par l'algorithme *RDS+MAC+DAC*, pour un sous-problème du CELAR (CELAR6-SUB1)

Le tableau suivant résume les différentes simplifications $\Phi_e^{\otimes}$ qui ont été résolues par l'enchaînement. Pour chaque simplification est écrit son optimum *opt*, les minorants et majorants déduits, ainsi que le temps de résolution obtenu sur un *SUN UltraSparc* :

$\otimes$	e	*opt*	$\Psi_e^{\otimes}(opt)$	$\varphi(opt, P)$	**temps**
$\wedge$	1	$\top$	1	-	0.3 sec
max	1000	0	-	4329	0.6 sec
max	100	$\top$	100	-	0.4 sec
lex	100	$\{100^{24}\}$	1000	2672	58 min
Σ	100	2400	2400	”	241 min
Σ	10	2640	2640	”	336 min
Σ	1	2669	2669	2669	459 min

4.4.2 Extensions pour accélérer l'enchaînement des simplifications

Etapes possibilistes

Nous avons choisi de résoudre la simplification en possibiliste par un découpage en niveaux, en commençant par une résolution en satisfaction des contraintes de plus forte priorité pour favoriser la production de bons majorants et aussi garder une progression croissante de la complexité. Il existe d'autres techniques pour aborder le cadre possibiliste.

Plutôt que de résoudre en commençant par le haut, nous pourrions commencer par le bas, c-à-d résoudre en satisfaction uniquement les contraintes de priorité supérieure ou égale à un seuil. Ce seuil part de e_k et remonte vers e_1 tant que le problème simplifié est incohérent. A chaque incohérence, nous pouvons déduire un minorant, égal à la valuation du seuil, alors que notre approche initiale n'en génère qu'un seul à la fin des étapes possibilistes. En revanche, aucun majorant n'est produit, excepté à l'arrêt des simplifications possibilistes, lorsque toutes les contraintes du seuil sont satisfiables. Par ailleurs, la complexité de résolution reste globalement croissante, jusqu'à atteindre la frontière entre cohérence et incohérence.

Une autre approche serait plus efficace dans le cas où le nombre de coûts différents associés aux contraintes est grand. Cette approche, dite dichotomique, consiste à conserver deux bornes, une inférieure, notée $bmin$, initialisée à 1 et l'autre supérieure, notée $bmax$, initialisée à k. Le niveau de relaxation e_{bmax} est par construction toujours insatisfiable, ainsi que les niveaux supérieurs (le niveau e_k étant insatisfiable, ce qui a été établi par la précédente simplification avec la structure de valuation des CSP classiques). Donc e_{bmax} est un minorant du problème cible. La première résolution en satisfaction considère toutes les contraintes supérieures ou égales à la valuation $e_{E(\frac{bmax+bmin}{2})}$. $E()$ est la fonction "*partie entière*". Si le problème simplifié est satisfiable, la borne inférieure devient égale à $E(\frac{bmax+bmin}{2})$, sinon c'est la borne supérieure qui prend cette valeur moyenne. Le processus continue jusqu'à ce que $bmin = bmax + 1$. Alors, l'optimum en possibiliste vaut e_{bmax} si le problème au niveau de relaxation e_{bmin} est satisfiable, sinon il vaut e_{bmin}.

Enfin, une résolution en possibiliste sans effectuer de relaxation par niveau peut s'avérer plus efficace. A l'instar de l'approche présentée initialement, cette résolution peut produire plusieurs majorants mais seulement un minorant. Si la première étape qui correspond à trouver une solution satisfaisant toutes les contraintes échoue, l'information transmise à la résolution en possibiliste est une borne inférieure d'arrêt égale à e_k.

Etapes lexicographiques

Dans l'approche initiale présentée en section 4.3.1, nous avons donné un critère d'arrêt des étapes lexicographiques fondé sur l'absence d'amélioration du minorant. Soit $\{e_1^{m_1}, \ldots, e_i^{m_i}\}$, la solution lexicographique optimale au niveau e_i. Le critère d'arrêt est égal à $lex(\{e_1^{m_1}, \ldots, e_i^{m_i}\}) < \sum_{l=1}^{i} m_l e_l$. Nous pouvons remarquer que le minorant déduit à cette étape n'utilise pas pleinement l'information m_i (nombre minimum de contraintes violées au niveau de priorité e_i). En effet, il suffit que $m_i > \frac{e_{i-1}}{e_i}$ pour que le critère d'arrêt soit vérifié[21].

Dans le cas où la production de bons majorants n'est pas prioritaire pour la dernière étape lexicographique, il est possible d'accélérer la méthode complète de résolution. Cela est fait en imposant une borne initiale à la résolution, éventuellement inférieure au meilleur majorant déduit des simplifications précédentes pour la nouvelle étape. Soit $\{e_1^{b_1}, \ldots, e_i^{b_i}\}$, le meilleur majorant connu pour la simplification lexicographique au niveau e_i. La borne initiale, notée ub_{lex}, vaut $\{e_1^{b_1}, \ldots, e_{i-1}^{b_{i-1}}, e_i^{M}\}$, avec M égal à[22] :

$$M = \left\lceil \frac{(lex(\{e_1^{b_1}, \ldots, e_i^{b_i}\}) - \sum_{l=1}^{i-1} b_l e_l)}{e_i} \right\rceil$$

A chaque niveau de relaxation des contraintes, la résolution en lexicographique peut utiliser ub_{lex}, en recalculant M. Ainsi, ces résolutions consacreront leurs efforts de calcul uniquement pour prouver des minorants de l'optimum du problème cible.

Utilisation de propriétés algorithmiques

Un facteur principal d'accélération des simplifications par changement de la structure de valuation réside dans les propriétés algorithmiques des sous-cadres VCSP employés (cf. section 2.1.2). En effet, lorsque l'opérateur d'agrégation est idempotent (valable en satisfaction et en possibiliste), nous pouvons appliquer des techniques de renforcement de cohérence locale qui s'avèrent efficaces lors d'une résolution [BR96, Sch92, SFV95].

Ces techniques sont applicables sur les contraintes dures même si l'opérateur d'agrégation n'est pas idempotent. Cela nous a amené à développer une extension de l'algorithme *Branch and Bound* pour prendre en compte de façon séparée les contraintes devant être obligatoirement satisfaites, à l'instar de ConPlan *solver* [aH96]. L'originalité de notre approche réside dans la prise en compte dynamique de ces contraintes. Le principe consiste à maintenir une liste des contraintes impératives qui contienne toutes les contraintes dont la valuation est supérieure ou égal à la valuation de la borne supérieure courante du *Branch and Bound*. Cette liste est remise à jour à chaque fois qu'une instanciation complète de meilleure valuation est trouvée. A chaque noeud de l'arbre de recherche, nous appliquons un maintien d'arc-cohérence sur cette liste de contraintes impératives[23].

Nous avons expérimenté cette approche en la combinant avec divers algorithmes de *Branch and Bound* dont l'algorithme *Russian Doll Search* [LV96], la combinaison étant notée *RDS+MAC3*.

[21] Cette condition est suffisante mais pas forcément nécessaire.

[22] $\lceil\rceil$ correspond à la partie entière plafond (entier juste supérieur à la valeur réelle).

[23] Nous avons utilisé dans nos expérimentations l'algorithme *MAC3* [Tsa93, Pro95], choisi pour sa simplicité de codage. Des versions plus récentes, telle que *MAC7* [R95], devraient offrir de meilleurs performances, surtout si le nombre de contraintes impératives est grand.

Les tests ont été faits sur plusieurs sous-problèmes de l'instance 6 du CELAR[24]. Ces sous-problèmes, appelés *CELAR6-SUB[1-4]*, contiennent de 14 à 22 variables (avec *CELAR6-SUBi* qui contient *CELAR6-SUBi-1*), des domaines contenant au plus 44 valeurs et plusieurs centaines de contraintes de coût $\{1, 10, 100, 1000\}$. Le tableau suivant montre les temps de résolution sur un *SUN Sparc5* avec ou sans ajout de *MAC3* pour différentes simplifications $\Phi_e^\otimes$ de ces sous-problèmes. *nbvar* indique le nombre de variables ; ub_{init} est une borne initiale fournie à l'algorithme ; *opt* correspond à l'optimum de la simplification $\Phi_e^\otimes$; les minorants, $\Psi_e^\otimes(opt)$, et majorants, $\varphi(opt, \textit{CELAR6-SUBi})$, déduits des simplifications sont aussi indiqués, ainsi que l'optimum du sous-problème, $\varphi(\textit{CELAR6-SUBi})$; en dernière colonne figure le temps de résolution (en minutes) :

nbvar	**algorithme**	$\otimes$	e	ub_{init}	opt	$\Psi_e^\otimes(opt)$	$\varphi(opt, P)$	$\varphi(P)$	**temps**
14	RDS	Σ	1	2669	2669	2669	2669	2669	900 min
"	"	"	100	2400	2400	2400	?	"	464 min
"	"	*lex*	"	$\{1000\}$	$\{100^{24}\}$	1000	2681	"	205 min
"	RDS+MAC3	"	"	"	"	"	"	"	28 min
16	"	"	"	"	"	"	2746	2746	35 min
18	"	"	"	"	$\{100^{27}\}$	"	3091	3005	119 min
22	"	"	"	"	$\{100^{28}\}$	"	3242	3230	596 min

Nous observons que la simplification par relaxation des contraintes de coût inférieur à 100 puis la simplification en lexicographique accélèrent chacune d'un facteur 2 environ la résolution. En revanche, l'ajout du maintien de cohérence locale en lexicographique sur les contraintes de coût 1000 accélère encore d'un facteur 8. Ces gains importants permettent de résoudre des sous-problèmes avec plus de variables : il faut presque 10 heures pour résoudre la simplification à 22 variables, alors que le même sous-problème non simplifié nécessite environ 40 jours de calcul (cf. section 5.2.1.4). D'autre part, les majorants déduits des simplifications sont très proches des optima. En revanche, les minorants déduits des simplifications en lexicographique ne dépassent pas 1000, à cause de la fonction de transfert Ψ^{lex}.

En pratique, l'ajout de *MAC3* améliore les résolutions des simplifications lexicographiques lorsque les premiers niveaux de contraintes sont satisfiables car ces contraintes deviennent impératives et participent à l'élimination de valeurs réalisée par *MAC3*, ce qui favorise aussi le calcul du minorant de la valuation partielle. L'ajout de *MAC3* peut aussi améliorer les résolutions en additif lorsque l'optimum est très faible (inférieur aux coûts des contraintes les plus prioritaires).

Apprentissage et problèmes dynamiques

Jusqu'à présent, les seules informations échangées d'une simplification à l'autre sont les bornes initiales de la valuation des instanciations. Le gain produit par ces informations est relativement limité. La figure 2.17 montre que la connaissance d'un bon majorant (valuation α supérieure à 20) ne permet pas, sur cet exemple, de gagner plus d'un facteur 2 sur le nombre de noeuds développés.

[24] Ces sous-problèmes proviennent de la décomposition présentée dans la section 5.2. Il est intéressant de noter que la combinatoire de l'instance 6 complète est très grande (100 variables, domaines de 44). Ce qui fait que même une résolution en satisfaction peut être très longue. Nous avons testé l'algorithme *MAC7* [RB95] sur l'instance 6, en ayant auparavant retiré les contraintes de coût 10 et 1. L'algorithme ne termine pas au bout d'un week-end de calcul sur SUN UltraSparc. Or, ce même problème résolu en optimisation risque d'être bien plus dur... C'est pourquoi nous limitons nos expérimentations à des sous-problèmes.

Une autre forme de réutilisation des raisonnements, pouvant offrir des gains plus élevés, repose sur la notion d'apprentissage des instanciations partielles qui ne conduisent pas à de bonnes solutions. C'est ce que propose l'apprentissage par mémorisation de contraintes induites, appelées *Nogoods* (cf. section 2.3.1.4). Cette technique s'applique aussi bien pour résoudre un seul problème, que pour résoudre une succession de problèmes proches, dits dynamiques (cf. section 2.1.3.2). La relaxation par niveau des contraintes implique uniquement un ajout de contraintes d'un problème à l'autre. Ainsi, tous les *Nogoods* appris lors de la résolution d'un niveau sont réutilisables au niveau suivant. Cela devrait permettre des gains intéressants lorsque l'opérateur d'agrégation est idempotent [VS94], et peut-être aussi dans le cas non-idempotent [Dag97].

Dans le cas lexicographique, il semble possible de déduire des *Nogoods* identiques à ceux du cadre de la satisfaction et donc plus simples à gérer que dans le cas additif. Pour cela, il faudrait retenir les instanciations partielles A telles que leurs valuations globales $\varphi(A, \Phi^{lex}_{e_i}(P))$ soient strictement supérieures à l'optimum du problème simplifié $\Phi^{lex}_{e_i}(P)$. Ainsi, ces instanciations seront aussi interdites dans la prochaine simplification du problème, $\Phi^{lex}_{e_{i+1}}(P)$.

Finalement, une idée à approfondir consiste à exploiter le fait que l'arbre de recherche d'un problème simplifié par une simplification *agrégation compatible* est inclus dans l'arbre de recherche du problème non simplifié (cf. théorème 9, uniquement valable dans le cas d'ordres statiques). Il s'agit d'éviter de reparcourir plusieurs fois les mêmes chemins ne conduisant pas à des solutions.

Chapitre 5

Simplifications structurelles par relaxation des contraintes

Une approche courante pour simplifier un problème consiste à relaxer ses contraintes. L'optimum du problème relaxé est directement un minorant du problème cible. Nous présentons dans la section 5.1 un panorama des techniques potentielles de simplifications par relaxation des contraintes. Nous décrivons plus en détail deux techniques de relaxation dans les sections 5.2 et 5.3. Dans ce chapitre, nous traitons uniquement des problèmes additifs Σ-VCSP.

5.1 Aperçu des techniques potentielles pour relaxer des contraintes

La relaxation des contraintes peut s'exprimer de trois manières différentes :

- par suppression de certaines contraintes ; ainsi la connectivité du graphe des contraintes diminue ;
- par ajout de n-uplets autorisés à l'intérieur de certaines contraintes ; ainsi la dureté de ces contraintes diminue ;
- par suppression de la contrainte d'intégrité sur les domaines ; c-à-d que les variables peuvent être affectées à des valeurs réelles comprises dans l'intervalle défini par la plus petite et la plus grande valeur discrète de leurs domaines ; nous pouvons alors résoudre le problème relaxé par un algorithme issu de la *Programmation Linéaire* [NW88].

Quelle que soit la relaxation faite, l'objectif consiste à obtenir un problème plus simple à résoudre et qui produise un minorant proche de l'optimum. Les relaxations visent une stratégie de résolution particulière, c-à-d un certain type de simplification du problème cible en sous-problèmes suivi d'une certaine méthode complète de résolution appliquée aux sous-problèmes. Dans certains cas, la relaxation ne fait pas d'hypothèse sur la méthode complète utilisée. C'est le cas lorsque la taille des sous-problèmes diminue : une diminution du nombre de variables à traiter en même temps, ou une diminution du nombre de valeurs possibles implique une diminution de la taille de l'arbre de recherche complet (sans coupes), quelle que soit la méthode complète.

Nous donnons dans les deux sections suivantes des stratégies de résolution en précisant le type de simplification visé, si nécessaire la méthode complète visée, les contraintes qui seront relaxées, le calcul du minorant et des pistes pour réaliser ces simplifications. La première section traite des relaxations par suppression de contraintes et la deuxième des relaxations par ajout de n-uplets. La relaxation de la contrainte d'intégrité dans le but d'utiliser les techniques de la *Programmation Linéaire* est abordée dans le chapitre 6 section 6.1.2.2.

5.1.1 Relaxation par suppression de contraintes

Nous donnons une liste de stratégies de résolution fondées sur la suppression de contraintes :

- supprimer les contraintes de faible coût pour ne s'intéresser qu'aux contraintes les plus importantes ; cette technique simple est étudiée dans le chapitre 4 (simplification notée Φ_e) ; elle permet de réduire la connectivité, donc la difficulté de résolution, tout en produisant des bons minorants ; l'optimum du problème simplifié est directement un minorant de l'optimum du problème cible ;

- séparer arbitrairement les variables en composantes connexes disjointes et résoudre séparément chaque sous-problème ; il faut supprimer les contraintes inter-composantes ; la somme des optima des sous-problèmes est un minorant ; nous donnons quelques pistes pour séparer les variables en s'appuyant sur la structure du graphe des contraintes :

  - recherche d'une partition des variables en un nombre minimum de **cliques** [1] ; c'est un problème NP-complet [CK95] ; il existe un schéma d'approximation polynomial[2] sous $O(|V|\frac{(\log\log|V|)^2}{(\log|V|)^3})$ [CK95], avec $|V|$, le nombre de variables du problème, voir aussi [GHHY96] ;
  - recherche d'une partition des variables en composantes fortement connectées entre elles par des contraintes (mais pas forcément des cliques), que nous appellerons des **grappes** ; nous présentons dans la section 5.2.2 une heuristique fondée sur la largeur des variables (cf. section 2.3.1.3) ;
  - recherche d'une partition des variables qui minimise la somme des coûts des contraintes inter-composantes ; nous présentons dans la section 5.2.1 une méthode de recherche locale pour résoudre ce problème NP-complet de *K-partition équilibrée de graphe* [PS89, HN89] ;
  - recherche d'un sous-ensemble des variables souvent impliquées dans la violation des contraintes ; une décomposition en composantes connexes peut alors être appliquée à ce sous-ensemble ; le reste des variables et leurs contraintes associées pouvant être ignorés ; [MGS96] propose une technique pour déterminer ce sous-ensemble, dit *noyau dur*, en exécutant un algorithme de recherche locale sur le problème cible et en comptabilisant le nombre de fois où les contraintes sont violées au cours des itérations ; les variables liées aux contraintes les plus souvent violées constituent le *noyau dur* ;
  - recherche de **ponts** , contraintes dont la suppression rend le graphe non connexe [CLR94] ; cela peut se faire en temps linéaire et permettrait d'augmenter le nombre de composantes connexes préexistantes dans le graphe (obtenues aussi en temps linéaire [Tsa93]) ;

[1] Une clique dans un VCSP binaire est un sous-ensemble de variables entièrement reliées entre elles par des contraintes (connectivité égale à 100%).

[2] Un schéma d'approximation polynomial sous $O(n)$ signifie que la mesure de la solution, notée m, est au pire à une distance bornée de l'optimum, noté opt, telle que $max\{\frac{m}{opt}, \frac{opt}{m}\} \leq O(n)$.

- décomposer les contraintes en sous-ensembles disjoints générant des sous-problèmes qui puissent être résolus en temps polynomial, comme par exemple les arbres, résolus par une méthode de programmation dynamique [FW92] ; dans cette approche, il n'y a pas de contraintes à supprimer ; les sous-problèmes peuvent avoir des variables en commun ; la somme des optima des sous-problèmes est un minorant (car les contraintes n'interviennent que dans un seul problème à la fois) ; nous pouvons envisager deux techniques de décomposition, valables pour les VCSP binaires[3] :
  - recherche d'un ensemble *coupe-cycle* minimal[4] ; c'est un problème NP-dur [AS94] ; il existe des heuristiques pour y trouver des solution approchées [Tsa93] ; les contraintes liées au *coupe-cycle* sont soit supprimées, soit isolées ; la même décomposition est alors appliquée récursivement dessus ; cette technique de décomposition par ensemble *coupe-cycle* existe aussi comme méthode de résolution exacte [DP87] ;
  - recherche d'un arbre couvrant dont la somme des coûts des contraintes qui forment l'arbre soit maximale ; c'est un problème qui est résolu par un algorithme glouton en $O(|C| \cdot \log(|V|))$ [CLR94], avec $|V|$, le nombre de variables et $|C|$, le nombre de contraintes ; nous appliquons récursivement la recherche d'un arbre couvrant maximal sur les composantes connexes formées par les contraintes restantes ;
- minimiser la largeur de bande du graphe des contraintes ; une plus faible largeur de bande pour une méthode de résolution utilisant un ordre statique sur les variables diminue le nombre de retour-arrière [Tsa93] ; cela a été observé expérimentalement sur l'algorithme *Russian Doll Search* dans [LV96] ; une technique consisterait à chercher par une méthode gloutonne un ordre de faible largeur de bande [Tsa93] ; puis à supprimer les contraintes de manière à diminuer la largeur de bande de cet ordre ; l'optimum du problème simplifié est directement un minorant.

5.1.2 Relaxation par ajout de n-uplets

Nous donnons une liste de stratégies de résolution fondées sur l'ajout de n-uplets autorisés à l'intérieur des contraintes :

- réduire les domaines en regroupant des valeurs similaires, c-à-d presque interchangeables[5] ; pour un VCSP binaire, il faut éventuellement ajouter des couples autorisées à l'intérieur des contraintes pour obtenir une propriété d'interchangeabilité [Fre91] ; les solutions optimales du problème après regroupement des valeurs restent optimales pour le problème relaxé avant regroupement ; ainsi, l'optimum du problème simplifié est un minorant de l'optimum du problème cible ; la recherche des valeurs similaires peut se faire par un algorithme glouton (cf. section 5.3.2) ; plusieurs propriétés moins strictes ou plus faciles à détecter que l'interchangeabilité peuvent être considérées et étendues au cadre des VCSP :
  - la voisinage-interchangeabilité : l'ensemble des supports de a est égal à l'ensemble des supports de b ; $a, b \in D(x), \forall c = (v, r) \in C / x \in v$, $\{i/(a, i) \in r\} = \{i/(b, i) \in r\}$;

[3] Dans le cadre des VCSP naires, ces techniques doivent être étendues aux hyper-arbres.

[4] Un ensemble coupe-cycle est un ensemble de sommets qui une fois retirés rendent le graphe acyclique.

[5] Dans un CSP, deux valeurs sont interchangeables si pour toute solution cohérente contenant la première valeur, alors le fait de remplacer cette première valeur par la deuxième conserve la solution cohérente et *vice versa*. Dans un VCSP, cette définition peut s'étendre en remplaçant un solution cohérente par une solution optimale (cf. section 5.3.1).

  - la k-interchangeabilité : deux valeurs a et b, associées à la variable v, sont interchangeables quel que soit le sous-problème constitué par v et $k-1$ autres variables ;
  - la substituabilité : l'ensemble des supports de a inclus l'ensemble des supports de b ; $a, b \in D(x), \forall c = (v, r) \in C / x \in v, \{i/(a, i) \in r\} \supseteq \{i/(b, i) \in r\}$;

- réduire les domaines en regroupant des valeurs interchangeables par permutation[6] ; par exemple, une détection des symétries sur les solutions d'un problème permet de le simplifier[7] [Ben94] ;

- modifier les contraintes pour les rendre linéaires, c-à-d vérifiant une fonction linéaire entre variables pour assurer une meilleure conversion du problème dans le cadre de la *Programmation Linéaire* ; cela évite de décomposer les contraintes du VCSP en autant de contraintes linéaires que de n-uplets autorisés (cf. section 6.1.2.2) ; cette stratégie de résolution combine donc deux relaxations : ajout de n-uplets autorisés puis relaxation de la contrainte d'intégrité.

Remarque 33 La structure interne des contraintes a été beaucoup étudiée dans le cadre de la satisfaction [Dv95, GR96]. L'objectif consistait à exhiber des classes polynomiales. Cependant, il semble que ces contraintes aux structures internes particulières n'impliquent pas l'appartenance à une classe polynomiale dans le cadre de l'optimisation [Lau96, Ist97].

◇

5.1.3 Bilan

Nous avons aperçu que certaines techniques de simplification proposées constituent des problèmes NP-difficiles. Néanmoins, une solution approchée à ces problèmes est suffisante. Elle est construite par une méthode incomplète, une recherche heuristique ou stochastique, qui prend généralement peu de temps comparé à celui pris par la méthode complète pour résoudre le problème simplifié. Ces techniques de simplification sont donc un bon exemple de coopération de méthodes complètes et incomplètes. Notons que les stratégies de résolution par relaxation de la contrainte d'intégrité et par décomposition des contraintes en sous-ensembles sont les seules stratégies qui visent une méthode de résolution polynomiale (sur un système d'inéquations linéaires, ou sur des arbres), les autres impliquent la résolution d'un ou de plusieurs sous-problèmes NP-durs.

Il est très difficile de mesurer la pertinence de ces stratégies de résolution sans faire des expérimentations sur plusieurs benchmarks. En effet, la qualité des minorants déduits n'est pas connue à l'avance. Le seul critère évident est l'importance de la relaxation, en terme de coût des contraintes supprimées, ou des n-uplets ajoutés. Ce coût permet de borner la distance maximale du minorant à l'optimum : $\varphi(P) - \varphi(\Phi(P)) \leq \sum_{c \in C_{relaxe}} \varphi(c)$, avec $\varphi(P)$, l'optimum du

[6] Soit p, une permutation des valeurs telle que $p(a) = b$. Dans un CSP, deux valeurs sont interchangeables par permutation si pour toute solution cohérente contenant a, alors l'application de la permutation sur les valeurs de cette solution produit une nouvelle solution cohérente. Dans un VCSP, deux valeurs sont interchangeables par permutation si cette permutation appliquée à toutes les valeurs d'une instanciation complète conserve la valuation de cette instanciation complète, et ceci pour toutes les instanciations complètes où figurent les deux valeurs interchangeables.

[7] Nous avons observé une symétrie centrale sur les contraintes d'un sous-problème de l'instance 6 du CELAR, noté *CELAR6-SUB4* (cf. section 5.2.1.4). Sur ce problème, la permutation $p : f \mapsto 808 - f$ conserve la valuation des solutions. Cela permet de diviser par deux le temps de résolution d'une méthode complète en ne parcourant que la première moitié des valeurs du domaine de la première variable (cf. symétrie sur le nombre de noeuds développés pour une valeur de la première variable, dans la figure 5.1).

problème cible, $\varphi(\Phi(P))$, l'optimum du problème simplifié et C_{relaxe}, toutes les contraintes qui ont été relaxées. En pratique, le coût des relaxations est souvent bien plus grand que l'optimum du problème et pourtant le minorant peut être assez proche de l'optimum.

Pour ce qui est du gain en temps produit par une relaxation, un critère simple est de comparer la taille de l'arbre de recherche complet (égale à l'espace des combinaisons de valeurs affectées aux variables). Ceci n'est possible que dans les cas où la simplification diminue le nombre de variables ou de valeurs. C'est précisément deux de ces cas qui sont décrits dans les sections suivantes.

5.2 Décomposition des variables en grappes

L'étude expérimentale menée sur des Σ-VCSP binaires aléatoires dans la section 4.2.2 a montré que la difficulté de la résolution croît lorsque la connectivité et la dureté augmentent (cf. figure 4.8). Cette croissance est accompagnée d'une augmentation de l'optimum des problèmes aléatoires. Nous en déduisons un premier critère pour séparer les variables en plusieurs paquets, appelés grappes : le minorant déduit d'une grappe devrait être meilleur si la connectivité et la dureté de la grappe sont fortes. Un deuxième critère consiste à rechercher des grappes qui ont une somme des coûts de leurs contraintes importante. La combinaison de ces deux critères devrait favoriser l'obtention de bons minorants.

En contrepartie, les grappes ne doivent pas contenir trop de variables, sinon les temps de résolution par une méthode complète risquent d'être démesurés. En fonction de la taille des domaines, de la distribution des coûts des contraintes et de la structure du graphe des contraintes, le nombre maximum de variables par grappe varie. C'est pourquoi, une approche itérative croissante de ce nombre permet une progression continue du temps de résolution.

5.2.1 Décomposition par une recherche locale

5.2.1.1 Problème de K-partition

Nous exprimons le problème de la recherche de grappes sous forme d'un problème de K-partition équilibrée de graphe [PS89, HN89] :

Définition 28 ——————————————————————————

K-partition équilibrée de graphe *[Cab96] :*
étant donné un graphe non orienté $G = (S, A)$ composé de N sommets S et M arêtes A pondérées par un coût $c_i, i \in \{1, M\}$, la K-partition équilibré vise à découper ce graphe en K sous-graphes distincts tels que la somme des coûts des arêtes d'interconnexion entre les K sous-graphes soit minimum, et tels que l'écart entre les tailles des K sous-graphes soit minimum (le cas idéal correspondant à N/K sommets par sous-graphes).

Un codage classique, sous la forme d'un problème d'optimisation discrète non linéaire multi-critères, consiste à associer à chaque sommet une variable vectorielle $\vec{\sigma}_i \in \{0,1\}^K$ avec la convention suivante :

$$\begin{cases} \sigma_{ik} = 1 & \textit{si le sommet indexé par i appartient au sous-graphe indexé par k} \\ \sigma_{ik} = 0 & \textit{sinon} \end{cases} \tag{5.1}$$

Un sommet ne pouvant appartenir qu'à un seul sous-graphe, une contrainte d'unicité est ajoutée au problème :

$$\forall i \in \{1 \ldots N\}, \quad \sum_{k=1}^{K} \sigma_{ik} = 1 \tag{5.2}$$

Soit T_{ij}, un matrice décrivant la présence d'arêtes entre deux sommets : $T_{ij} = T_{ji}$ et $T_{ij} = 1$ s'il y a une arête entre le sommet indexé i et le sommet indexé j, sinon $T_{ij} = 0$. La fonction objectif à minimiser, notée E, est la somme, pondérée par un facteur h, d'un coût d'interconnexion[8] *et d'un coût de déséquilibre :*

$$E = -\sum_{k=1}^{K} \sum_{i=1}^{N} \sum_{j>i}^{N} T_{ij} \cdot \sigma_{ik}\sigma_{jk} + h \sum_{k=1}^{K} \left(\sum_{i=1}^{N} \sigma_{ik} - \frac{N}{K} \right)^2 \tag{5.3}$$

Nous résolvons ce problème de K-partition par une méthode incomplète. De manière à réutiliser les mêmes algorithmes incomplets développés pour résoudre des VCSP, nous proposons de transformer un problème Σ-VCSP P en un autre problème Σ-VCSP P', tel que P' corresponde au problème de la K-partition de P. Néanmoins, la contrainte de déséquilibre, ainsi que l'optimisation multi-critères ne sont pas spécifiées dans P' et seront ajoutés directement au sein de la méthode incomplète.

5.2.1.2 Transformation d'un problème de K-partition en un problème Σ-VCSP

Soit le problème Σ-VCSP $P = (V, D, C, S, \varphi)$ auquel nous voulons appliquer une K-partition. Alors, le problème Σ-VCSP $P' = (V', D', C', S', \varphi')$ exprimant partiellement[9] la K-partition est défini par :

- $V' = V$, l'ensemble des variables reste inchangé ;
- $D' = \{1 \ldots K\}^{|V|}$, les domaines sont tous identiques, la valeur d'une variable indiquant le numéro de la partition dans laquelle est rangée la variable ;
- $C' = \{c' = (v_i, r'_i) / c = (v_i, r_i) \in C \wedge r'_i = \{1^{|v_i|} \ldots K^{|v_i|}\}\}$, les contraintes de P deviennent des contraintes d'égalité ;
- $S' = S$, la structure de valuation reste additive ;
- $\varphi'(c') = \varphi(c)$, les coûts des contraintes restes identiques.

La valuation d'une instanciation complète A dans P' est égale au coût d'interconnexion de la K-partition déduite de A.

5.2.1.3 Résolution par un algorithme de recuit simulé

Nous supposons que l'algorithme de recuit simulé est celui présenté dans la section 2.3.2.1. Pour prendre en compte le terme de déséquilibre, il faut ajouter à cet algorithme une structure de donnée mémorisant le nombre de variables par partition. Ce tableau de K entiers est noté *partsz*[].

Soit A, l'instanciation complète, générée aléatoirement au début de l'algorithme. A chaque itération de l'algorithme, des modifications locales, correspondant à changer la valeur i en j d'une variable x, sont testées et acceptées suivant la probabilité d'acceptation égale à :

[8] Dans la formule, le coût d'interconnexion est en fait représenté par l'opposé de la somme des coûts des contraintes à l'intérieur des sous-graphes.

[9] La contrainte de déséquilibre, ainsi que l'optimisation multi-critères ne sont pas spécifiées dans P'.

$$\begin{cases} 1 & \text{si } E(A \cup \{x \leftarrow j\}) \leq E(A \cup \{x \leftarrow i\}) \\ e^{-\frac{E(A\cup\{x\leftarrow j\})-E(A\cup\{x\leftarrow i\})}{T}} & \text{sinon} \end{cases} \tag{5.4}$$

Un schéma de décroissance est appliqué à la température T. La fonction énergie E d'une instanciation complète est définie à partir de la fonction objectif de l'équation 5.3 :

$$E(A) = \varphi'(A, P') + h \sum_{k=1}^{K} \left(partsz[k] - \frac{N}{K} \right)^2 \tag{5.5}$$

Puisque c'est seulement la variation de l'énergie qui est examinée, nous pouvons simplifier son calcul, en notant que seul $partsz[i]$ et $partsz[j]$ ont été modifiés par le *flip* (le calcul utilise les valeurs de $partsz[]$ avant sa mise à jour) :

$$\begin{aligned}
E(A \cup \{x \leftarrow j\}) - E(A \cup \{x \leftarrow i\}) &= \varphi'(A \cup \{x \leftarrow j\}, P') - \varphi'(A \cup \{x \leftarrow i\}, P') \\
&\quad + h \sum_{k=1}^{K} \left(\left(partsz_{\{x \leftarrow j\}}[k] - \frac{N}{K} \right)^2 - \left(partsz_{\{x \leftarrow i\}}[k] - \frac{N}{K} \right)^2 \right) \\
&= \varphi'(A \cup \{x \leftarrow j\}, P') - \varphi'(A \cup \{x \leftarrow i\}, P') \\
+h \Big(\big((partsz[i] - 1) - \tfrac{N}{K} \big)^2 - \big(partsz[i] - \tfrac{N}{K} \big)^2 &+ \big((partsz[j] + 1) - \tfrac{N}{K} \big)^2 - \big(partsz[j] - \tfrac{N}{K} \big)^2 \Big) \\
&= \varphi'(A \cup \{x \leftarrow j\}, P') - \varphi'(A \cup \{x \leftarrow i\}, P') \\
+h \Big(\big((partsz[i] - \tfrac{N}{K}) - 1 \big)^2 - \big(partsz[i] - \tfrac{N}{K} \big)^2 &+ \big((partsz[j] - \tfrac{N}{K}) + 1 \big)^2 - \big(partsz[j] - \tfrac{N}{K} \big)^2 \Big) \\
&= \varphi'(A \cup \{x \leftarrow j\}, P') - \varphi'(A \cup \{x \leftarrow i\}, P') \\
&\quad + h \left(1 - 2(partsz[i] - \frac{N}{K}) + 1 + 2(partsz[j] - \frac{N}{K}) \right) \\
&= \varphi'(A \cup \{x \leftarrow j\}, P') - \varphi'(A \cup \{x \leftarrow i\}, P') \\
&\quad + 2h \left(partsz[j] - partsz[i] + 1 \right) \\
&= \varphi'(A \cup \{x \leftarrow j\}, P') - \varphi'(A \cup \{x \leftarrow i\}, P') \\
&\quad + H \left(partsz[j] - partsz[i] + 1 \right) \\
&\quad \text{avec } H = 2h
\end{aligned}$$

5.2.1.4 Expérimentations sur l'instance 6 du CELAR

Nous avons appliqué cet algorithme de recuit simulé modifié pour effectuer une 6-partition de l'instance 6 du CELAR. Le choix de $K = 6$ implique des grappes de 16 variables environ. Pour des domaines de 44 fréquences, la résolution d'un sous-problème du CELAR à 16 variables correspond à peu près à la limite accessible dans une journée par une méthode complète de type *Russian Doll Search*. Les résultats présentés dans cette section ont été obtenus sur une station *SUN UltraSparc*, à partir de notre librairie de méthodes de simplification et de résolution écrite en *C++*.

Pour un facteur de pondération $h = 100$, et des paramètres de recuit tels que $l = 1000, g = 50, m = 0.9$, il fallu 80 secondes pour trouver une décomposition de coût d'interconnexion égal à 16725 (à comparer avec la somme des coûts des contraintes conservées, égale à 238468) et de coût de déséquilibre égal à 2933. Le tableau suivant montre cette décomposition, en donnant pour chaque numéro de grappe, le nombre de variables, la somme des coûts des contraintes à l'intérieur de la grappe et la liste des variables composant la grappe :

k	**nbvar**	$\sum_{c \in part[k]} \varphi(c)$	**liste des variables**
1	21	73709	233 235 237 303 305 309 329 349 351 471 517 561 577 579 591 605 621 623 625 633 801
2	16	11255	53 79 81 215 263 287 289 325 327 359 365 651 743 747 763 879
3	15	48693	143 145 273 275 277 281 283 285 341 343 713 717 719 721 723
4	15	27874	269 271 291 555 583 607 645 661 787 789 791 793 795 797 799
5	18	47785	13 15 83 85 321 323 519 557 593 597 599 613 629 631 665 765 767 785
6	15	29152	261 265 267 559 565 585 587 589 595 601 603 617 635 663 745

Une fois la 6-partition effectuée, et avant la résolution par une méthode complète, nous cherchons un bon majorant pour chaque grappe à l'aide d'un recuit simulé classique. Or les grappes ne sont pas toutes connexes. Nous appliquons donc une décomposition en composantes connexes puis exécutons l'algorithme du recuit simulé sur chaque composante ($l = 1000, g = 50, m = 0.9$). Le tableau suivant résume les différentes composantes avec leurs meilleurs majorants trouvés par la méthode incomplète ainsi que les temps en secondes consommés par cette méthode :

k'	**nbvar**	**nbcontr**	**majorant**	**temps du recuit (sec)**
1	18	229	159	37.5
2	6	10	0	0.1
3	14	300	2737	280.4
4	9	76	0	3.7
5	21	188	0	8.6
6	14	91	0	2.3
7	13	78	110	103
8	1	0	0	0
9	1	0	0	0
10	2	4	0	0.1
11	1	0	0	0

Il n'y a donc que les composantes, k' = 1,3 et 7, qui nécessitent une résolution complète. Ces composantes proviennent des grappes k = 5, 3 et 4. Les deux premières composantes sont résolues par un algorithme *depth-first branch and bound* avec *forward checking* et *directed arc consistency counts* (voir $lb_{dac}(A, P)$ section 2.3.1.2). Les variables sont triées par degré décroissant (heuristique *maximum degree*). L'optimalité des majorants est prouvé en 259.5 secondes pour la composante 1 et 0.2 secondes pour la composante 7. En revanche, la composante 3 nécessite une autre méthode complète pour la résoudre avec preuve d'optimalité. Nous utilisons l'algorithme *Russian Doll Search* [LV96] qui met 459 minutes pour prouver l'optimum à 2269 ; cet optimum ayant précédemment été trouvé par une série de recuits simulés. Finalement, le minorant obtenu sur l'instance 6 à l'aide de la K-partition est égal à 2938, $= 159 + 2669 + 110$. Pour comparaison, le meilleur majorant connu pour cette instance vaut 3389 [Kol96]. Ainsi, la qualité normée approchée vaut $(255194 - 3389)/(255194 - 2938) = 99.82\%$ (cf. equation 3.3 section 3.1.2).

Pour tenter d'améliorer ce minorant, nous avons exécuté d'autres 6-partitions avec des facteurs de pondération plus faible (h variant entre 10 et 50), de manière à détecter des grappes plus intéressantes mais aussi plus dures à résoudre. Une série de sous-problèmes de l'instance 6 du CELAR a donc été générée à partir des grappes dont l'optimum était suffisamment élevé. Tous les sous-problèmes sont une extension de la composante $k' = 3$. Le premier sous-problème est noté

CELAR6-SUB1, il est identique à cette composante. Ensuite, les trois autres sous-problèmes rajoutent des variables, tel que *CELAR-SUBi+1* inclus le sous-problème *CELAR-SUBi*. Le sous-problème *CELAR-SUB0* est différemment construit : il provient de la composante $k' = 1$ avec les variables 83 et 85 en moins. Les deux tableaux suivants décrivent ces sous-problèmes qui ont été résolus par un algorithme *Russian Doll Search*. De manière à calculer leurs connectivités, le deuxième tableau présente les sous-problèmes sans la simplification qui regroupe les paires de variables (cf. annexe A.3) :

CELAR6-SUB*i*	**nbvar**	**nbcontr**	**liste des variables**
CELAR6-SUB0	16	207	13 15 321 323 519 557 593 597 599 613 629 631 665 765 767 785
CELAR6-SUB1	14	300	143 145 273 275 277 281 283 341 343 713 717 719 721 723
CELAR6-SUB2	16	353	143 145 273 275 277 281 283 341 343 555 583 713 717 719 721 723
CELAR6-SUB3	18	421	143 145 269 271 273 275 277 281 283 341 343 555 583 713 717 719 721 723
CELAR6-SUB4	22	477	143 145 269 271 273 275 277 281 283 291 341 343 555 583 607 713 717 719 721 723 787 789

CELAR6-SUB*i*	**nbvar**	**nbcontr**	**connex**	**opt**	**temps**
CELAR6-SUB0	32	223	0.4697	159	107 secondes
CELAR6-SUB1	28	314	0.8306	2669	430 minutes
CELAR6-SUB2	32	369	0.7439	2746	289 minutes
CELAR6-SUB3	36	439	0.6968	3079	$\simeq$ 18 jours[10]
CELAR6-SUB4	44	499	0.5274	3230	$\simeq$ 26 jours[10]
CELAR6	200	1322	0.0664	3389	-

Les sous-problèmes *CELAR6-SUB0* et *CELAR6-SUB4* n'ont aucune variable en commun et donc ne partagent pas de contrainte. La somme de leurs optima est un minorant de l'instance 6 : $159 + 3230 = 3389$. Or, le meilleur majorant connu sur ce problème vaut aussi 3389 [Kol96]. Notre décomposition a donc permis de prouver l'optimum de l'instance 6 du CELAR, qui était auparavant un problème ouvert [dGV97a].

Les sous-problèmes *CELAR6-SUBi* ont été mis accessibles par *ftp* à l'adresse :
ftp://ftp.cert.fr/pub/degivry/SUBCELAR.dir/
Ils représentent des instances particulièrement difficiles permettant de tester des nouvelles méthodes de résolution [LL98]. Ils constituent donc un nouveau benchmark, plus accessible que les instances initiales du CELAR [CdGL+98].

[10] La résolution de *CELAR6-SUB3* et *CELAR6-SUB4* a été faite par un algorithme *Russian Doll Search* amélioré et parallélisé. L'amélioration porte sur le calcul du minorant de la valuation globale d'une instanciation partielle. Cela consiste à prendre en compte à la fois le calcul du *RDS* et celui du *DAC* statique : $lb(A,P) = \max\{lb_{dac}(A,P), lb_{rds}(A,P)\}$. Le gain en temps produit par ce nouveau lb est néanmoins assez faible, environ 5%. Une seconde amélioration plus profitable vient de la parallélisation. L'approche parallèle est simple : à chaque niveau du *RDS*, nous répartissons sur autant de stations de travail (à l'aide d'un réseau *PVM*) qu'il y a de valeurs dans le domaine de la première variable du sous-problème emboîté à résoudre ; ce sous-problème ayant sa première variable affectée à la valeur associée à la station. Cette approche permet de mémoriser non pas l'optimum de n sous-problèmes, mais l'optimum de $n \cdot n$ sous-problèmes dont la première variable est affectée. C'est donc une information plus riche qui est utilisée par le calcul de $lb(A,P)$. La figure 5.1 montre le nombre de noeuds développés au dernier niveau du *RDS* en fonction de la valeur du domaine de la première variable du sous-problème *CELAR6-SUB4*. En moyenne, l'approche parallèle a permis de gagner un facteur 2 en temps de résolution. Un réseau de 40 stations a résolu les deux sous-problèmes en un week-end chacun.

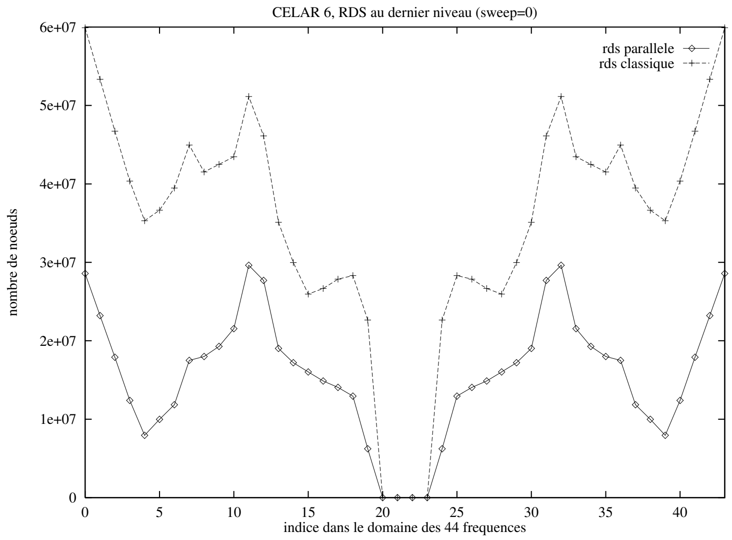


Fig. 5.1 - Nombre de noeuds développés par l'algorithme *Russian Doll Search* au dernier niveau de *CELAR6-SUB4*, en fonction de la valeur affectée à la première variable, avec utilisation de $(n-1)\cdot(n-1)$ optima pour la version parallélisée, et seulement $n-1$ optima pour la version classique

Par ailleurs, il serait intéressant d'examiner une autre approche de la décomposition qui soit plus hiérarchique. Il s'agirait d'utiliser récursivement une bi-partition jusqu'à obtenir des grappes suffisamment petites, tout en favorisant le critère d'interconnexion minimum lors des premières bi-partitions.

5.2.2 Décomposition suivant une heuristique

Nous présentons un algorithme glouton paramétrable fondé sur l'heuristique de cardinalité maximum dont le but consiste à détecter des grappes fortement connexes au sein du graphe du problème cible. Bien que ne prenant presque pas en compte le coût des contraintes, nous montrons l'efficacité de cet algorithme sur des problèmes SPOT5, ainsi que sur l'instance 6 du CELAR.

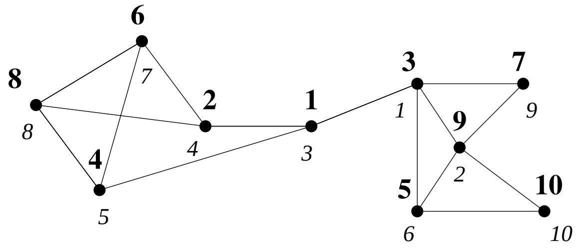
5.2.2.1 Heuristique de cardinalité maximum

L'heuristique de cardinalité maximum, décrite dans [Tsa93] (*max-cardinality ordering*, notée MCO), peut être vue comme une heuristique assez grossière minimisant la largeur d'un graphe (cf. section 2.3.1.3). MCO est également utilisée dans une méthode complète de résolution de CSP par décomposition en arbres [Tsa93]. Son pseudo-code est le suivant :

```
MCO( V, C )
   N := Taille( V )
   ordre[N] := PremierElement( V )
   V := V - ordre[N]
   pour i = N-1 a 1 faire
         ordre[i] := CardinaliteMax( V, ordre, i+1, N )
         V := V - ordre[i]
```

fin pour
retourne ordre

Soit P = (V, D, C, S, φ), alors MCO(V,C) retourne les variables ordonnées dans un tableau, noté ordre, tel que la largeur de l'ordre soit *a priori* faible. La fonction CardinaliteMax retourne la variable de V qui a la plus grande cardinalité, c-à-d qui a le plus grand nombre de contraintes connectées avec les variables déjà insérées dans ordre, l'ensemble {ordre[i+1], ..., ordre[N]}. Lorsqu'il y a plusieurs variables de même cardinalité, nous choisissons la première dans l'ordre initial des variables. Ceci permettra de combiner MCO avec d'autres ordres sur les variables.



cardinalités : 2 2 2 1 3 2 1 1 1 0
Ordre de cardinalité maximum: 10, 7, 9, 5, 8, 6, 4, 3, 2, 1

Ordre de degré maximum : 3, 9, 1, 2, 4, 5, 6, 8, 7, 10

cardinalités : 3 2 1 1 1 2 2 2 1 0
Ordre de degré maximum puis : 8, 6, 4, 2, 1, 10, 7, 5, 9, 3
ordre de cardinalité maximum

Fig. 5.2 - Ordres de degré et de cardinalité maxima

La figure 5.2 montre un graphe comportant 10 variables et 15 contraintes. L'ordre initial des variables est donné par la numérotation en gras, au dessus de chaque sommet. L'ordre produit par MCO est indiqué sur la première ligne, ainsi que la cardinalité (écrite au dessus, dans une fonte plus petite) pour chaque variable. Ici, la largeur de l'ordre vaut 3. Nous avons aussi indiqué l'ordre fourni par une autre heuristique, celle de degré maximum, notée MDO : les variables sont triées suivant leurs nombres de contraintes. Ensuite, nous recalculons l'ordre par MCO à partir, non pas de l'ordre initial, mais de l'ordre de degré maximum. Cette combinaison des deux ordres doit nous permettre de discerner deux grappes constituées des variables $\{8, 6, 4, 2, 1\}$ et des variables $\{10, 7, 5, 9, 3\}$. Nous allons voir comment nous établissons les points de séparation dans l'ordre final MDO + MCO.

5.2.2.2 Algorithme glouton de décomposition

L'algorithme glouton effectuant la décomposition des variables, noté HVD (*Heuristic Variable Decomposition*), est contrôlé par un unique paramètre, Tmax, la taille maximale des grappes. Ainsi, toutes les grappes contiendront au plus Tmax variables. Son pseudo-code est le suivant :

HVD(V, C, Tmax)

```
g := 0
tant que V ≠ ∅
    (ordre, cardinalites) := MCO( MDO(V, C), C )
    i := IndexCardinaliteMax( cardinalites, 1, Taille(V) )
    j := IndexDecroissanceCardinaliteMax( cardinalites, 1, i )
    g := g + 1
    grappes[g] := {ordre[j]}
    k := j + 1
    tant que k ≤ Taille(V) et k < j+Tmax et cardinalites[k-1] > 0
        grappes[g] := grappes[g] ∪ ordre[k]
        k := k + 1
    fin tant que
    pour l = j+1 a k-1 faire
        si PasConnectee(grappes[g], ordre[l]) alors
            grappes[g] := grappes[g] - ordre[l]
        fin si
    fin pour
    V := V - grappes[g]
fin tant que /* j ≤ i et j ≤ k */
retourne g, grappes
```

A chaque itération de l'algorithme une grappe est créée, jusqu'à ce que toutes les variables soient réparties. Lors d'une itération, les variables restantes sont retriées par MDO puis MCO. ordre reçoit les variables triées et cardinalites reçoit les cardinalités de chaque variable dans cet ordre. IndexCardinaliteMax permet de repérer l'index i de la variable ayant la plus grande cardinalité ; en cas d'égalité, c'est la première dans l'ordre qui est retenue. Cet index doit nous permettre de repérer la grappe dont la connexion interne maximale est la plus grande. Ensuite, IndexDecroissanceCardinaliteMax repère l'index j de la variable précédant ordre[i] telle que la différence de cardinalité cardinalites[j] - cardinalites[j-1] soit maximum. En effet une décroissance brusque de la cardinalité signifie que la variable en j-1 n'est plus trop connectée à la grappe que nous cherchons à isoler. Cette décroissance est examinée à partir de l'index i jusqu'à ce que cardinalites[j - 1] ≤ 1[11] ou que j = 1. L'index j constitue la première variable de la nouvelle grappe. La nouvelle grappe d'index g est alors constituée en prenant les variables de l'index j jusqu'à j + Tmax -1 ; ou moins si la variable à insérer n'est pas connectée aux variables suivantes (cas où cardinalites[k-1] == 0). Finalement, la dernière boucle retire les variables qui ne sont pas connectées à la nouvelle grappe (cas où PasConnectee(grappes[g], ordre[l]) retourne vrai)[12]. Enfin, les variables de la nouvelle grappe sont retirées à l'ensemble V.

Dans l'exemple de la figure 5.2, l'exécution de l'algorithme HVD avec Tmax = 5 retourne les deux grappes $\{8, 6, 4, 2, 1\}$ puis $\{10, 7, 5, 9, 3\}$. Si l'ordre MDO n'avait pas été appliqué, nous aurions obtenu trois grappes $\{8, 6, 4, 2\}$ puis $\{10, 7, 9, 5, 3\}$ et enfin $\{1\}$. Le fait que Tmax soit fixe rend l'algorithme HVD moins puissant qu'une K-partition équilibrée de graphe. En revanche la K-partition est plus délicate à configurer (paramètres du recuit simulé). De plus, le coût des contraintes n'intervient pas dans les heuristiques de HVD. Une façon de le prendre en compte est de rajouter un trie supplémentaire entre MDO et MCO, comme par exemple, l'heuristique *min criticality (Ω)* présentée dans la section 2.3.1.3.

[11] Une cardinalité égale à 1 signifie qu'il n'y a plus qu'une contrainte, appelée *pont*, qui relie les variables avant j de celles après j.

[12] Cette opération ne suffit pas à assurer que la nouvelle grappe est connexe. Elle ne retire de la grappe que les variables isolées

5.2.2.3 Expérimentations sur des problèmes SPOT5 et sur l'instance 6 du CELAR

Expérimentations sur des problèmes SPOT5

Nous avons expérimenté l'algorithme glouton de décomposition sur les problèmes SPOT5. Du fait que l'ordre initial est très important pour obtenir une résolution rapide avec l'algorithme *Russian Doll Search*, nous n'avons pas appliqué l'heuristique de cardinalité maximum ni celle de degré maximum. Ainsi ordre conserve l'ordre initial et cardinalites contient les cardinalités des variables suivant l'ordre initial. Le tableau suivant montre les minorants, les qualités (lb/opt) et les temps de résolution sur une station *SUN Sparc4* pour résoudre 10 problèmes SPOT5 avec une taille maximale des grappes égale soit à 50, soit à 70, soit à 100, soit au nombre total de variables du problème (sans décomposition) :

	Tmax = 50			Tmax = 70			Tmax = 100			-		
Nom	**lb**	**q**	**t**	**lb**	**q**	**t**	**lb**	**q**	**t**	**opt**	**q**	**t**
28	247105	91.5	25	250105	92.6	27	249105	92.2	191	270105	100	50000
42	147050	94.8	20	151050	97.4	20	151050	97.4	21	155050	100	34
505	20238	95.2	27	20247	95.3	139	21251	100	136	21253	100	98
412	30372	93.8	16	31368	96.9	119	31380	96.9	122	32381	100	194
5	253	96.9	61	256	98.1	94	254	97.3	502	261	100	970
408	6215	99.8	19	6215	99.8	104	6220	99.9	104	6228	100	92
507	26383	96.3	32	26378	96.3	60	27382	100	56	27390	100	206
503	11108	100	18	11108	100	16	11108	100	16	11113	100	14
11	35485	92.2	38	36477	94.8	96	37473	97.4	88	38478	100	320
509	33437	91.7	37	33437	91.7	127	34444	94.5	155	36446	100	370
TOTAL	557646	93.1	294	566641	94.6	803	569667	95.1	1391	598705	100	52298

Nous pouvons noter que la résolution est environ 10 fois plus rapide sur les problèmes durs avec une décomposition limitant la taille des grappes à 50 variables (Tmax = 50). De même, la qualité des minorants est bonne, supérieure à 90%. Sur les problèmes plus faciles, le temps pris par l'algorithme de décomposition et par la résolution de plusieurs sous-problèmes au lieu d'un seul dépasse le temps de résolution pris par la méthode complète appliquée directement sur les problèmes sans simplification.

Expérimentations sur l'instance 6 du CELAR

L'algorithme glouton avec toutes ses heuristiques (y compris *min criticality*) produit des grappes similaires à l'algorithme du recuit simulé effectuant une K-partition. En effet, lorsque Tmax = 14, l'algorithme HVD trouve une première grappe identique à *CELAR6-SUB1*, à l'exception d'une variable manquante, 273, remplacée par 583. De même, lorsque Tmax = 22, l'algorithme HVD trouve une première grappe identique à *CELAR6-SUB4*, à l'exception de deux variables manquantes, 787 et 789, remplacées par 589 et 629.

5.3 Réduction des domaines

La réduction de domaine consiste à détecter des valeurs qui peuvent être indifféremment employées dans les instanciations complètes, ne modifiant pas leurs valuations. Ce comportement équivalent de deux valeurs n'est que rarement observé en pratique. En revanche, lorsque les domaines sont composés de valeurs numériques proches, celles-ci ont souvent un comportement presque équivalent. C'est l'objectif de la simplification de rendre ces valeurs complètement équivalentes au sens des valuations des instanciations complètes pour ensuite pouvoir n'en considérer qu'une seule durant la résolution. Les deux valeurs proches sont dites *valeurs similaires*.

5.3.1 Valeurs similaires et propriété de substituabilité valuée

Ces travaux s'inspirent de la notion d'interchangeabilité étudiée dans le cadre des CSP [Fre91], ainsi que d'une notion plus faible, la substituabilité [Fre91, K$\ddot{9}$4, BCH$^+$96]. Une méthode complète pour résoudre un CSP en utilisant la réduction de domaine par abstraction est présentée dans [FS95]. L'abstraction consiste à regrouper des valeurs arbitrairement (ou en fonction de connaissance sur le domaine d'application), pour former des valeurs abstraites ; cela implique une simplification des contraintes par ajout de n-uplets. Ensuite , une détection de la substituabilité est appliquée sur le problème abstrait permettant de regrouper les valeurs abstraites interchangeables. Ce problème réduit est résolu ; chaque solution satisfiable est évaluée dans le problème initial (phase de reformulation), jusqu'à ce qu'une solution satisfiant toutes les contraintes soit trouvée, ou bien que le CSP soit prouvé insatisfiable. Dans le cas où le problème réduit est insatisfiable, nous pouvons en déduire que le problème initial est aussi insatisfiable. [SM96] étudie le degré d'abstraction maximum pour pouvoir prouver l'insatisfiabilité d'instances de CSP en résolvant uniquement le problème abstrait. Ils montrent que l'abstraction subit un phénomène de seuil au delà duquel le problème abstrait devient cohérent, alors que le problème cible était incohérent.

5.3.1.1 Extension de l'interchangeabilité et de la substituabilité au cadre des VTCSP

Nous proposons d'étendre ces travaux au cas de l'optimisation sous contraintes. Pour faciliter cette extension, nous considérons un cadre légèrement différent du cadre des VCSP présenté en section 2.1. Premièrement, nous limitons notre étude aux VCSP binaires additifs sans contrainte dure. Deuxièmement, la fonction de valuation φ ne s'applique plus sur des contraintes, mais sur des couples de valeurs. Cela permet de faire varier la valuation d'une contrainte suivant l'affectation des variables. Ainsi, pour deux variables données, il n'y a entre elles au plus qu'une seule contrainte. C'est le cadre des *Valued Tuples Constraint Satisfaction Problems* (VTCSP) décrit dans la section 2.1.3.2, réduit aux contraintes binaires. Finalement, φ peut être considérée comme un tableau à 4 dimensions : soit a, b, deux valeurs affectées respectivement à deux variables, x_1, x_2 ; alors $\varphi[x_1][x_2][a][b] \in E$ indique la valuation attribuée à cette affectation. Ainsi, la valuation d'une instanciation complète A s'écrit :

$$\varphi(A,P) = \sum_{x_1 \in V} \sum_{x_2 \in V, x_2 > x_1} \varphi[x_1][x_2][A(x_1)][A(x_2)] \qquad (5.6)$$

Les propriétés autour de l'interchangeabilité s'étendent alors au cas de l'optimisation en remplaçant la notion de cohérence par la notion de valuation identique :

Définition 29 ———

Interchangeabilité valuée :
deux valeurs a, b sont interchangeables valuées ssi :

- *pour toute instanciation complète optimale A du* VTCSP *qui contient a , alors l'instanciation complète A où a est remplacée par b est aussi optimale ;*
- *pour toute instanciation complète optimale A du* VTCSP *qui contient b , alors l'instanciation complète A où b est remplacée par a est aussi optimale.*

Définition 30 ---
Interchangeabilité valuée totale :
deux valeurs a, b sont totalement interchangeable valuée ssi toute instanciation complète A du VTCSP *qui contient a possède la même valuation que l'instanciation complète A où a est remplacée par b.*

Dans les deux cas, le regroupement des deux valeurs en une seule ne modifie pas l'ensemble des solutions optimales ; il permet de le représenter de façon compacte par un produit cartésien [Les94, Les95] et accélère la résolution. L'interchangeabilité valuée totale implique l'interchangeabilité valuée, mais pas l'inverse.

Une façon d'établir l'interchangeabilité valuée totale, sans parcourir toutes les solutions, consiste à examiner localement les coûts des affectations de deux variables, exprimés dans le tableau φ :

Définition 31 ---
Voisinage-interchangeabilité valuée :
deux valeurs a, b d'une variable x_1 sont voisinages-interchangeables valuées ssi :

$$\forall x_2 \in V, x_2 \neq x_1, \forall v \in D(x_2),\ \varphi[x_1][x_2][a][v] = \varphi[x_1][x_2][b][v] \tag{5.7}$$

Nous redéfinissons une notion plus faible que l'interchangeabilité, la substituabilité :

Définition 32 ---
Substituabilité valuée :
une valeur a est substituable valuée à une autre valeur b ssi toute instanciation complète optimale contenant b est aussi optimale en substituant b par a.

Ainsi, nous pouvons supprimer la valeur b en garantissant que l'ensemble des solutions optimales n'est pas vide (dans le cas où les contraintes dures sont satisfiables) et est inclus dans l'ensemble des solutions optimales du problème initial. En effet, les solutions optimales contenant a ne sont pas forcément des solutions optimales en remplaçant a par b.

De même, la substituabilité valuée peut être obtenue par une propriété locale :

Définition 33 ---
Voisinage-substituabilité valuée :
une valeur a d'une variable x_1 est voisinage-substituable valuée à une autre valeur b ssi :

$$\forall x_2 \in V, x_2 \neq x_1, \forall v \in D(x_2),\ \varphi[x_1][x_2][a][v] \leq \varphi[x_1][x_2][b][v] \tag{5.8}$$

La voisinage-substituabilité valuée signifie qu'utiliser la valeur a conduit toujours à des instanciations complètes de plus faible valuation que les mêmes instanciations complètes avec b à la place de a. Naïvement, la vérification de la propriété 33 implique l'examen d'au plus $2 \cdot (n-1) \cdot d$ valuations, avec n, le nombre de variables et d, le nombre maximum de valeurs par domaine.

5.3.1.2 Réduction de domaine par détection de valeurs similaires

Nous définissons la notion de valeurs similaires à partir de la notion de voisinage-interchangeabilité valuée :

Définition 34

Valeurs similaires à ϵ près par voisinage-interchangeabilité valuée :
deux valeurs a, b d'une variable x_1 sont dites similaires à un degré de similitude près, noté ϵ, par voisinage-interchangeabilité valuée ssi $\bar{\delta}(x_1, a, b)$, la somme des valeurs absolues des différences de valuation de tous les couples interdits impliquant a ou b est inférieure au degré de similitude ϵ :

$$\bar{\delta}(x_1, a, b) = \sum_{x_2 \in V / x_2 \neq x_1} \sum_{v \in D(x_2)} |\varphi[x_1][x_2][a][v] - \varphi[x_1][x_2][b][v]| \leq \epsilon \tag{5.9}$$

Ainsi, $\bar{\delta}(x_1, a, b)$ représente le coût de la relaxation des contraintes pour que les valeurs a et b soient totalement interchangeables valuées ($\bar{\delta}(x_1, a, b) = \bar{\delta}(x_1, b, a)$). Néanmoins, il est possible de relaxer moins les contraintes tout en ayant la possibilité de regrouper des valeurs grâce à la propriété plus faible de voisinage-substituabilité valuée. Nous définissons qu'une valeur est quasi-substituable à une autre valeur de la manière suivante :

Définition 35

Valeurs quasi-substituables à ϵ près
Soit $S[x_1][x_2][a][b][v]$, une matrice booléenne telle que :

$$\forall x_1, x_2 \in V, a, b \in D(x_1), v \in D(x_2),\ S[x_1][x_2][a][b][v] = \begin{cases} 1 & si\ \varphi[x_1][x_2][a][v] > \varphi[x_1][x_2][b][v] \\ 0 & sinon \end{cases}$$

Une valeur a d'une variable x_1 est quasi-substituable à une autre valeur b à un degré ϵ près ssi $\delta(x_1, a, b)$, la somme des différences de valuation des couples interdits tels que la valuation du couple passant par a est supérieure à la valuation du couple passant par b, est inférieure au degré ϵ :

$$\delta(x_1, a, b) = \sum_{x_2 \in V / x_2 \neq x_1} \sum_{v \in D(x_2)} S[x_1][x_2][a][b][v] \cdot (\varphi[x_1][x_2][a][v] - \varphi[x_1][x_2][b][v]) \leq \epsilon \tag{5.10}$$

La valuation $\delta(x_1, a, b)$ représente le coût de la relaxation pour pouvoir supprimer b du domaine de x_1 sans modifier l'optimum du problème relaxé. Lors de la recherche de valeurs quasi-substituables, il est important de tester les substituabilités dans les deux sens : en général, $\delta(x_1, a, b) \neq \delta(x_1, b, a)$. Par ailleurs, nous avons toujours $\delta(x_1, a, b) \leq \bar{\delta}(x_1, a, b)$ et $\delta(x_1, b, a) \leq \bar{\delta}(x_1, a, b)$. En fait, le calcul de ces trois termes, $\bar{\delta}(x_1, a, b)$, $\delta(x_1, a, b)$ et $\delta(x_1, b, a)$, peut se faire dans une seule boucle. Il faudra évaluer au plus $2 \cdot (n-1) \cdot d$ valuations de couples différents pour tester si deux valeurs sont similaires à ϵ près et si oui, laquelle il faut supprimer.

La figure 5.3 montre un exemple de relaxation des contraintes pour transformer une valeur quasi-substituable (a) en une valeur substituable. Le problème initial contient deux variables (x_1 et x_2) et une seule contrainte (c_1). Le but étant de réduire le domaine de x_1 à une seule valeur, c'est la valeur a qui est choisie pour être substituée à b (b étant supprimée dans le problème simplifié) car $\delta(x_1, a, b) < \delta(x_1, b, a)$. L'opération consiste à modifier les valuations des couples en choisissant toujours le minimum entre $\varphi[x_1][x_2][a][i]$ et $\varphi[x_1][x_2][b][i]$. La figure 5.5 montre l'effet de la réduction des domaines sur une contrainte d'un problème plus complexe, le sous-problème *CELAR6-SUB1* (cf. section 5.3.3).

Réduction du domaine de x_1 et simplification de la contrainte c_1

	$\varphi(c_1)$	u (x_2)	v (x_2)	w (x_2)
x_1	a	100	0	10
x_1	b	10	10	110

$\bar{\delta}(x_1, a, b) = 90 + 10 + 100 = 200$

$\boxed{\delta(x_1, a, b) = 90}$

$\delta(x_1, b, a) = 10 + 100 = 110$

suppression de la valeur b
(a quasi-substituable à b)

et simplification de la contrainte c_1

	$\Phi \circ \varphi(c_1)$	u (x_2)	v (x_2)	w (x_2)
x_1	a	10	0	10
x_1	~~b~~	~~10~~	~~0~~	~~10~~

Fig. 5.3 - Exemple de relaxation d'une contrainte binaire lors d'une réduction de domaine

Remarque 34 [Fre91] a aussi défini une propriété intermédiaire, appelée k-interchangeabilité, moins exigeante que la voisinage-interchangeabilité, dans le cadre des CSP. Dans le cas valué, il s'agirait de tester localement la propriété de k-interchangeabilité valuée, non pas vis à vis d'une seule autre variable (x_2, pour la voisinage-interchangeabilité), mais étendue à $k-1$ autres variables. Cela permettrait de diminuer le coût des relaxations par rapport à la voisinage-interchangeabilité, moyennant une transformation des contraintes binaires en contraintes k-aires pour ensuite pouvoir les simplifier. Le temps de calcul pour vérifier ces propriétés augmente de manière exponentielle en fonction de k.

$\diamond$

5.3.2 Réduction cyclique gloutonne

La simplification par relaxation à l'intérieur des contraintes pour rendre substituables des valeurs quasi-substituables est un problème d'optimisation combinatoire. Il s'agit de maximiser le nombre de regroupement de valeurs, pour accélérer la résolution, tout en minimisant les coûts de relaxation, dans l'espoir que le minorant déduit de l'optimum du problème simplifié reste proche de l'optimum du problème cible.

Une difficulté importante pour ce problème est que l'ordre des regroupements influe sur le coût des relaxations. En effet, il peut être préférable de regrouper d'abord a, b de x_1 puis u, v de x_2, ou l'inverse.

Une approche sous-optimale pour résoudre le problème de la simplification consiste à itérer une sélection de deux valeurs dont l'une est quasi-substituable à l'autre, puis une relaxation des différences sur les valuations des couples contenant ces deux valeurs, suivant la matrice S et enfin, une suppression de l'une des deux valeurs de son domaine. Or, le choix d'un regroupement de deux valeurs influe sur le coût des regroupements suivants. Nous avons choisi une approche cyclique, considérant variable après variable, la meilleure sélection au sein de la variable courante. Soit $V(j)$, la *jème* variable dans la liste V ; soit $D(x,k)$, la *kème* valeur dans le domaine de la variable x ; alors l'algorithme glouton de regroupement s'écrit :

```
REDUCE( V, D, φ, R )
  pour i = 1 a R faire
      pour j = 1 a |V| faire
          x1 := V(j) ; v1 := 0 ; v2 := 0 ; δ := +∞
          pour k = 1 a |D(x1)| faire
              a := D(x1, k)
              pour l = k + 1 a |D(x1)| faire
                  b := D(x1, l)
                  calculer δ(x1, a, b) et δ(x1, b, a)
                  si δ(x1, a, b) < δ ou δ(x1, b, a) < δ alors
                    si δ(x1, a, b) < δ(x1, b, a) alors
                      v1 := k ; v2 := l ; δ := δ(x1, a, b)
                    sinon
                      v1 := l ; v2 := k ; δ := δ(x1, b, a)
                    fin si
                  fin si
              fin pour
          fin pour
          si δ < +∞ alors
            effectuer la relaxation de cout δ en modifiant φ
            D(x1) := D(x1) − D(x1, v2)
          fin si
      fin pour
  fin pour
```

Cet algorithme supprime au maximum R valeurs par domaine. Une variable ayant un domaine réduit à une seule valeur n'est plus considérée par les réductions. En cas d'égalité $\delta(x_1,a,b) = \delta(x_1,b,a)$, il est préférable de choisir la relaxation qui minimise le nombre de valuations de couples à modifier. Par ailleurs, l'ordre d'examen des variables est important ; il est préférable de commencer par les variables les moins contraintes en premier. Une amélioration de l'algorithme REDUCE consisterait à détecter les couples de valeurs devenues substituables valuées ($\bar{\delta} = 0$), après une étape de la relaxation.

La complexité de cet algorithme en nombre de valuations de couples testées est en $O(R \cdot n^2 \cdot d^3)$. Une autre approche aurait consisté à rechercher le meilleur regroupement sur toute les variables, pour chaque réduction. La complexité de cet algorithme augmente, elle est en $O(R \cdot n^3 \cdot d^3)$. Il s'avère que dans ce cas les réductions ont tendance à s'effectuer principalement sur les variables les moins contraintes, jusqu'à ce que leurs domaines soient réduits à une seule valeur. Les minorants obtenus par ce type de réduction sont généralement nettement moins bons que ceux obtenus par l'approche cyclique.

5.3.3 Expérimentations sur l'instance 6 du CELAR

Nous avons appliqué l'algorithme REDUCE sur deux sous-problèmes de l'instance 6 du CELAR. Le tableau suivant résume les minorants (lb) déduits directement des optima des problèmes simplifiés, leurs qualités par rapport au sous-problème ($q = lb/opt$) et par rapport à l'instance 6 complète ($q' = lb/3389$), ainsi que le temps mis pour les résoudre. Les temps indiqués ne comportent que les temps pour résoudre les problèmes simplifiés par un algorithme *Russian Doll Search* exécuté sur une station *SUN Sparc4*, sans tenir compte des temps mis pour simplifier les problèmes. L'algorithme de réduction glouton REDUCE prend 8 minutes pour R = 20 sur le problème *CELAR6-SUB1*. Le coût total des relaxations pour R = 20 est de 1362339, avec δ variant entre 1676 et 11281.

CELAR6-SUB*i*	**nbvar**	**R**	*lb*	*q*	*q'*	**temps (sec)**
CELAR6-SUB1	14	34	985	36.9%	29.1%	9[13]
		32	1739	65.2%	51.3%	55[13]
		30	1852	69.4%	54.6%	130
		28	2055	77%	60.6%	325
		26	2288	85.7%	67.5%	1250
		24	2338	87.6%	69%	476
		22	2556	95.8%	75.4%	1275
		20	2656	99.5%	78.4%	2952
		0	2669	100%	78.7%	900 min
CELAR6-SUB4	22	26	2603	80.6%	76.8%	5404
		24	2885	89.32%	85.1%	15992
		0	3230	100%	95.3%	$\simeq$ 54 jours

Les domaines de ces sous-problèmes contiennent au plus 44 valeurs. La réduction de moitié des domaines (R = 22) permet un gain de temps supérieur à 40 pour le problème *CELAR6-SUB1*, tout en obtenant un minorant à seulement 4.2% de l'optimum. La figure 5.4 de gauche montre l'évolution des minorants déduits des simplifications de *CELAR6-SUB1*. Nous observons qu'une trop forte réduction des domaines fait juter brutalement le minorant (R = 34). A l'inverse, sur ce problème, le minorant a presque atteint l'optimum à partir de la simplification R = 20. Les simplifications pour R inférieur à 20 deviennent inutiles. La figure 5.4 de droite montre l'évolution du temps de résolution des simplifications de *CELAR6-SUB1*. Globalement, ce temps croît de manière exponentielle lorsque R diminue. Notons la discontinuité observée à l'étape R = 26. Le temps de résolution est plus grand qu'aux étapes précédente et suivante, moyennant un minorant en nette progression.
La figure 5.5 permet de visualiser les relaxations effectuées sur une contrainte binaire de *CELAR6-SUB1*, une fois la simplification par réduction des domaines à 22 valeurs (R = 22) effectuée. Chaque point représente un couple de fréquences autorisées par la contrainte. Le fait qu'il y ait sur la figure moins de points pour la contrainte relaxée par rapport à la contrainte initiale vient de ce que les domaines des variables ont été réduits de moitié par la simplification. En revanche, nous pouvons noter qu'il y a très peu de points où la contrainte simplifiée est satisfaite et la contrainte initiale insatisfaite (uniquement 6 sur cet exemple). Pour cette contrainte, le problème relaxé semble très proche du problème initial.

5.4 Enchaînement des relaxations

Stratégie de résolution

Dans le cadre d'une approche interruptible, nous cherchons à encadrer progressivement l'optimum des problèmes à résoudre (cf. chapitre 3). La stratégie de résolution que nous proposons ici ne

[13] A la place de l'algorithme *Russian Doll Search*, un simple *depth-first branch and bound* avec *forward checking valué* et heuristiques dynamiques a été utilisé pour résoudre plus rapidement ces deux problèmes simplifiés.

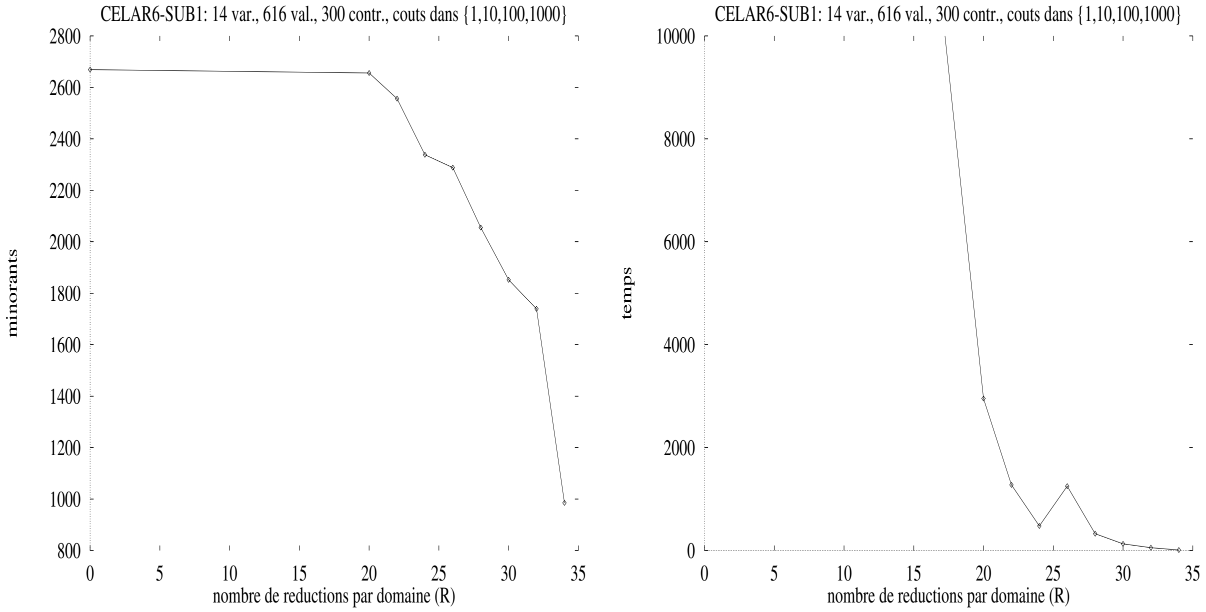


Fig. 5.4 - Evolution des minorants et des temps de résolution des simplifications par réduction des domaines du sous-problème *CELAR6-SUB1*

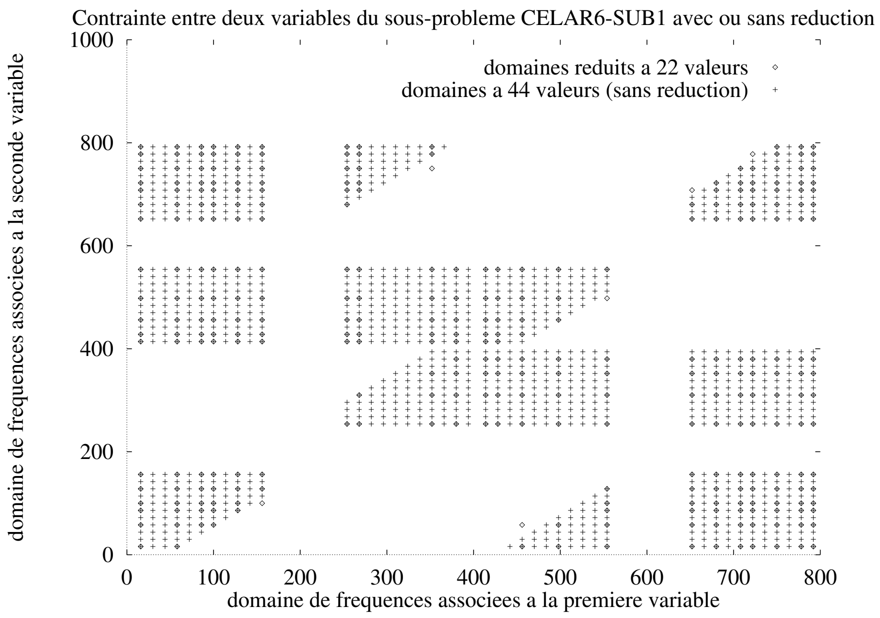


Fig. 5.5 - Effet d'une relaxation interne dans une contrainte binaire du sous-problème *CELAR6-SUB1*, lors d'une réduction des domaines (les points représentent les couples de fréquences autorisés)

concerne que la production de minorants. Les majorants sont produits par une succession de méthodes incomplètes sur les problèmes cibles. Pour un problème cible donné, la stratégie de résolution consiste à itérer le schéma suivant jusqu'à ce que l'optimum du problème cible soit atteint et prouvé :

- décomposition des variables du problème en grappes, en augmentant progressivement la taille maximale des grappes à chaque itération du schéma de résolution ; lors des premières décompositions, nous pouvons utiliser l'algorithme glouton HVD avec Tmax petit ; plus tard, nous pouvons appliquer un algorithme de recuit simulé effectuant une K-partition avec K qui augmente et le facteur de pondération multi-critère h qui diminue ;

- analyse de la difficulté des grappes générées ; un premier critère est la taille de la grappe ; un second critère est la connaissance d'un majorant de la grappe ; pour cela, nous appliquons une méthode incomplète sur les différentes grappes, avec un temps d'exécution fixé assez court ; certaines grappes sont alors éliminées si la méthode incomplète produit une solution de coût nul ; les grappes restantes sont traitées par ordre de majorant décroissant ; la stratégie diffère suivant la difficulté de la grappe :

  - résolution des grappes jugées faciles (soit parce qu'elles sont de taille petite, soit parce qu'elles ont un majorant proche de zéro) par une méthode complète de type *Depth First Branch and Bound* avec une heuristique dynamique sur le choix des variables et une borne initiale fixée à la valeur du majorant ;
  - simplification des grappes jugées difficiles par réduction de domaines :
    * application de l'algorithme REDUCE avec un nombre d'itération R fixé environ aux trois quarts de la taille moyenne des domaines de la grappe ciblée ; à chaque itération, nous mémorisons la grappe simplifiée ;
    * résolutions successives par une méthode complète des simplifications de la grappe, avec R qui décroît progressivement ; une méthode incomplète est préalablement exécutée pour déterminer un majorant des grappes simplifiées ; ainsi seules les simplifications qui peuvent apporter une amélioration suffisante du minorant sont résolues ; l'enchaînement des résolutions se termine lorsque $\mathsf{R} = 0$; les dernières résolutions peuvent ne pas avoir été faites si la méthode incomplète a détecté une absence de progression du minorant.

L'ajustement fin des paramètres de ce schéma de résolution pour un certain domaine d'application nécessite plusieurs expérimentations pour obtenir une stratégie efficace. Nous présentons dans le paragraphe suivant la stratégie de résolution, obtenue après expérimentations, de l'instance 6 du CELAR.

Expérimentation sur l'instance 6 du CELAR

La figure 5.6 montre l'application de la stratégie de résolution précédente sur l'instance 6 du CELAR. Nous voyons l'évolution de l'encadrement de l'optimum (égal à 3389) en terme de valuations des minorants et des majorants, en fonction du temps de résolution (échelle logarithmique). La production des majorants et des minorants est faite en parallèle (ressources matérielles doublées). Les temps affichés ne prennent pas en compte les durées des algorithmes incomplets qui effectuent ou analysent les simplifications. Les majorants sont produits par trois algorithmes de recuit simulé successifs, de plus en plus lents. Pour les minorants, nous avons repris les résultats indiqués dans le tableau de la section 5.3.3. En premier, la grappe *CELAR6-SUB1* à

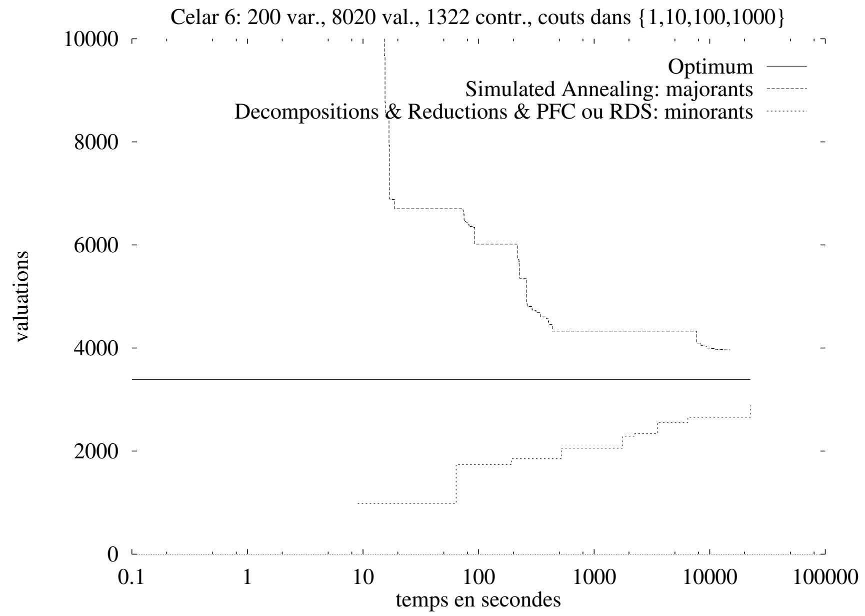


Fig. 5.6 - Encadrement progressif de l'optimum pour l'instance 6 du CELAR

14 variables a été résolue avec au préalable une réduction des domaines pour R allant de 32 à 22. Une seconde grappe, légèrement différente de *CELAR6-SUB0*, a également été résolue sans réduction des domaines. Puis, au cycle suivant, c'est la grappe *CELAR6-SUB4* à 22 variables qui a été résolue, avec une réduction des domaines $\mathsf{R} = 24$.

Le tableau suivant résume l'évolution de l'encadrement de l'optimum, ainsi que l'évolution de la qualité normée approchée (cf. équation 3.3 de la section 3.1.2). Il a suffi de dix minutes pour obtenir une solution de qualité normée supérieure à 99%. En six heures, l'encadrement de l'optimum est inférieur à 1000. Rappelons qu'il nous a fallu un week-end sur un réseau de quarante *SUNs* pour prouver l'optimum.

temps	lb	ub	$ub - lb$	$(\omega - ub)/(\omega - lb)$
~ 10 sec	1000	10000	9000	96.5%
~ 1 min	1735	6700	4965	98%
~ 10 min	2155	4328	2173	99.1%
~ 1 h	2656	4328	1672	99.3%
~ 6 h	3044	3963	919	99.6%
~ 54 j	3389	3389	0	100%

Chapitre 6

Simplifications de l'objectif

Dans le cas des simplifications structurelles, nous utilisions une méthode complète qui résolvait entièrement un problème simplifié. La résolution impliquait de trouver l'optimum et surtout de prouver son optimalité. Les simplifications de l'objectif consistent à changer l'objectif de la résolution sans modifier la structure du problème. L'objectif n'est plus uniquement de prouver l'optimum d'un problème, mais d'abord de produire un ou plusieurs minorants. Nous abordons deux façons d'obtenir un ou plusieurs minorants en simplifiant l'objectif d'une méthode complète.

La première solution, traitée dans la section 6.1, consiste à établir un minorant par renforcement de cohérence locale. Dans ce cas, le nouvel objectif consiste à atteindre un certain niveau de cohérence locale.

La deuxième solution, traitée dans la section 6.2, consiste à limiter l'arbre de recherche d'une exploration complète et exploiter les minorants des instanciations partielles pour en déduire un minorant du problème cible durant la recherche. Dans ce cas, le nouvel objectif est de produire plusieurs minorants, suivant le temps disponible, en utilisant une méthode complète spécifique.

6.1 Relaxations localement cohérentes et relaxation linéaire

Dans le cadre CSP, un certain niveau de cohérence locale peut être établi par un algorithme de renforcement de cohérence. Par exemple, un problème CSP binaire est rendu arc-cohérent en éliminant les valeurs de ses variables qui n'ont pas de support dans le domaine des autres variables reliées par une contrainte (cf. section 2.2.1.2). L'algorithme de renforcement de cohérence simplifie le problème sans perdre aucune instanciation complète cohérente si celui-ci en possède.

Dans le cadre VCSP, la cohérence locale prend une nouvelle signification. En effet, il n'est plus question d'instanciations complètes cohérentes ou incohérentes, mais d'instanciations complètes optimales ou sous-optimales. Le renforcement de cohérence étendu à l'optimisation a pour objectif de calculer la valuation minimum entraînée par des instanciations partielles, c-à-d un minorant de la valuation globale de ces instanciations. Ensuite, il faut combiner ces minorants pour en déduire un minorant du problème cible. Nous présentons quelques solutions obtenues par des algorithmes polynomiaux dans la section 6.1.1.

Il existe une autre façon d'aborder la cohérence locale dans les VCSP. Cela consiste à déterminer la relaxation minimale des contraintes pour obtenir un problème relaxé qui soit localement cohérent. Notons que l'optimum d'un VCSP est égal à la valuation de la relaxation minimale des contraintes pour obtenir un problème relaxé qui soit cohérent. Puisque tout problème relaxé cohérent est aussi localement cohérent, alors la relaxation optimale des contraintes pour obtenir un problème relaxé localement cohérent fournit bien un minorant du problème cible. Hélas, cette modification de l'objectif ne s'exprime pas aisément dans le cadre des VCSP. En revanche, nous montrons, dans la section 6.1.2, comment cela peut s'exprimer facilement dans un autre cadre, celui de la programmation linéaire en nombre 0/1.

6.1.1 Méthodes de renforcement de cohérence étendues à l'optimisation

Nous avons vu dans la section 2.2.2 que le renforcement par arc-cohérence s'étend bien aux VCSP avec un opérateur d'agrégation idempotent. Il permet de construire des contraintes valuées induites unaires, notées $C_{unary}(x, v)$, pour chaque valeur v de chaque variable x. La valuation maximale des minimum des valuations de ces contraintes par domaine est un minorant du problème cible :

$$P \in max\text{-VCSP}, \quad lb(P) = \max_{\forall x \in V}(\min_{\forall v \in D(x)} \varphi(C_{unary}(x, v)))$$

En revanche, lorsque l'opérateur d'agrégation n'est pas idempotent, il n'est pas possible d'ajouter directement des contraintes valuées induites sans modifier le problème. Néanmoins, il existe des formes affaiblies de renforcement d'arc-cohérence, construisant des valuations par valeur, sans faire de propagation. C'est le cas des *Arc Consistency Counts*, notés *ACC(x,v)*, et des *Directed Arc Consistency Counts*, notés *DACC(x,v)* (cf. exemple de la figure 2.4). Nous pouvons déduire les minorants suivants :

$$lb_{AC}(P) = \max_{\forall x \in V}(\min_{\forall v \in D(x)} ACC(x, v)) \tag{6.1}$$

$$lb_{DAC}(P) = \sum_{\forall x \in V}(\min_{\forall v \in D(x)} DACC(x, v)) \tag{6.2}$$

Notons que lb_{DAC} utilise une somme de valuations et pas lb_{AC}. En effet, la construction des *DACCs* considère un sous-ensemble de contraintes différent pour chaque variable. Or, nous avons vu dans le chapitre 5, que la somme des optima, a fortiori des minorants, de plusieurs sous-problèmes qui ne partagent aucune contrainte est un minorant du problème initial. Ce n'est pas en général le cas pour la construction des valuations *ACCs*. Sur un problème donné, il serait intéressant d'examiner une décomposition des contraintes telle que la somme des ACCs indépendants déduits de cette décomposition soit maximale. Une approche similaire pour améliorer $lb_{dac}(A, P)$ au sein d'une méthode complète de résolution est proposée dans [LMSV98].

Enfin, [Dag97] propose des algorithmes capables de générer au cours d'une recherche complète des *Nogoods valués* qui sont des minorants de la valuation globale d'instanciations partielles. Ces informations peuvent éventuellement être combinées pour produire des minorants du problème cible durant la recherche.

Expérimentations sur des problèmes aléatoires de lb_{DAC}

La qualité des minorants obtenus à l'aide des calculs de *DACCs* est très médiocre. La figure 6.1 montre l'évolution de la qualité moyenne de ces minorants sur des problèmes Σ-VCSP binaires aléatoires, notés *Pa* (cf. section 4.2.2), en fonction de la connectivité et de la dureté. La qualité moyenne est égale à la moyenne sur 100 problèmes du rapport entre le minorant et l'optimum,

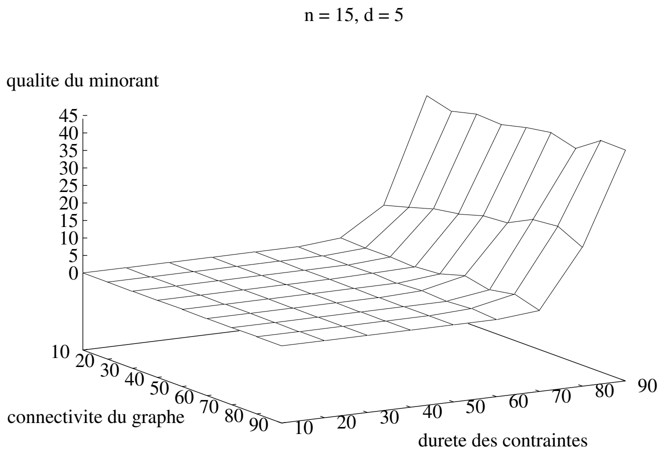


Fig. 6.1 - Qualité des minorants de Pa obtenus par $lb_{DAC}(P)$

soit $lb_{DAC}(P)/\varphi(P)$[1]. C'est seulement pour des duretés supérieures à 70% que les minorants $lb_{DAC}(P)$ sont non nuls. La qualité atteint environ 40% à la dureté égale à 90%. La connectivité ne semble pas influencer la qualité des minorants.

De plus, les valuations *DACCs* varient suivant l'ordre statique choisi pour les variables (ordre *max-degree* pour la figure 6.1). De même, la qualité a tendance à diminuer lorsque la taille des domaines augmente (les minorants sont presque tous nuls sur les problèmes Pm avec des domaines de 10 valeurs). En revanche, le calcul des *DACCs* est relativement peu coûteux. Il se fait en temps polynomial, en $O(c \times d^2 + n \times d)$, avec c, le nombre de contraintes binaires, d, le nombre maximum de valeurs par domaine et n, le nombre de variables.

6.1.2 Programmation linéaire en nombre 0/1

Initialement, la **Programmation linéaire** (*PL*) [NW88, Van95] est un ensemble de techniques pour résoudre des problèmes d'optimisation en variables positives réelles, décrits par une fonction à minimiser, l'objectif, et un système d'équations linéaires représentant des contraintes dures qui limitent le domaine de validité des variables. Un problème se note classiquement de la façon suivante, avec n, le nombre de variables et m, le nombre de contraintes :

$$\left\{\begin{array}{ll} \text{minimiser} & \sum_{j=1}^{n} c_j x_j \\ \text{soumis à} & \\ & \sum_{j=1}^{n} a_{ij} x_j = b_i \quad \forall\, 1 \leq i \leq m \\ & x_j \geq 0 \qquad\qquad\;\; \forall\, 1 \leq j \leq n \end{array}\right. \tag{6.3}$$

Cette représentation peut être facilement étendue pour accepter :

- des variables quelconques, positives ou négatives, exprimées par la différence de deux variables positives ;
- une fonction objectif à maximiser, par négation de la fonction ;

[1] Lorsque les problèmes sont tous satisfiables, la qualité est fixée à zéro.

- des contraintes d'inéquation : $t \geq 0 \Leftrightarrow t - s = 0$, avec s, une nouvelle variable ;
- des contraintes exprimant la disjonction : $x_1 \geq x_2 + d_2 \vee x_2 \geq x_1 + d_1 \Leftrightarrow x_1 \geq x_2 + d_2 - bM \wedge x_2 \geq x_1 + d_1 - (1-b)M$, avec $0 <= b <= 1$, une variable booléenne (réellement prise en compte dans le modèle *PL01*) et M, une constante assez grande.

Il existe plusieurs modèles dérivés de ce formalisme initial. C'est le cas de la **Programmation linéaire en nombre entier** (*PLNE*)[2] qui stipule que toutes les variables prennent des valeurs entières. Une sous-classe de ce modèle est la **Programmation linéaire en nombre** 0/1 (*PL01*) qui restreint les variables aux valeurs 0 ou 1 (les variables sont dites variables booléennes). Enfin, un modèle plus générale, appelé *Mixed Integer Programming* (*MIP*) combine des variables entières avec des variables réelles.

La résolution du problème 6.3 en variables réelles est faite en temps polynomial, au pire en $O(n \times m^2)$. L'algorithme du *Simplex* [Dan74], bien que de complexité exponentielle dans le pire cas, prend un temps polynomial en moyenne pour résoudre ces problèmes. C'est l'algorithme le plus utilisé et le plus efficace lorsqu'il s'agit de résoudre la relaxation linéaire d'un problème *MIP*[3]. Il procède par une descente en gradient (suivi des côtés d'un polytope défini par les contraintes) qui mène à une solution optimale correspondant à un sommet du polytope. Le problème 6.3 peut également être traité en passant par son *dual* (inversion des variables et des contraintes, inversion de la matrice des coefficients a_{ij}, permutation des coefficients b_j et c_j). L'optimum du problème dual est identique au problème initial et il existe une transformation simple pour passer d'une solution du dual dans l'original. La complexité maximale du problème dual est en $O(m \times n^2)$. L'ajout de nouvelles contraintes est donc moins coûteux.

En revanche, l'ajout de variables discrètes rend le problème NP-complet. Dans le modèle *MIP*, seul l'ajout de variables réelles n'augmente pas la complexité maximale du problème. Les techniques de résolution de problèmes *MIP* sont de type *branch and bound*. Pour la méthode *branch and relax*, un minorant de la valuation globale d'une instanciation partielle est donnée par l'optimum du problème relaxé (les variables non instanciées pouvant prendre des valeurs réelles). En effet l'optimum de la relaxation linéaire est un minorant de l'optimum du problème à variables entières. Ce minorant permet de faire un retour arrière s'il est plus grand que le meilleur majorant trouvé. Sinon, deux nouveaux sous-problèmes sont examinés en séparant une variable x à valeur réelle par l'ajout respectivement de la contrainte $x \leq [f]$ et de la contrainte $x \geq [f]+1$ (f étant la valeur réelle de x et $[f]$, sa partie entière). La relaxation linéaire de ces nouveaux sous-problèmes peut se faire de manière incrémentale, par un algorithme *Simplex* appliqué au *dual* des sous-problèmes. D'autres méthodes de résolution, appelées *branch and cut*, ajoutent de nouvelles contraintes (*Cutting Planes*, *Polyhedra Cuts*) lorsque la solution n'est pas entière. L'efficacité de ces contraintes dépend beaucoup de la structure du problème (par exemple, la présence de cliques).

6.1.2.1 Transformation d'un problème Σ-VCSP en *PL01*

Nous présentons une transformation générique d'un problème Σ-VCSP dont les contraintes sont définies en extension dans le modèle *PL01*.

[2] "*Integer Linear Programming* (*ILP*)", en anglais.

[3] Il existe d'autres algorithmes pouvant être plus efficaces sur des problèmes très grands. C'est le cas des méthodes *Interior Point methods*, provenant de la *Programmation Non Linéaire*, qui réalisent un déplacement à l'intérieur du polytope.

Soit $P = (V, D, C, S, \varphi)$, un problème Σ-VCSP avec C, un ensemble de contraintes définies en extension. Soit n, le nombre de variables. Soit $V(j)$, la j ième variable de V. Soit d, le nombre maximum de valeur par domaine. Soit $D(V(j))$, le domaine de la variable $V(j)$, d_j, sa taille et $D(V(j), k)$, la k ième valeur dans le domaine de $V(j)$. Soit $C = (C(1), \ldots, C(m), C(m+1), \ldots, C(m+h))$, la liste des m contraintes molles de C, suivie des h contraintes dures de C. Soit $C(i)$, la ième contrainte de C, N_i, son arité et $\varphi(C(i))$, son coût ($i \leq m$). Enfin, soit $C(i) = (v_i, r_i)$, la description en extension de la ième contrainte de C, avec $v_i = (v_i^1, \ldots, v_i^{N_i})$, la liste des indices des variables reliées par la contrainte et $r_i(l) = (r_{il}^1, \ldots, r_{il}^{N_i})$, la liste des indices des valeurs du l ième n-uplet interdit par cette contrainte et L_i, le nombre de n-uplets interdits par $C(i)$.

Nous associons une variable booléenne x_{jk} dans le modèle *PL01* à chaque couple variable $V(j)$,valeur $D(V(j), k)$ de P : $x_{jk} = 1$ signifie que la valeur $D(V(j), k)$ est affectée à la variable $V(j)$. Il y a donc création au plus de $n \times d$ variables booléennes. Des contraintes impératives de domaine imposent qu'une et une seule valeur soit affectée à chaque variable : $\forall 1 \leq j \leq n,\ \sum_{k=1}^{d_j} x_{jk} = 1$.

Chaque contrainte dures de C est décomposée en autant d'inéquations linéaires qu'elle contient de n-uplets interdits. Ces inéquations s'écrivent $\forall m+1 \leq i \leq m+h, \forall 1 \leq l \leq L_i,\ \sum_{j=1}^{N_i} x_{v_i^j r_{il}^j} \leq N-1$.

Dans le cas des contraintes molles de C, nous effectuons la même décomposition en inéquations par n-uplets, sauf qu'une nouvelle variable booléenne c_i, associée à chaque contrainte molle, est ajoutée pour exprimer que la contrainte peut être violée : $c_i = 1$ si la contrainte $C(i)$ est violée et $c_i = 0$ si elle est satisfaite. L'inéquation devient alors : $\forall 1 \leq i \leq m, \forall 1 \leq l \leq L_i,\ \sum_{j=1}^{N_i} x_{v_i^j r_{il}^j} \leq N - 1 + c_i$, ou encore $\forall 1 \leq i \leq m, \forall 1 \leq l \leq L_i,\ \sum_{j=1}^{N_i} x_{v_i^j r_{il}^j} - c_i \leq N-1$.

Le critère à optimiser est de minimiser la violation des contraintes molles, donc la fonction objectif s'écrit $\sum_{i=1}^{m} \varphi(C(i)) \cdot c_i$.

En résumé, le problème $P = (V, D, C, S, \varphi)$ transformé dans le modèle *PL01* se note :

$$
\left\{
\begin{array}{llll}
\text{minimiser} & \displaystyle\sum_{i=1}^{m} \varphi(C(i)) \cdot c_i & & \\
\text{soumis à} & & & \\
 & \displaystyle\sum_{k=1}^{d_j} x_{jk} = 1 & \forall\, 1 \leq j \leq n & \\
 & \displaystyle\sum_{j=1}^{N_i} x_{v_i^j r_{il}^j} \leq N-1 & \forall\, m+1 \leq i \leq m+h & \forall\, 1 \leq l \leq L_i \\
 & \displaystyle\sum_{j=1}^{N_i} x_{v_i^j r_{il}^j} - c_i \leq N-1 & \forall\, 1 \leq i \leq m & \forall\, 1 \leq l \leq L_i \\
 & 0 \leq x_{jk} \leq 1 & \forall\, 1 \leq j \leq n & \forall\, 1 \leq k \leq d_j \\
 & x_{jk} \text{ entier} & \forall\, 1 \leq j \leq n & \forall\, 1 \leq k \leq d_j \\
 & 0 \leq c_i \leq 1 & \forall\, 1 \leq i \leq m & \\
 & c_i \text{ entier} & \forall\, 1 \leq i \leq m &
\end{array}
\right.
\tag{6.4}
$$

Cette transformation conduit à un programme linéaire en nombre 0/1 comportant au plus $n \times d + m$ variables booléennes. C'est donc beaucoup plus que les n variables discrètes du modèle VCSP.

Lorsque les contraintes sont binaires ($N_i = 2$), nous pouvons les représenter de façon plus compacte dans le modèle *PL01*. Pour cela, nous utilisons la notion de support : pour toute contrainte $C(i)$, toute valeur $D(V(v_i^1), k)$ de la variable v_i^1 doit être supportée par au moins une autre valeur dans le domaine de l'autre variable v_i^2 reliée par la contrainte ; et vice et versa pour les valeurs de v_i^2.

Les contraintes dures se traduisent alors par les inéquations suivantes : $\forall m+1 \leq i \leq m+h \forall 1 \leq k \leq d_{v_i^1}(1 - x_{v_i^1 k}) + \sum_{l=1}^{d_{v_i^2}} x_{v_i^2 l} \cdot ((k,l) \notin r_i) \geq 1$ et $\forall m+1 \leq i \leq m+h \forall 1 \leq k \leq d_{v_i^2}(1 - x_{v_i^2 k}) + \sum_{l=1}^{d_{v_i^1}} x_{v_i^1 l} \cdot ((l,k) \notin r_i) \geq 1$; avec $((k,l) \notin r_i) = 1$ si le couple d'indice (k,l) n'est pas interdit par la relation r_i et $((k,l) \notin r_i) = 0$ sinon.

Pour les contraintes molles, il suffit d'ajouter une variable booléenne c_i indiquant si la contrainte peut être violée ou non, à l'instar de la traduction vue précédemment. La traduction s'écrit alors : $\forall 1 \leq i \leq m \forall 1 \leq k \leq d_{v_i^1} c_i + (1 - x_{v_i^1 k}) + \sum_{l=1}^{d_{v_i^2}} x_{v_i^2 l} \cdot ((k,l) \notin r_i) \geq 1$ et $\forall 1 \leq i \leq m \forall 1 \leq k \leq d_{v_i^2} c_i + (1 - x_{v_i^2 k}) + \sum_{l=1}^{d_{v_i^1}} x_{v_i^1 l} \cdot ((l,k) \notin r_i) \geq 1$.

Finalement, nous obtenons une nouvelle traduction d'un problème $P = (V, D, C, S, \varphi)$ dans le modèle *PL01* :

$$\left\{\begin{array}{llll}
\text{minimiser} & \sum_{i=1}^{m} \varphi(C(i)) \cdot c_i & & \\
\text{soumis à} & & & \\
 & \sum_{k=1}^{d_j} x_{jk} = 1 & \forall 1 \leq j \leq n & \\
 & (1 - x_{v_i^1 k}) + \sum_{l=1}^{d_{v_i^2}} x_{v_i^2 l} \cdot ((k,l) \notin r_i) \geq 1 & \forall m+1 \leq i \leq m+h & \forall 1 \leq k \leq d_{v_i^1} \\
 & (1 - x_{v_i^2 k}) + \sum_{l=1}^{d_{v_i^1}} x_{v_i^1 l} \cdot ((l,k) \notin r_i) \geq 1 & \forall m+1 \leq i \leq m+h & \forall 1 \leq k \leq d_{v_i^2} \\
 & c_i + (1 - x_{v_i^1 k}) + \sum_{l=1}^{d_{v_i^2}} x_{v_i^2 l} \cdot ((k,l) \notin r_i) \geq 1 & \forall 1 \leq i \leq m & \forall 1 \leq k \leq d_{v_i^1} \\
 & c_i + (1 - x_{v_i^2 k}) + \sum_{l=1}^{d_{v_i^1}} x_{v_i^1 l} \cdot ((l,k) \notin r_i) \geq 1 & \forall 1 \leq i \leq m & \forall 1 \leq k \leq d_{v_i^2} \\
 & x_{jk} \text{ entier} & \forall 1 \leq j \leq n & \forall 1 \leq k \leq d_j \\
 & 0 \leq c_i \leq 1 & \forall 1 \leq i \leq m & \\
 & c_i \text{ entier} & \forall 1 \leq i \leq m &
\end{array}\right. \tag{6.5}$$

Cette nouvelle formulation est plus compacte que la formulation 6.4. En effet, au lieu de L_i inéquations par contrainte, nous en avons au maximum $2 \times d$. Plus la dureté des contraintes est forte (L_i grand), plus notre représentation est compacte par rapport à la formulation 6.4. De plus, nous allons voir l'intérêt de cette formulation dans le cadre d'une relaxation arc-cohérente optimale.

6.1.2.2 Relaxation linéaire

La relaxation linéaire consiste à supprimer la contrainte d'intégrité sur les domaines, les variables entières pouvant prendre des valeurs réelles. Nous avons introduit cette simplification dans le chapitre 5 en la classant dans les simplifications structurelles. Une fois le problème Σ-VCSP P converti en *PL01*, un minorant de P est fourni par l'optimum de la relaxation linéaire. Celle-ci est résolue par l'algorithme du *Simplex*.

Nous avons expérimenté plusieurs outils pour la résolution de problèmes *PLNE*, dont *CPLEX* [CPL95] (algorithmes *Best First Branch and Relax* et *Simplex*), *lp_solve* [BD95] (algorithmes *Branch and Relax* et *Dual Simplex* en matrices creuses) et *opbdp* [Bar96] (algorithme de type *Depth First Branch and Bound* adapté au modèle *PL01*). L'utilisation de *CPLEX* permet de connaître l'optimum de la relaxation linéaire puis d'avoir des minorants croissants au cours du temps grâce au parcours "meilleur en premier" (cf. section 6.2.1).

Les résultats expérimentaux sont hélas très décevants. Par exemple, pour le problème Σ-VCSP binaire aléatoire de la figure 4.20 ($n = 15, d = 10, c = t = 80\%$), sa relaxation linéaire par la formulation 6.5 produit un minorant égal à 564. Au bout de 20 minutes, *CPLEX* retourne un minorant égal à 1500. Or l'optimum du problème vaut 3477 et l'algorithme du *depth first branch and bound* met 23.8 secondes à trouver et prouver cet optimum. Pire, la relaxation linéaire par la formulation 6.4 produit un minorant égal à 5 au bout d'une minute. Par ailleurs, pour le problème *CELAR6-SUB1* (cf. section 5.2.1.4), sa traduction par la formulation 6.5 donne un problème *PL01* avec 916 variables booléennes et 26414 contraintes. Sa relaxation linéaire produit un minorant égal à zéro (calculée en 3097 secondes par *CPLEX* sur un *SUN Sparc10*).

Notons qu'il existe d'autres formes de relaxation pour un problème *PLNE*. La plus connue est la *relaxation lagrangienne* qui consiste à faire passer des contraintes difficiles du système d'inéquation dans la fonction objectif :

$$
\begin{array}{lcl}
\text{Primal} & \Rightarrow & \text{Relaxation lagrangienne} \\
\left\{\begin{array}{ll}
\text{minimiser} & fx \\
\text{soumis à} & \\
 & Ax \geq b \\
 & Cx \geq d \\
 & x \in X \\
 & x \text{ entier}
\end{array}\right.
& &
\left\{\begin{array}{lll}
\text{minimiser} \quad fx + u(Ax - b) & & \\
\Leftrightarrow -ub + \text{minimiser}(f + uA)x & & \\
\text{soumis à} & & \\
\qquad\qquad Cx \geq d & & \\
\qquad\qquad x \in X & & u \in \Re^{+m}
\end{array}\right.
\end{array}
\tag{6.6}
$$

Le paramètre u est un vecteur de valeurs réelles positives. Nous supposons que le fait de retirer le système d'inéquations $Ax \geq b$ aboutit à un problème plus simple à résoudre. Alors le maximum des optimums des relaxations lagrangiennes quel que soit $u \in \Re^{+m}$ est égal à l'optimum du problème initial, appelé *primal*. Pour un paramètre u fixé, c'est un minorant du problème que nous obtenons. Il est de meilleur qualité que celui produit par une simple relaxation linéaire. D'autres techniques faisant aussi passer des contraintes dans l'objectif améliorent encore la qualité des minorants obtenus. C'est le cas par exemple de la décomposition lagrangienne qui produit de meilleurs minorants que la relaxation lagrangienne [NW88].

6.1.2.3 Relaxation arc-cohérente optimale

Nous cherchons à exprimer dans le modèle *PL01* un problème Σ-VCSP avec un objectif différent. Il ne s'agit plus de trouver une relaxation optimale des contraintes pour obtenir un problème cohérent, ce que font les méthodes complètes de résolution, mais seulement une relaxation optimale des contraintes pour obtenir un problème arc-cohérent. La notion d'optimalité signifie une relaxation telle que la somme des coûts des contraintes supprimées soit minimale. [SFV95] démontre que le problème de la relaxation arc-cohérente optimale est NP-complet (cf. théorème 7 de la section 2.2.2.2). Néanmoins, ce problème peut aisément se traduire en *PL01* à partir de la formulation 6.5 puis nous fournir des minorants en résolvant sa relaxation linéaire, à l'instar de l'approche décrite en section 6.1.2.2.

Un problème CSP est arc-cohérent si un renforcement par arc-cohérence sur ce problème n'aboutit pas à un ou plusieurs domaines vides ; c-à-d si toute variable possède une valeur viable. Une valeur, notée x_{jk} dans le modèle *PL01*, est viable ($x_{jk} = 1$) si, quelle que soit la contrainte $C(i)$, elle est supportée par au moins une autre valeur viable dans le domaine de l'autre variable reliée par la contrainte. C'est précisément cette propriété qui est décrite dans la formulation 6.5. La seule différence avec la formulation 6.5 porte sur la contrainte de domaine. L'arc-cohérence impose uniquement qu'il y ai au moins une valeur viable par variable, donc la contrainte se réécrit : $\forall 1 \leq j \leq n,\ \sum_{k=1}^{d_j} x_{jk} \geq 1$.

L'ajout des variables booléennes c_i, pour indiquer que des contraintes sont molles, permet d'exprimer la notion de relaxation arc-cohérente optimale, en passant de l'arc-cohérence en satisfaction à l'arc-cohérence en optimisation. La formulation complète de la relaxation arc-cohérente optimale dans le modèle *PL01* s'écrit donc :

$$\left\{\begin{array}{llll}
\text{minimiser} & \displaystyle\sum_{i=1}^{m} \varphi(C(i)) \cdot c_i & & \\
\text{soumis à} & & & \\
& \displaystyle\sum_{k=1}^{d_j} x_{jk} \geq 1 & \forall 1 \leq j \leq n & \\
& (1 - x_{v_i^1 k}) + \sum_{l=1}^{d_{v_i^2}} x_{v_i^2 l} \cdot ((k,l) \notin r_i) \geq 1 & \forall m+1 \leq i \leq m+h & \forall 1 \leq k \leq d_{v_i^1} \\
& (1 - x_{v_i^2 k}) + \sum_{l=1}^{d_{v_i^1}} x_{v_i^1 l} \cdot ((l,k) \notin r_i) \geq 1 & \forall m+1 \leq i \leq m+h & \forall 1 \leq k \leq d_{v_i^2} \\
& c_i + (1 - x_{v_i^1 k}) + \sum_{l=1}^{d_{v_i^2}} x_{v_i^2 l} \cdot ((k,l) \notin r_i) \geq 1 & \forall 1 \leq i \leq m & \forall 1 \leq k \leq d_{v_i^1} \\
& c_i + (1 - x_{v_i^2 k}) + \sum_{l=1}^{d_{v_i^1}} x_{v_i^1 l} \cdot ((l,k) \notin r_i) \geq 1 & \forall 1 \leq i \leq m & \forall 1 \leq k \leq d_{v_i^2} \\
& x_{jk} \text{ entier} & \forall 1 \leq j \leq n & \forall 1 \leq k \leq d_j \\
& 0 \leq c_i \leq 1 & \forall 1 \leq i \leq m & \\
& c_i \text{ entier} & \forall 1 \leq i \leq m &
\end{array}\right. \tag{6.7}$$

Nous avons testé les mêmes problèmes que dans la section 6.1.2.2 avec l'outil *CPLEX*. La relaxation linéaire de la formulation 6.7 du problème Σ-VCSP binaire aléatoire produit un minorant égal aussi à 564. De même, nous obtenons un minorant nul pour le problème *CELAR6-SUB1*. Les temps de résolution pour atteindre et prouver l'optimum ne semblent pas plus courts que dans le cas de la formulation 6.5. C'est donc une voie de recherche qui ne paraît guère prometteuse. La principale explication semble venir de la transformation trop systématique des problèmes VCSP dans le modèle *PL01*. Une traduction particulière pour un problème donné devrait aboutir à un modèle plus facile à résoudre. C'est le cas pour les problèmes SPOT5 qui sont traduits en *PLNE* (les valeurs des variables étant le numéro de l'instrument de prise de vue) [Ben97]. Leurs relaxations linéaires donnent de bien meilleurs minorants (distants de l'ordre de 10% seulement de l'optimum).

6.2 Méthodes complètes de type *Branch and Bound* adaptées à un cadre temps réel

Nous avons vu que les méthodes complètes de type *Depth First Branch and Bound* fournissent un minorant, égal à l'optimum du problème, uniquement lorsqu'elles terminent. Les minorants calculés durant la résolution ne sont valables que pour des instanciations partielles. Pour déduire à partir de ces minorants d'instanciations partielles, un minorant du problème, il faut modifier la stratégie de parcours en profondeur d'abord de l'arbre de recherche. Une première solution consiste à effectuer un parcours "meilleur en premier". A partir d'une instanciation vide, l'algorithme étend cette instanciation en choisissant toujours celle qui a le plus petit minorant. Ce minorant est donc aussi un minorant du problème complet. C'est la stratégie de l'algorithme *Best First Branch and Bound*, présenté dans la section 6.2.1. Une deuxième solution consiste à brider le parcours en profondeur d'abord, de différentes manières. En itérant des parcours de moins en moins bridés, nous obtenons une succession de minorants croissants. Cette approche est présentée dans la section 6.2.2. Nous présentons dans la section 6.2.3 une combinaison prometteuse d'un algorithme *Russian Doll Search* combiné avec l'une des solutions de la section 6.2.2. Enfin, nous terminons par des expérimentations de ces approches sur des problèmes aléatoires et sur l'instance 6 du CELAR.

Les principaux résultats présentés dans cette section ont été soumis dans [CdGV98]. Les pseudo-codes des algorithmes ont été écrits par Gérard Verfaillie. Les expérimentations, à l'exception de la figure 6.14, ont été faites par Bertrand Cabon, dans le langage C.

Pour la plus part des algorithmes présentés, nous donnons leurs pseudo-codes, de manière la plus uniformisée possible entre les divers algorithmes. De même, pour des raisons de clarté, nous ne présentons que des algorithmes utilisant un ordre statique sur les variables et les valeurs. Lors des expérimentations, une ordre dynamique sur les valeurs et sur les variables (à l'exception des algorithmes *rds* et *idrds*) a été ajouté. Les algorithmes dont le pseudo-code est fourni sont les suivants :

- *Recursive Best First* (*rbf*), présenté en section 6.2.1 ;
- *Iterative Deepening* (*id*), présenté en section 6.2.2 ;
- *Iterative Objective Relaxing* (*ior*), présenté en section 6.2.2 ;
- *Iterative Approximating* (*ia*), présenté en section 6.2.2 ;
- *Russian Doll Search* (*rds*), présenté en section 6.2.3 ;
- *Iterative Deepening + Russian Doll Search* (*idrds*), présenté en section 6.2.3.

Pour s'habituer au type de pseudo-code employé, nous donnons en figure 6.2 le pseudo-code de l'algorithme *Depth First Branch and Bound.*

Les variables globales glb et gub reçoivent le minorant et le majorant courant du problème. Celui-ci est composé de n variables avec $\textsc{ds}(j)$, la taille du domaine de la variable j. Un tableau global, noté A, de dimension n, retient l'instanciation partielle courante.

Un minorant de la valuation globale d'une instanciation partielle est donné par la fonction BOUND. $\textsc{bound}(A, i, j)$ retourne un minorant du sous-problème limité aux variables après i (i inclus) et tel que les variables entre i et j (i et j inclus) soient instanciées par A. Ainsi $\textsc{bound}(A, i, n)$ retourne le nombre de contraintes violées par l'instanciation A des variables après i (i toujours inclus).

6.2.1 Utilisation de l'algorithme *Best First Branch and Bound* en temps limité

La stratégie de parcours du meilleur noeud en premier (cf. section 2.3.1.1, ainsi que [AS94]) offre à tout instant un minorant du problème global. Ce minorant croît au cours de l'exploration

```
DFBB()
    gub ← +∞
    glb ← 0
    SEARCH-DFBB(1)
    glb ← gub

SEARCH-DFBB(j)
    for k = 1 to DS(j)
        A[j] ← k
        lb ← BOUND(A, 1, j)
        if lb < gub
        then if j = n
             then gub ← lb
             else SEARCH-DFBB(j + 1)
```

Fig. 6.2 - *Depth First Branch and Bound*

jusqu'à atteindre l'optimum. Cet algorithme semble être un bon candidat pour produire des minorants croissants en vue d'un encadrement progressif de l'optimum. En revanche, il ne procure aucune solution intermédiaire (instanciation complète sous-optimale) en cours de recherche, sauf à la fin, lorsque les noeuds pendants générés sont des instanciations complètes.

L'inconvénient majeur d'une telle stratégie réside dans son occupation mémoire liée à la liste des noeuds pendants. Le plus souvent, cette liste croît de façon linéaire avec le temps d'exécution et de façon exponentielle avec la taille du problème à résoudre [Kor95]. Typiquement, sur un ordinateur de 300 MIPS[4] avec 1Go d'espace mémoire, si un nouveau noeud pendant est généré toutes les 1000 instructions pour résoudre un problème et qu'un noeud occupe 8 octets, alors la mémoire sera saturée en moins de 8 minutes, à moins que l'optimum ne soit atteint. De plus, lorsqu'un noeud pendant est sélectionné comme noeud courant, il est nécessaire de recalculer des informations intermédiaires, utiles pour construire le minorant courant $lb(A, P)$ (par ex., les $fcvals$ du minorant $lb_{fc}(A, P)$ dans la figure 2.11). Ce recalcul entraîne un ralentissement important de l'exploration arborescente. Il peut être évité en mémorisant d'avantage pour chaque noeud pendant, ce qui risque de saturer d'autant l'espace mémoire.

Pour palier à ces problèmes d'occupation mémoire, plusieurs solutions ont été proposées :

- utilisation d'un algorithme *Depth First Branch and Bound*, avec une limite sur l'exploration arborescente, soit en limitant la profondeur de recherche (cf. *id*, section 6.2.2.1), soit en limitant l'exploration aux noeuds dont la valeur du minorant est inférieure à une certaine borne (cf. *ior*, section 6.2.2.2) [Kor85] ; la limite est augmentée de façon conservative ou non à la fin de chaque exploration ; ces techniques sont décrites dans la section 6.2.2 ;
- utilisation d'un algorithme *Best First Branch and Bound*, puis lorsque la mémoire est saturée :
  - à partir du meilleur noeud pendant, application d'un algorithme *Depth First Branch and Bound* avec une limite de type *ior*, c-à-d en prenant comme limite de minorant, la

[4] MIPS : Million d'Instructions Par Secondes.

deuxième valuation dans la liste des noeuds pendants ; cette liste est à nouveau triée après chaque exploration en profondeur d'abord, en prenant la valuation retournée par l'algorithme *ior* comme nouvelle valeur associée au noeud pendant ;

  – rétraction des noeuds pendants fils du plus mauvais noeud père, celui-ci prenant la valeur minimum de ses fils ; cette approche plus flexible que la précédente enchaîne une succession de rétractions et d'expansions de noeuds qui peut être prohibitive ; des améliorations de cette approche sont décrites dans [Rus92] ;

- utilisation d'une combinaison entre recherche en profondeur d'abord limitée (de type *ior*) et rétraction de noeuds ; cette méthode, appelée *Recursive Best First Search* (*rbf*) [Kor93], est décrite dans la section 6.2.1.1 ;

- utilisation de la mémoire disponible à l'aide d'une recherche bi-directionnelle, appelée *perimeter search* [Man95] ; cela suppose un ordre statique sur les variables ; dans un premier temps, une recherche en arrière et en largeur d'abord est effectuée à partir de la dernière variable jusqu'à quasi-saturation de la mémoire ; puis une recherche en avant et en profondeur d'abord (de type *ior* ou *rbf*) est exécutée à partir de la première variable jusqu'à atteindre la frontière de la recherche en largeur ; celle-ci permet de calculer la valuation globale de l'affectation partielle à la frontière ; la recherche en largeur procure également un minorant du problème, indépendant de celui établi par la recherche en avant.

6.2.1.1 Algorithme *Recursive Best First Branch and Bound*

Le principe de cet algorithme, proposé dans [Kor93], consiste à maintenir sur le chemin allant de la racine au meilleur noeud courant, tous les minorants des valuations globales des noeuds frères. Ainsi, lorsque le minorant courant dépasse le second meilleur minorant, l'algorithme effectue un retour-arrière sur le noeud frère juste inférieur, le plus tôt dans le chemin par rapport au noeud d'arrêt. Ainsi l'occupation mémoire est linéaire, en $O(n \cdot d)$, avec n, le nombre de variables et d, la taille des domaines. L'algorithme découvre les noeuds de la même façon qu'un algorithme *Best First Branch and Bound* classique. En revanche, il peut redévelopper de nombreuses fois les même noeuds lors des retours-arrières. Dans le pire des cas, il parcourt d^{2n-1} noeuds pour résoudre un problème. C'est le cas si tous les noeuds du problème ont un coût unique différent et si deux noeuds de coûts consécutifs dans l'ordre des valuations se trouvent dans un sous-arbre différent partant de la racine.

Nous présentons en figure 6.3 le pseudo-code de l'algorithme *Recursive Best First Branch and Bound* dans sa version simplifiée (notée *Simple Recursive Best First Search* dans [Kor93]). Le minorant d'un noeud est mémorisé dans le tableau global $lb[n, d]$. L'appel fonctionnel $\textsc{best}(j)$ retourne le premier fils du noeud courant qui a le plus petit minorant $lb[j, k]$. La version complète qui a été employée pour les expérimentations réutilise le minorant du noeud père lors de la mise à jour des noeuds fils.

6.2.2 Extension de l'algorithme *Depth First Branch and Bound* au temps limité

L'extension au temps limité porte uniquement sur la production de minorants croissants au cours du temps. L'algorithme *Depth First Branch and Bound* produit un optimum lorsqu'il termine. Il est possible de le brider en profondeur mais pas en largeur pour qu'il produise un minorant plus tôt. Nous présentons trois techniques classiques qui brident soit directement la profondeur de recherche, soit indirectement en limitant la borne supérieure courante de l'algorithme *Branch*

```
RBF()
    gub ← +∞
    glb ← 0
    SEARCH-RBF(1, +∞)

SEARCH-RBF(j, lb)
    for k = 1 to DS(j)
        A[j] ← k
        lb[j, k] ← BOUND(A, 1, j)
        if j = n
        then gub ← min(gub, lb[j, k])
    k* ← BEST(j)
    while lb[j, k*] ≤ lb
        glb ← lb[j, k*]
        if j = n
        then stop
        lb' ← min(lb, min_{k≠k*} lb[j, k]))
        A[j] ← k*
        lb[j, k*] ← SEARCH-RBF(j + 1, lb')
        k* ← BEST(j)
    return lb[j, k*]
```

Fig. 6.3 - *Recursive Best First*

and Bound.

Pour obtenir une succession de minorants croissants, il faut augmenter progressivement la limite imposée à la résolution. Ceci est fait de façon arbitraire (en n étapes) pour les algorithmes *Iterative Deepening* et *Iterative Approximating* et de façon conservative (nombre d'étapes déterminées par les résolutions) pour l'algorithme *Iterative Objective Relaxing.* Visant une approche contractuelle, les travaux sur la planification temps réel en environnement statique (cf. section 1.2.3.2) [CW91, Chu94, WC97] tentent d'adapter cette limite sur la résolution en fonction de l'échéance temporelle. Ils utilisent un modèle mathématique pour prédire le temps de résolution en fonction de la limite. Ce modèle peut être informé par la connaissance des durées des premières itérations. Ensuite, une unique résolution est exécutée avec une limite telle que la résolution produise le meilleur minorant possible dans le temps restant.

6.2.2.1 Algorithme *Iterative Deepening*

C'est la première version de l'algorithme proposé dans [Kor85] que nous appelons *Iterative Deepening*, voir la figure 6.4. Elle correspond bien à exécuter l'algorithme *depth-first branch and bound* avec une limite sur la profondeur d'exploration, notée h, avec h variant de 1 à n. A la fin d'une itération, le plus petit minorant calculé sur les noeuds à la profondeur h est un minorant du problème cible.

```
ID()
    gub ← +∞
    glb ← 0
    for h = 1 to n
        ub ← +∞
        SEARCH-ID(1, h)
        glb ← ub

SEARCH-ID(j, h)
    for k = 1 to DS(j)
        A[j] ← k
        lb ← BOUND(A, 1, j)
        if lb < ub
        then if j = h
             then ub ← lb
                  if h = n
                  then gub ← ub
             else SEARCH-ID(j + 1, h)
```

Fig. 6.4 - *Iterative Deepening*

6.2.2.2 Algorithme *Iterative Objective Relaxing*

L'algorithme *Iterative Objective Relaxing* correspond à la deuxième version présentée dans [Kor85]. Le principe consiste à chercher une solution de coût égal au minorant courant du problème. L'algorithme effectue une série d'hypothèses sur ce minorant, en commençant par la valeur 0.

Si la résolution échoue, le plus petit minorant associé à un noeud d'arrêt devient le minorant courant du problème. Si la résolution trouve une solution, alors cette solution est optimale et l'algorithme termine (voir figure 6.5). Notons que cet algorithme découvre les noeuds suivant le même ordre "meilleur en premier" que l'algorithme *Best First Branch and Bound*. En revanche, dans le pire cas, il parcourt d^{2n} noeuds pour résoudre un problème dont tous les noeuds auraient une valuation différente.

```
IOR()
    gub ← +∞
    glb ← 0
    while gub = +∞
        ub ← +∞
        SEARCH-IOR(1, glb)
        glb ← ub

SEARCH-IOR(j, glb)
    for k = 1 to DS(j)
        A[j] ← k
        lb ← BOUND(A, 1, j)
        if lb ≤ glb
        then if j = n
                then gub ← ub ← lb
                else SEARCH-IOR(j + 1, glb)
        else ub ← min(ub, lb)
```

Fig. 6.5 - *Iterative Objective Relaxing*

Le nombre d'étapes de l'algorithme *ior* n'est pas connu à l'avance. Il peut être particulièrement grand si l'optimum est élevé. Or, à chaque itération, l'algorithme redéveloppe les mêmes noeuds. Puisqu'en pratique, il ne fait progresser le minorant que de 1 en 1 (dans nos expérimentations, les sauts de valuation sont rares et plutôt au début de la recherche), son utilisation semble restreinte à des petits problèmes. L'algorithme *Recursive Best First Branch and Bound* connaît le même handicap, bien qu'il puisse éviter de redévelopper tous les noeuds grâce à ses structures de données. Pour remédier à la profusion d'itérations de l'algorithme *ior*, il faut procéder à une augmentation non conservative du minorant, par exemple, en suivant une suite géométrique de raison 2.

6.2.2.3 Algorithme *Iterative Approximating*

L'algorithme *Iterative Approximating*, proposé dans [Pea84] sous le nom de A^*_ϵ, est d'abord une technique simple pour trouver des bons majorants en utilisant un algorithme *depth-first branch and bound* très peu modifié. La seule différence porte sur la condition de retour-arrière : au lieu de $\text{BOUND}(A, 1, j) \geq gub$, la condition devient $\text{BOUND}(A, 1, j) \geq \epsilon \cdot gub$, avec $0 < \epsilon \leq 1$. Ainsi, dès qu'une meilleure solution est trouvée, l'algorithme tentera ensuite d'en trouver une autre $1/\epsilon$ fois meilleure. Cela peut éviter de perdre du temps sur des solutions de qualité intermédiaire. A la fin, l'algorithme échoue, ne pouvant pas trouver de meilleure solution. Nous en déduisons alors un minorant du problème cible égal à $\epsilon \cdot gub$. Il suffit d'itérer ce processus avec des valeurs

ϵ croissantes pour avoir une succession de minorants croissants. La figure 6.6 décrit l'algorithme *ia* avec $\epsilon = \frac{i}{n}$ et $1 \leq i \leq n$. L'évolution des minorants est proportionnelle au nombre d'étapes, égal à n.

```
IA()
    gub ← +∞
    glb ← 0
    for i = 1 to n
        SEARCH-IA(1, i)
        glb ← gub.i/n

SEARCH-IA(j, i)
    for k = 1 to DS(j)
        A[j] ← k
        lb ← BOUND(A, 1, j)
        if lb < gub.i/n
        then if j = n
                then gub ← lb
                else SEARCH-IA(j + 1, i)
```

Fig. 6.6 - *Iterative Approximating*

6.2.3 Extension de l'algorithme *Russian Doll Search* au temps limité

La résolution efficace des problèmes d'optimisation sous contraintes nécessite d'évaluer au mieux les instanciations partielles, donc de calculer des bons minorants $lb(A, P)$ (cf. section 2.3.1.2). Sur les problèmes que nous avons testés, l'algorithme *Russian Doll Search* est généralement celui qui produit les minorants $lb(A, P)$ les plus pertinents. Bien que produisant aussi des minorants $lb(P)$ du problème cible, l'algorithme *rds* est mal adapté à une utilisation en temps limité. C'est pourquoi nous proposons de le combiner avec une technique qui limite son exploration en profondeur.

6.2.3.1 Algorithme *Russian Doll Search*

A la fin de la section 2.3.1.4, nous avons présenté les principes de l'algorithme *Russian Doll Search*, récemment proposé dans [LV96]. Nous donnons son pseudo-code dans la figure 6.7. Les optima des sous-problèmes imbriqués sont mémorisés dans le tableau global $opt[n + 1]$. Le défaut de cet algorithme, dans un cadre temps limité, est qu'il passe trop de temps à prouver l'optimum de sous-problèmes, au lieu de considérer le problème entièrement. Cela conduit à des minorants assez faibles, car considérant trop peu de variables. De plus, un ordre statique des variables, classées par degré décroissant, implique une progression des minorants des sous-problèmes surtout vers la fin, lorsque les premières variables les plus contraintes sont considérées.

6.2.3.2 Algorithme *Iterative Deepening + Russian Doll Search*

L'algorithme *Russian Doll Search* modifié pour le temps réel est intégré dans un schéma de recherche à profondeur limitée. Cela donne l'algorithme *Iterative Deepening + Russian Doll Search* (voir figure 6.8), initialement proposé par Bertrand Cabon [CdGV98]. Au lieu de calculer

```
RDS()
    gub ← +∞
    glb ← 0
    opt[n + 1] ← 0
    for i = n downto 1
        ub ← +∞
        SEARCH-RDS(i, i)
        glb ← opt[i] ← ub

SEARCH-RDS(j, i)
    for k = 1 to DS(j)
        A[j] ← k
        lb ← BOUND(A, i, j) + opt[j + 1]
        if lb < ub
        then if j = n
              then ub ← lb
                    if i = 1
                    then gub ← ub
              else SEARCH-RDS(j + 1, i)
```

Fig. 6.7 - *Russian Doll Search*

l'optimum des sous-problèmes imbriqués, l'algorithme *idrds* ne calcule que des minorants de ces sous-problèmes, mémorisés dans le tableau global $lb[n+1]$. Ce tableau remplace *opt* pour le calcul des $lb_{rds}(A, P)$. A chaque profondeur maximale h, l'algorithme *idrds* parcourt à profondeur limitée par h les sous-problèmes de l'indice $i = n - h + 1$ jusqu'à l'indice $i = 1$. La profondeur h varie de 1 à n. Ainsi les défauts indiqués dans la section précédente disparaissent : l'algorithme *idrds* considère rapidement toutes les variables, y compris les premières, qui sont souvent les plus contraintes, faisant d'avantage grossir le minorant du problème cible.

6.2.4 Expérimentations

Nos expérimentations montrent le comportement des méthodes spécifiques de résolution face à des problèmes aléatoires ou réels. Aux six algorithmes précédemment décrits, nous leur ajoutons un ordre dynamique sur les valeurs, classées par valeur d'inconsistance $fcval$ (cf. section 2.3.1.3) croissante. Lorsque cela est possible, nous leur ajoutons également un ordre dynamique sur les variables, classées par taille courante des domaines croissante et en cas d'égalité par nombre de contraintes associées décroissant. La fonction BOUND est réalisée par une méthode calculant les *Forward Checking valués* ($fcval$) [FW92] (cf. section 2.3.1.3, figure 2.11).

6.2.4.1 Expérimentations sur des problèmes aléatoires

Les problèmes aléatoires testés sont des MAX-CSP, c-à-d que toutes les contraintes ont le même coût, égal à 1. Les paramètres classiques de génération sont $n = 30$, le nombre de variables, $d = 10$, la taille des domaines, $c = 50\%$, la connectivité du graphe des contraintes et t, la dureté des contraintes. A chaque fois, 100 problèmes sont générés avec les mêmes paramètres. Les figures 6.9, 6.10 et 6.11 représentent l'évolution de la valeur moyenne du minorant en fonction du temps. Chaque algorithme est limité à 300 secondes. Les trois figures sont données pour une

```
IDRDS()
    gub ← +∞
    glb ← 0
    lb[n + 1] ← 0
    for h = 1 to n
        for i = n − h + 1 downto 1
            ub ← +∞
            SEARCH-IDRDS(i, i, h)
            lb[i] ← max(ub, lb[i + 1])
            glb ← max(ub, glb)

SEARCH-IDRDS(j, i, h)
    for k = 1 to DS(j)
        A[j] ← k
        lb ← BOUND(A, i, j) + lb[j + 1]
        if lb < ub
        then if j = i + h − 1
                then ub ← lb
                     if h = n
                     then gub ← ub
                else SEARCH-IDRDS(j + 1, i, h)
```

Fig. 6.8 - *Iterative Deepening + Russian Doll Search*

dureté de valeur différente : $t = 50\%$, $t = 70\%$ et $t = 90\%$. La taille de ces problèmes est telle qu'aucune méthode complète ne termine, même dans un temps important (une semaine). Les optima des problèmes ne sont donc pas connus.

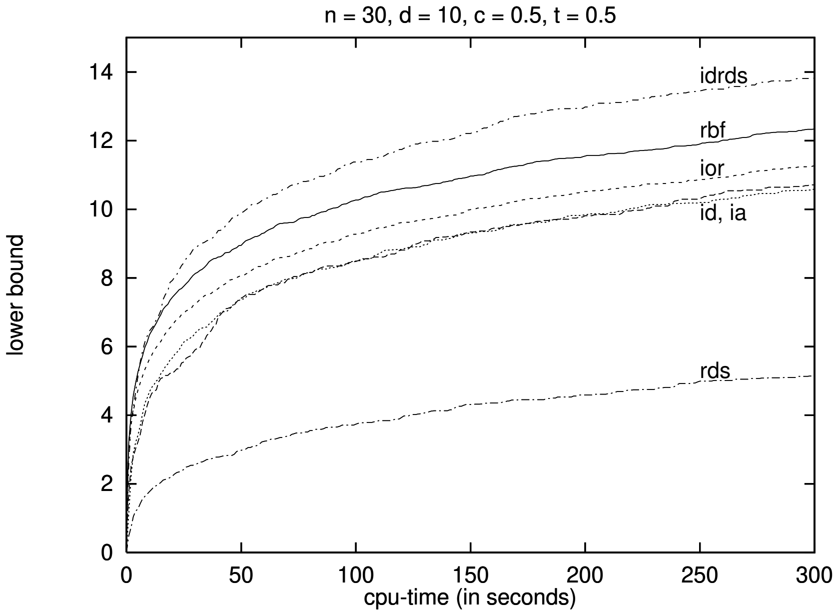


Fig. 6.9 - Comparaison des méthodes complètes en temps limité sur des problèmes Σ-VCSP aléatoires à 30 variables (dureté = 50%)

Les trois figures montrent des résultats similaires :

- l'algorithme *rds* n'est effectivement pas la bonne approche pour produire des bons minorants en temps contraint ;
- les quatre algorithmes *rbf*, *id*, *ior* et *ia* ont un comportement très proche ;
- l'algorithme *idrds* produit à chaque instant les meilleurs minorants.

Notons que la difficulté des problèmes, ainsi que la valuation de leurs optima croît avec la dureté des contraintes. De même, le comportement des quatre algorithmes se ressère et l'efficacité relative de l'algorithme *idrds* augmente, lorsque la dureté croît. A la dureté 90%, au bout de 300 secondes, *idrds* a produit un minorant deux fois plus élevé que les quatre autres algorithmes. Cet écart est dû à la fonction d'évaluation $lb_{idrds}(A, P)$ de *idrds* qui est une information plus forte que $lb_{fc}(A, P)$. Par ailleurs, le comportement similaire des quatre algorithme peut s'expliquer par la trop grande faiblesse de $lb_{fc}(A, P)$.

La figure 6.12 montre la qualité des minorants produits par l'algorithme *idrds* par rapport à des majorants fournis par un algorithme de recuit simulé. Les deux algorithmes sont arrêtés au bout de 300 secondes. La qualité est mesurée simplement par $q = \frac{lb}{ub}$. Nous observons que cette qualité augmente avec la dureté. Ceci doit venir des minorants $lb[j]$ des sous-problèmes emboîtés qui sont plus élevés avec une forte dureté des contraintes.

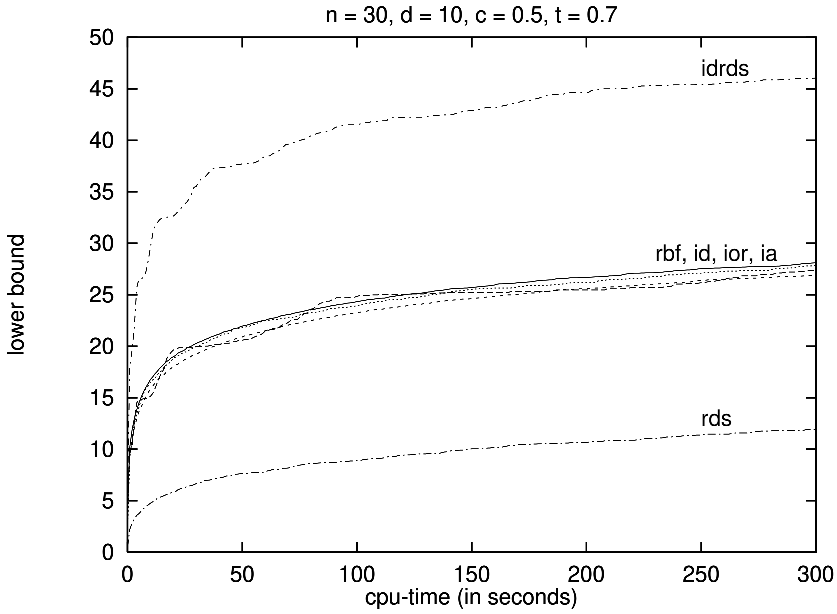


Fig. 6.10 - Comparaison des méthodes complètes en temps limité sur des problèmes Σ-VCSP aléatoires à 30 variables (dureté = 70%)

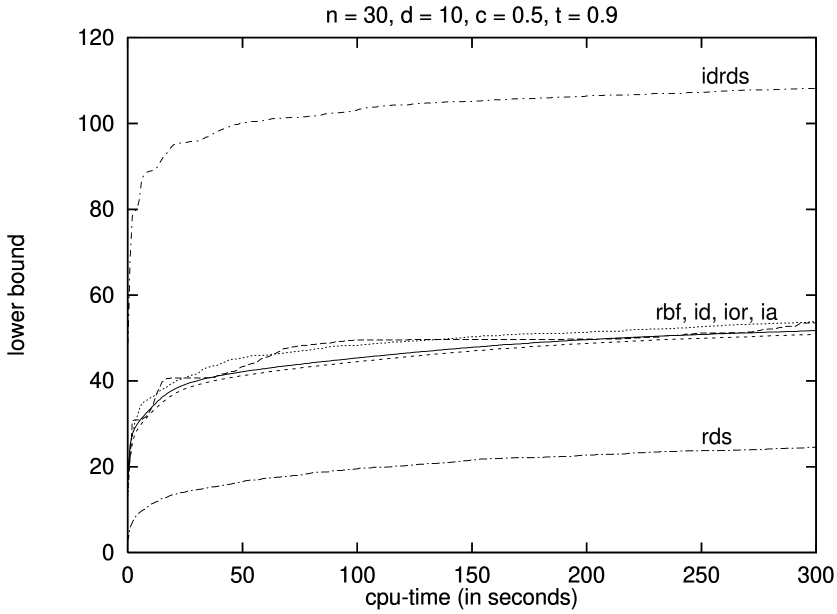


Fig. 6.11 - Comparaison des méthodes complètes en temps limité sur des problèmes Σ-VCSP aléatoires à 30 variables (dureté = 90%)

t	0.5	0.7	0.9
ub	31.24	73.88	135.08
lb	13.82	46.04	108.14
q	44.24%	62.32%	80.06%

Fig. 6.12 - Qualité de l'encadrement de l'optimum produit par un algorithme de recuit simulé et l'algorithme *idrds*

6.2.4.2 Expérimentations sur l'instance 6 du CELAR

Les observations précédentes sur des problèmes aléatoires sont aussi valables sur un problème réel de grande taille (100 variables), l'instance 6 du CELAR. La figure 6.13 montre l'évolution du minorant, avec une limite de temps à 70000 secondes (environ 20 heures), pour l'algorithme *idrds* et l'algorithme *ior*. A la fin du temps imparti, le minorant produit par l'algorithme *ior* vaut seulement 104, alors que celui produit par *idrds* a atteint 2338. L'optimun de l'instance 6 étant égal à 3389, cela fait une qualité du minorant égale à 2338/3389 = 69%. Par ailleurs, les algorithmes *rbf*, *id* et *ia* ont un comportement similaire à *ior*.

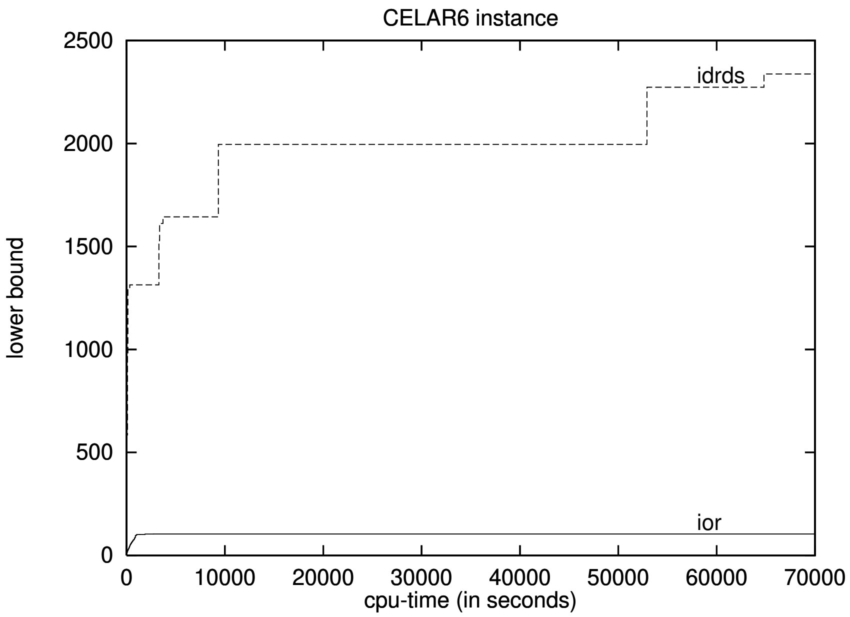


Fig. 6.13 - Comparaison des méthodes *idrds* et *ior* en temps limité sur l'instance 6 du CELAR

6.2.5 Quelques améliorations de l'algorithme *Iterative Deepening + Russian Doll Search*

Arrêt de la procédure SEARCH-IDRDS en fonction des minorants

Lorsque la variable de coupe ub est mise à jour à l'étape i, h, la procédure SEARCH-IDRDS peut être stoppée si jamais $ub = lb[i]$ ou $ub = lb[i+1]$. Dans ce cas, aucune amélioration du minorant n'est possible à cette étape par rapport aux étapes précédentes. Il vaut mieux passer directement à l'étape suivante.

Réutilisation des solutions des résolutions successives

Il est possible d'améliorer le fonctionnement de l'algorithme *idrds* en profitant à l'étape i, h de la solution partielle optimale trouvée à l'étape $i, h-1$. Il suffit de mémoriser, après l'instruction $ub \leftarrow lb$, les instanciations partielles dans un tableau, noté $sol[n, n]$ et indicé par i et h. Ensuite, à l'étape i, h, l'heuristique sur les valeurs est légèrement modifiée de façon à ordonner en premier les valeurs des variables contenues dans la solution $sol[i, h-1]$. Ainsi, l'algorithme trouve plus rapidement des noeuds d'arrêt qui font diminuer ub, ce qui améliore la coupe. En pratique, cette amélioration fait gagner environ un facteur deux en temps de résolution.

Expérimentation sur un sous-problème de l'instance 6 du CELAR et comparaison avec l'enchaînement des simplifications par changement de la structure de valuation

La figure 6.14 compare trois algorithmes appliqués sur le sous-problème CELAR6-SUB1 (cf. section 5.2.1.4) dont l'optimum vaut 2669. La valuation des majorants et minorants est donnée sur une échelle temporelle logarithmique. Le premier algorithme est un recuit simulé qui trouve l'optimum en moins de 20 secondes. Le deuxième algorithme est l'algorithme *idrds* avec les améliorations décrites précédemment. Enfin, le troisième algorithme est fondé sur l'enchaînement des simplifications décrit dans la section 4.3 . Il utilise l'algorithme *rds* pour résoudre chaque simplification (cf. fin de la section 4.4.1). Nous voyons que les deux approches, soit par simplifications structurelles, soit par simplifications de l'objectif, produisent des minorants assez proches, avec un léger avantage pour l'algorithme *idrds* durant la première heure de résolution. De plus cet algorithme a l'avantage de produire plus de résultats intermédiaires.

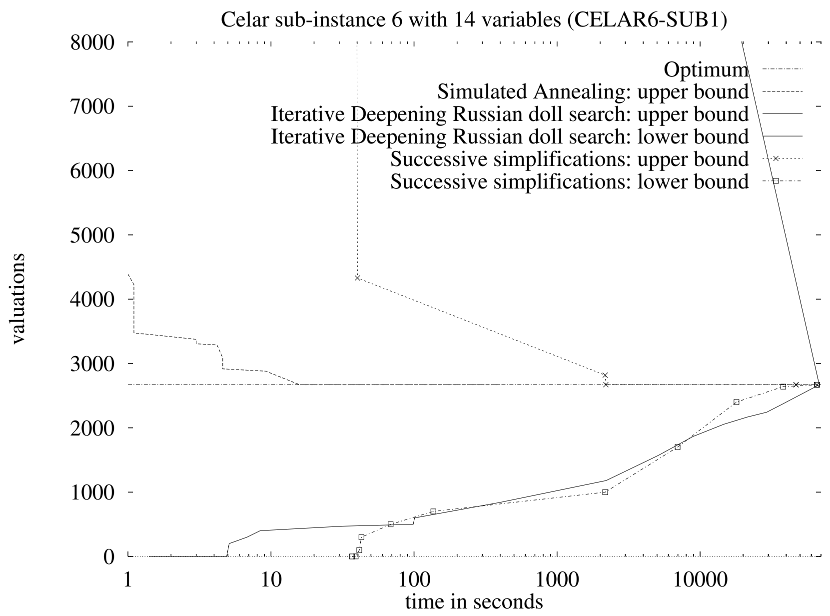


Fig. 6.14 - Comparaison de l'algorithme *Iterative Deepening + Russian Doll Search* avec l'enchaînement des simplifications $\Phi_e^{\otimes}$

Conclusion générale

L'approche délibérative est ambitieuse. Elle vise à concilier deux domaines qui sont celui de l'*Intelligence Artificielle* (*IA*) et celui du *Temps Réel*. Or les problèmes traités en *IA* sont souvent des problèmes de prise de décision NP-durs qui, au pire cas, peuvent être très longs à résoudre complètement et qui en pratique peuvent avoir une variance importante sur leur temps de résolution complète. La solution proposée pour raisonner en temps contraint consiste à accepter des solutions de qualité dégradée. Le nouvel objectif des systèmes temps réel intelligents devient la gestion optimisée des ressources matérielles, tout en respectant les contraintes temporelles. Concrètement cela revient à contrôler le compromis entre temps et qualité des résultats partiels. Le contrôle uniquement de manière statique paraît peu fiable, du fait de la variance des temps de résolution. C'est pourquoi nous optons pour un contrôle dynamique, qui peut se faire sur l'estimation de la qualité courante et aussi sur une prédiction de son évolution.

Les problèmes de prise de décision, au sens de la recherche d'une bonne solution, peuvent se modéliser sous la forme de problèmes d'optimisation sous contraintes. L'étude des Problèmes de Satisfaction de Contraintes Valuées (VCSP) permet de formaliser ces problèmes. Le cadre des VCSP définit un formalisme simple, offrant potentiellement un fort pouvoir d'expression. A partir de là, des algorithmes génériques de résolution sont proposés, avec un souci d'efficacité algorithmique. La contrepartie de la généricité et de l'expressivité est qu'en pratique, cela se traduit par une difficulté algorithmique plus grande que celle des approches d'optimisation *ad-hoc*. Notons également que nous nous plaçons dans le cadre ardu de l'optimisation avec un critère global additif, fréquemment utilisé dans les problèmes réels.

Nos travaux ont consisté à enrichir les algorithmes de résolution existants pour qu'ils soit interruptibles. Nous avons fait le choix de l'approche interruptible qui paraît plus souple que l'approche contractuelle. Cette interruptibilité se traduit par une stratégie de résolution fondée sur un encadrement progressif de l'optimum. Ainsi, les bonnes solutions sont obtenues par un enchaînement de méthodes incomplètes. De plus, le contrôle de l'avancement de la résolution du problème est réalisé à l'aide de minorants du problème obtenus par un enchaînement de méthodes complètes sur des simplifications du problème. La complexité croissante des enchaînements permet une amélioration progressive de l'encadrement. A partir de l'encadrement, nous avons donc défini la notion de qualité normée approchée d'une solution. Nous avons également proposé des critères heuristiques utilisés par un ordonnanceur. L'approche a été validée sur un système simplifié (par une série d'hypothèses) composé de problèmes de planification du satellite SPOT5. De cette étude, il ressort que :

- les gains les plus importants sont obtenus par l'enchaînement des premières méthodes incomplètes ;
- le calcul de la qualité prend du temps à l'exécution ; son application retardée (favoriser les méthodes incomplètes au début) ne procure sur notre système qu'un gain relativement

faible, du fait que l'essentiel du gain a déjà été obtenu ;

- la connaissance de la qualité courante permet de répartir plus astucieusement l'effort de calcul, ce qui offre à notre approche une plus grande robustesse ; néanmoins, cela reste à confirmer par d'autres expérimentations ;

- la connaissance de la qualité courante donne une information intéressante en cas d'interaction avec l'humain (systèmes interactifs interruptibles) ; elle permet de garantir des seuils de qualité, hors contraintes temporelles ;

- notre approche peut garantir l'optimalité des solutions, hors contraintes temporelles ; en temps limité, cela permet de suspendre ou d'arrêter la résolution d'un problème si son encadrement est suffisant.

Nous voyons que le calcul de la qualité n'est intéressant que s'il existe des méthodes performantes pour le faire. Notre apport a consisté à proposer, étudier et expérimenter des solutions techniques. Il y avait également la volonté de rester dans le cadre des VCSP pour fournir des techniques qui soient génériques. Un panorama des approches montre qu'elles utilisent toutes des méthodes complètes associées à des techniques de simplification. Ces simplifications portent soit sur la structure du problème, soit sur les objectifs de la résolution. Notons que toutes les combinaisons de simplifications sont permises !

Une première approche générique, novatrice et motivée par le cadre des VCSP combine des changements sur la structure de valuation avec une relaxation des contraintes par niveau. Les objectifs étaient la garantie, obtenue de façon théorique et expérimentale, que le problème simplifié est effectivement plus simple, ainsi que la possibilité de déduire au mieux des minorants. Nous avons proposé un enchaînement de simplifications qui réalise bien un encadrement progressif de l'optimum, tout en cherchant à réutiliser au mieux les informations des résolutions intermédiaires. Il s'avère efficace pour des problèmes ayant plusieurs valuations différentes de leurs contraintes, des valuations espacées et un niveau de difficulté faible (efficacité des premières résolutions) ou dur (diversification des résolutions). Bien sûr, une telle approche générique subit les mêmes limitations que celles des méthodes complètes qu'elle utilise.

Que faire sur des problèmes de grande taille ? Nos solutions visent à réduire de façon drastique l'espace de recherche, en s'appuyant sur la structure des problèmes (graphe des contraintes et n-uplets autorisés par contraintes) pour effectuer cette réduction. Nous avons proposé deux stratégies, réalisées par des méthodes incomplètes. D'une part, une décomposition des variables en sous-problèmes distincts peut être obtenue soit par une K-partition équilibrée des variables, soit par une heuristique gloutonne fondée sur une analyse des cardinalités calculées pour un ordre particulier des variables. D'autre part, nous effectuons une abstraction sur les domaines en regroupant des valeurs proches vis à vis des contraintes. Ces stratégies, utilisées en pré-traitement d'une résolution complète, sont relativement lourdes, mais permettent d'affronter des problèmes qui autrement serait totalement inaccessibles. La combinaison de ces deux stratégies donne de bons résultats sur un problème tel que l'instance 6 du CELAR.

Pourquoi utiliser toujours la même méthode complète de type *Depth First Branch and Bound* ? En effet, il est possible d'avoir des stratégies fondées sur le principe des méthodes complètes qui produisent uniquement ou en premier des minorants du problème cible. Nous avons observé que les méthodes de renforcement de cohérence étendues à l'optimisation additive ou le passage à des méthodes de relaxation linéaire donnent de mauvais résultats. En revanche, des algorithmes

Branch and Bound avec recherche "meilleur en premier" ou à profondeur limitée donnent de meilleurs résultats. Ils produisent des minorants de manière interruptible. Une combinaison judicieuse des algorithmes *Iterative Deepening* et *Russian Doll Search* a même montré des résultats très encourageants.

D'autres stratégies de simplification seraient à étudier, par exemple, la décomposition des contraintes par arbres couvrants, ou différentes implémentations de l'algorithme *Best First Branch and Bound*. Par ailleurs, est-il est intéressant de combiner toutes les stratégies ? Il n'est pas évident que des simplifications structurelles successives (de moins en moins grossières) soient plus efficaces si elles sont résolues avec des techniques de simplification de l'objectif. En effet, il faut pouvoir déterminer le temps de résolution alloué à chaque résolution intermédiaire. Il reste un travail important pour examiner les bonnes combinaisons de simplification et savoir sélectionner le bon enchaînement de simplifications suivant le type de problème à résoudre.

Nous avons présenté les limitations des méthodes complètes par rapport aux méthodes incomplètes (cf. section 3.1.1.1 et figure 2.21). Pour que la production de minorants obtenus par des méthodes complètes soit en phase avec la production de majorants obtenus par des méthodes incomplètes, il est crucial d'améliorer le mécanisme de coupe des méthodes complètes. Cela nécessite la construction de minorants de la valuation globale d'une affection partielle qui soient plus élevés que ceux développés par les techniques existantes. Il faudra s'interroger dans quelle mesure une approche moins générique, utilisant un formalisme plus riche que celui des VCSP apporterait une amélioration dans le calcul de ces minorants[5].

Un autre défi est celui de la réutilisation de raisonnement d'une résolution à l'autre, lors des enchaînements de simplifications, ou plus généralement, lorsque le problème est légèrement modifié par un événement extérieur imprévu. Il s'agit de réutiliser les solutions précédemment trouvées ou de réutiliser les déductions faites lors de la résolution précédente. Ces techniques sont applicables pour les méthodes complètes et incomplètes. Cette dynamicité dans la résolution devrait offrir des systèmes encore plus souples, réduisant leur charge de calcul par apprentissage dynamique.

L'estimation de la qualité d'une solution est une première étape dans le contrôle dynamique des résolutions de problèmes. L'étape suivante consiste à prédire son évolution. Parmi les voies de recherche, l'approche ANYTIME propose une solution fondée sur des statistiques dépendantes du domaine et apprises préalablement (profils de qualité dynamique probabiliste). Une autre voie, complémentaire, consiste à estimer de manière empirique, par sondage [LL97c], la complexité temporelle d'une méthode complète. Cette estimation permettrait de garantir une allocation suffisante du processeur pour effectuer complètement la résolution, ou bien de la refuser, de façon à traiter un autre problème plus simple.

[5] C'est le même débat qu'il y a entre une approche CLP [Van89], avec une sémantique riche, comparée à une approche CSP, avec un formalisme simple.

Annexe A

Benchmarks Σ-VCSP

Les benchmarks que nous avons employés pour expérimenter et valider nos méthodes sont soit des problèmes générés aléatoirement, soit des problèmes réels venant du domaine de la planification de satellite, les problèmes SPOT5, ou du domaine de l'allocation de fréquence, les problèmes du CELAR. Ces problèmes sont tous mis dans le domaine public, accessibles dans [Lem97b, Lem97a]. Les problèmes ont tous été créés ou convertis dans le formalisme des Problèmes de Satisfaction de Contraintes Valuées (VCSP). Michel Lemaître (*email: lemaitre@cert.fr*), qui maintient un répertoire contenant les problèmes SPOT5 et CELAR, propose également une librairie Common Lisp d'algorithmes pour résoudre tout type de VCSP [LL97a].

A.1 Problèmes Σ-VCSP binaires générés aléatoirement

Notre génération de problèmes aléatoires Σ-VCSP s'inspire des générations standards en CSP [Smi94]. La différence porte sur l'ajout d'un coût associé aux contraintes. Les paramètres de génération sont les suivants :

- n, le nombre de variables du problème ;
- d, le nombre de valeurs par domaine ; les domaines sont tous identiques ;
- c, la connectivité du graphe des contraintes (affichée en pourcentage) ; il y a alors exactement $round(\frac{c \cdot n \cdot (n-1)}{2})$ contraintes binaires qui sont créées aléatoirement parmi les $\frac{n \cdot (n-1)}{2}$ contraintes du graphe complet[1] ;
- t, la dureté des contraintes ; elle est identique pour toutes les contraintes ; pour chaque contrainte, il y a exactement $round(t \cdot d^2)$ couples de valeurs interdits ;
- Ω, la distribution de coûts associés aux contraintes ; elle est exprimée par une liste de k couples composés d'un coût e_i et d'un pourcentage p_i ; la somme des p_i doit être égale à 100% ; pour chaque coût e_i, nous sélectionnons aléatoirement $E(\frac{p_i \cdot c \cdot n \cdot (n-1)}{2})$ contraintes[2] ; le dernier coût e_k recueille éventuellement le surplus de contraintes du fait des divisions entières.

A l'inverse de certains générateurs du domaine CSP, nous n'assurons pas que le graphe des contraintes est connexe. Cela peut être obtenu en générant en premier un arbre aléatoire reliant toutes les variables. En pratique, nous observons que tous nos problèmes sont connexes, sauf

[1] La fonction *round(r)* retourne l'entier arrondi le plus proche du nombre réel r.

[2] La fonction *E()* retourne la partie entière d'un nombre réel.

lorsque la connectivité est très faible (inférieure à 10%).

Notre outil de génération est accessible à l'adresse :
ftp://ftp.cert.fr/pub/lemaitre/LVCSP/Sources/GenRndC/genrndbvcsp.c.
Il utilise la fonction *lrand48()* pour générer un nombre aléatoire. Il possède quelques paramètres supplémentaires. *pchard* (ou h) permet de fixer un pourcentage de contraintes dures (valuation égale à $\top$) par rapport au nombre total de contraintes ($n \cdot (n-1)/2$), les autres contraintes étant molles. *nbinst* permet de séparer la définition du problème (avec génération d'un graphe complet) de la définition de ses *nbinst* instances, chaque instance étant une sélection aléatoire de $round(\frac{c \cdot n \cdot (n-1)}{2})$ contraintes. Cela permet d'expérimenter les méthodes sur *nbinst* instances ayant les mêmes paramètres de génération.

A.2 Problèmes de prises de vue satellitaires (SPOT5)

Les problèmes de programmation journalière des prises de vue du satellite d'observation SPOT5 [VB95, BVA+96, Dag97] proviennent du Centre National d'Etudes Spatiales (CNES). Ce sont des problèmes de planification assez particuliers. Il s'agit de satisfaire au mieux des demandes de prises de vue, tout en respectant les contraintes physiques du satellite. Les prises de vue demandées correspondent à des requêtes de différents clients qui demandent l'image d'une zone de la terre. Chaque zone est alors décomposée en prises de vue élémentaires. Un poids est associé à chaque prise de vue, résultat de l'agrégation de plusieurs critères tels que l'importance du client, l'urgence de la demande, les possibilités ultérieures de réalisation, les prévisions météo...Le satellite comporte trois instruments photographiques, un avant, un central et un arrière. Une prise de vue est soit réalisée par l'un des trois instruments si c'est une prise *mono*, soit par les instruments avant et arrière si c'est une prise *stéréo*. Les contraintes physiques retenues sont :

- non chevauchement et respect des temps de transition entre prises de vue sur un même instrument ;
- non chevauchement des intervalles de démarrage ou d'arrêt de prise de vue sur des instruments différents ;
- limitation du débit instantané ;
- limitation de la capacité de l'enregistreur (contrainte mémoire cumulative éventuellement renouvelable).

La contrainte mémoire sur l'enregistreur n'existe pas lorsque la durée de la planification ne dépasse pas celle d'une orbite. En revanche les problèmes multi-orbites ont cette contrainte ; ils sont aussi beaucoup plus gros (jusqu'à 1000 prises de vue). L'objectif consiste à trouver un sous-ensemble des prises de vue qui soit réalisable et qui maximise la somme des poids associés aux prises de vue sélectionnées.

La modélisation des problèmes SPOT5 en Σ-VCSP consiste à :

- associer une variable v à chaque prise de vue ;
- associer à cette variable un domaine de valeurs d correspondant aux possibilités de réalisation de la prise de vue :
  - $d = \{1, 2, 3\}$ si la prise est *mono* ;

  - $d = \{13\}$ si la prise est *stéréo* ;
- ajouter à chaque domaine d une valeur spéciale de rejet (valeur 0), indiquant que la prise de vue n'est pas sélectionnée ;
- associer à chaque variable v une contrainte unaire molle interdisant la valeur de rejet, de coût égal au poids de la prise de vue associée ;
- traduire, en extension, sous forme de contraintes binaires dures (valuation égale à $\top$) les contraintes de non chevauchement et respect des temps de transition entre prises de vue sur un même instrument ;
- traduire, en extension, sous forme de contraintes binaires ou ternaires dures les contraintes de limitation du débit instantané ;
- traduire, en intention, sous forme d'une contrainte n-aire dure la contrainte de limitation de la capacité de l'enregistreur.

La seule contrainte qui n'est pas prise en compte dans cette modélisation est la contrainte de non chevauchement des intervalles de démarrage ou d'arrêt de prise de vue sur des instruments différents. Celle-ci peut être partiellement ignorée en jouant sur les dates de prise de vue une fois la planification effectuée. Dans de rares cas, cela conduit à des solutions légèrement sous-optimales. Notons que l'ordre initial des variables est primordial. En effet, chaque prise de vue ne possède qu'une seule opportunité de réalisation sur la durée de planification. C'est pourquoi le choix est binaire pour effectuer ou rejeter une prise de vue. Ainsi, l'ordre initial correspond à ordonner les variables selon leur date de début de prise de vue croissante.

Les jeux d'essai fournis par le CNES comportent 498 problèmes, soit 384 mono-orbites, sans contrainte mémoire, et 114 multi-orbites, avec contraintes mémoires. Dans les expérimentations, nous ne considérons que les dix instances SPOT5 numérotées 28, 5, 408, 412, 11, 507, 509, 42, 503 et 505 qui sont toutes mono-orbites. Le tableau suivant donne le nombre de variable, le nombre de contraintes, la somme des poids des prises de vue (ω_i), l'optimum ($\varphi(P_i)$), ainsi que sa conversion en terme d'utilité maximale ($U_I(P_i)$) :

SPOT5	**nbvar**	**nbcontr**	ω_i	$\varphi(P_i)$	$U_I(P_i) = \omega_i - \varphi(P_i)$
28	230	5226	326158	270105	56053
5	309	5621	376	261	115
408	200	2232	9310	6228	3082
412	300	4348	48483	32381	16102
11	364	10108	60598	38478	22120
507	311	5732	42527	27390	15137
509	348	8624	55571	36446	19125
42	190	1394	263117	155050	108067
503	143	635	20209	11113	9096
505	240	2242	34353	21253	13100
Total =	2635	46162	860702	598705	261997

Plusieurs études menées par le CNES et le CERT ont permis d'évaluer différentes stratégies de résolution. [BVA+96] compare des méthodes complètes et incomplètes sur les problèmes avec ou sans contrainte mémoire. Les meilleurs résultats sur les problèmes sans contrainte mémoire sont obtenus par l'algorithme *Russian Doll Search* adapté à la structure des problèmes

SPOT5 (formalisme VVCSP, cf. section 2.1.3.2) [LV96]. Enfin, [Dag97] présente une méthode de résolution bicritère pour traiter les problèmes avec contraintes mémoires, de façon à optimiser les prises de vue et limiter la charge mémoire.

A.3 Problèmes d'affectation de fréquences à des liens radio (CELAR)

Les problèmes d'affectation de fréquences à des liens radio (en anglais, *Radio Link Frequency Assignment Problem* (RLFAP)) ont été proposés par le Centre d'Electronique de l'Armement (CELAR) dans le cadre du projet européen EUCLID CALMA (*Combinatorial Algorithm for Military Applications*). Ce projet a pour objectif d'expérimenter et comparer plusieurs méthodes de résolutions sur des problèmes d'optimisation combinatoire difficiles [THL95].

Le problème d'affectation de fréquence consiste à affecter des fréquences à un ensemble de liens de communication radio définis entre paire de sites, en évitant les interférences. Chaque lien de communication est représenté par une variable dont le domaine correspond à l'ensemble des fréquences disponibles pour le couple émetteur-récepteur. Entre deux liens de communication proches, notés F_1 et F_2, il y a une contrainte de non interférence du type $|F_1 - F_2| > cste_{1,2}$. La constante $cste_{1,2}$ dépend de la proximité des deux liens et aussi de l'environnement géographique. Le calcul de cette constante étant assez grossier, elle est volontairement surdimensionné. Pour chaque paire de sites, il faut établir deux liens radio pour avoir une communication en duplex (paire de variables F_{2k} et F_{2k+1}). Une contrainte technologique impose que les fréquences des liens aller et retour soient à une distance égale à 238.

Les problèmes du CELAR ne sont pas des problèmes de satisfaction purs. En effet, afin de faciliter l'ajout de liens supplémentaires à posteriori, il est intéressant de pouvoir minimiser soit la fréquence maximum utilisée (problème noté SPAN), soit le nombre de fréquences utilisées (problème noté CARD). Il arrive aussi que le problème soit insatisfiable du fait des constantes $cste_{1,2}$ surdimensionnées, bien qu'en réalité il n'y ait pas d'interférence. Dans ce cas, l'objectif consiste à minimiser les interventions des techniciens pour vérifier et corriger les liens si nécessaire sur les sites discriminés. Le problème, noté MAX, revient à minimiser la somme des coûts d'intervention sur les sites. Pour cela, un coût est associé à chaque contrainte de non-interférence.

Ces problèmes sont présentés plus en détail dans [Sch97]. Ils ont également été proposés à la communauté Contraintes sous forme d'un benchmark [CdGL+98], qui contient également nos sous-problèmes de l'instance 6, obtenus dans le chapitre 5 (cf. section 5.2.1.4). Les problèmes du CELAR se modélisent particulièrement bien dans le cadre des VCSP [BSV95]. En effet les différents critères d'optimisation s'expriment soit en CSP classiques, soit en VCSP possibilistes, soit en VCSP additifs. Par ailleurs, les problèmes ont des contraintes binaires non linéaires (disjonctions) et les variables ont des domaines discrets et finis.

Il y a onze instances qui ont été générées à partir d'un unique réseau réel, avec des données simplifiées. Pour certaines des instances, une partie des variables sont préaffectées. Cette préaffectation est définitive (instance 4), ou bien peut être modifiée moyennant un coût supplémentaire (instance 9 et 10). Les instances sont résumées dans le tableau suivant. Nous indiquons le numéro d'instance, le type de problème d'optimisation, le nombre de variables et de contraintes, le nombre de variables préaffectées (entre parenthèse, le nombre de celles définitivement préaffectées), l'optimum du problème (ou un majorant, en italique) et enfin, un rappel sur le type d'optimisation. Ce rappel peut indiquer l'ensemble des coûts différents associés aux contraintes et aussi l'ensemble des coûts associés aux variables préaffectées non définitivement :

CELAR	**type**	**nbvar**	**nbcontr**	**préaffec**	$\varphi(P_i)$	**objectif**
1	CARD	916	5548	0	16	# fréq
2	CARD	200	1235	0	14	# fréq
3	CARD	400	2760	0	14	# fréq
4	CARD	680	3967	280(280)	46	# fréq
5	SPAN	400	2598	0	792	fréq max
6	MAX	200	1322	0	3389	1,10,100,1000
7	MAX	400	2865	0	*343592 ?*	1,100,10000,1000000
8	MAX	916	5744	0	*262 ?*	1,2,3,4
9	MAX	680	4103	680(280)	*15571 ?*	1,10,100,1000 ; 1,10,100,1000
10	MAX	680	4103	680(280)	*31516 ?*	1,2,100,1000 ; 10,100,10000,1000000
11	CARD	680	4103	0	22	# fréq

Il existe une astuce pour diminuer le nombre de variables par deux. En fait, la contrainte reliant deux variables d'une même paire s'avère être une contrainte bijective, du fait des domaines (pour chaque valeur du domaine d'une des deux variables, il existe une et une seule valeur du domaine de l'autre variable qui soit compatible). Ainsi, il est possible de remplacer une paire de deux variables F_{2k} et F_{2k+1} par une seule variable dont le domaine est formé des paires de valeurs compatibles. Les contraintes doivent aussi être modifiées : celles qui portent sur la variable F_{2k} (resp. F_{2k+1}) utiliseront la première (resp. deuxième) composante des paires de valeurs.

Dans nos expérimentations, nous avons uniquement traité l'instance 6 du CELAR qui correspond à un problème modélisable en Σ-VCSP.

Bibliographie

[ABCC+93] Adelantado (M.), Boniol (F.), Cubero-Castan (M.), Hifdi (N.), Lécussan (B.), David (V.) et Porche (R.). – *Projet SATURNE : Modèle de Programmation et Modèle d'Exécution pour un Système Temps-Réel d'Aide à la Décision (Etude DRET).* – Rapport technique n° 1/3447/DERI, CERT, 1993.

[ABCC+94a] Adelantado (M.), Boniol (F.), Cubero-Castan (M.), de Givry (S.) et Porche (R.). – *Projet SATURNE : chaîne de développement d'applications et expérimentation sur le Copilote Electronique.* – Rapport technique n° 1/3474.00/DERI, Toulouse, CERT-ONERA Dépt. d'Informatique, Février 1994.

[ABCC+94b] Adelantado (M.), Boniol (F.), Cubero-Castan (M.), Lécussan (B.), Champigneux (G.) et Lebiannic (Y.). – Modèles synchrones de programmation et d'exécution fondés sur une algorithmique interruptible dédiés à un système temps réel d'aide à la décision. *In : Journée Thématique Temps Réel et Sûreté de Fonctionnement dans les Applications de la Défense.* DRET. – Paris, France, 1994.

[ABdG95a] Adelantado (M.), Boniol (F.) et de Givry (S.). – *Programmation synchrone et algorithmique interruptible pour le développement de systèmes temps réel intelligents.* – Rapport technique, Toulouse, ONERA-CERT, 1995.

[ABdG95b] Adelantado (Martin), Boniol (Frédéric) et de Givry (Simon). – Saturne: a reactive - anytime programming model for intelligent embedded real-time systems. *In : 3rd IEEE Workshop on Parallel and Distributed Real-Time Systems.* – Santa Barbara, California, April 24-26 1995.

[ACP95] Ausiello (G.), Crescenzi (P.) et Protasi (M.). – Approximate solution of NP optimization problems. *Theoretical Computer Science*, vol. 150, 1995, pp. 1–55.

[AdG95] Adelantado (Martin) et de Givry (Simon). – Reactive/Anytime Agents: Towards Intelligent Agents with Real-Time Performance. *In : IJCAI'95 Workshop on Anytime Algorithms and Deliberation Scheduling.* – Montreal, Canada, August 21-25 1995.

[AF96] Adelantado (M.) et Forhan (J.-M.). – *Sytèmes complexes distribués coopératifs.* – Rapport technique n° Volet B, Toulouse, ONERA-CERT, Novembre 1996.

[aH96] auf'm Hofe (Harald Meyer). – Partial Satisfaction of Constraint Hierarchies in Reactive and Interactive Configuration. *In : ECAI'96 Workshop on Non-Standard Constraint Processing.* – Budapest, Hungary, 1996.

[AK89] Aarts (E.) et Korst (J.). – *Simulated Annealing and Boltzmann Machines : a Stochastic Approach to Combinatorial Optimization and Neural Network.* – John Wiley & Sons, 1989.

[AL97] Aarts (E.) et Lenstra (J.) (édité par). – *Local Search in Combinatorial Optimization.* – John Wiley & Sons, 1997.

[AS94] Alliot (J.-M.) et Schiex (T.). – *Intelligence Artificielle & Informatique Theorique.* – Cepadues Editions, 1994, 2nd édition. ISBN : 2-85428-324-4.

[Bar96] Barth (Peter). – *Optimizing with Pseudo-Boolean Davis-Putnam (opbdp).* – Max-Planck-Institut f. Informatik, ftp://ftp.mpi-sb.mpg.de/pub/guide/staff/barth/opbdp/opbdp.tar.Z, 1996.

[BB72] Bertelé (U.) et Brioschi (F.). – *Nonserial Dynamic Programming.* – Academic Press, 1972.

[BCH+96] Bellicha (Amit), Capelle (Christian), Habib (Michel), Kökény (Tibor) et Vilarem (Marie-Catherine). – Exploitation de la relation de substituabilité pour la résolution des CSP. *Revue d'Intelligence Artificielle, numéro spécial contraintes*, vol. 2, 1996.

[BD94] Boddy (M.) et Dean (T.). – Deliberation Scheduling for Problem Solving in Time-Constrained Environments. *Artificial Intelligence*, vol. 67, n° 2, 1994, pp. 245–285.

[BD95] Berkelaar (Michel) et Dirks (Jeroen). – *Mixed Integer Linear Program solver (lp_solve).* – Eindhoven University of Technology, ftp://ftp.es.ele.tue.nl/pub/lp_solve/, 1995.

[Ben94] Benhamou (Belaid). – Study of symmetry in Constraint Satisfaction Problems. *In : Second Workshop on the Principles and Practice of Constraint Programming*, pp. 246–254. – Washington, USA, May 1994.

[Ben97] Bensana (Eric). – *Cadre satisfaction de contrainte valué et programmation linéaire.* – Rapport technique, Toulouse, ONERA-CERT, Mars 1997.

[BFM+96] Bistarelli (Stefano), Fargier (Hélène), Montanari (Ugo), Rossi (Francesca), Schiex (Thomas) et Verfaillie (Gérard). – Semiring-based CSPs and Valued CSPs: Basic Properties and Comparison. *In : Over-Constrained Systems (LNCS 1106, Selected papers from the Workshop on Over-Constrained Systems at CP'95, reprints and background papers)*, éd. par Jampel (M.), Freuder (E.) et Maher (M.), pp. 111–150. – Springer, 1996.

[BFR95] Bessière (Christian), Freuder (Eugene C.) et Régin (Jean-Charles). – Using inference to reduce arc consistency computation. *In : Proc. of IJCAI-95*, pp. 592–598. – Montréal, Canada, 1995.

[BG92] Berry (G.) et Gonthier (G.). – The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, vol. 19, n° 2, 1992, pp. 87–152.

[BMMW89] Borning (A.), Maher (M.), Martindale (A.) et Wilson (M.). – Constraint Hierarchies and Logic Programming. *In : Proc. of ICLP'89.* – Lisbon, Portugal, 1989.

[BMR95] Bistarelli (Stefano), Montanari (Ugo) et Rossi (Francesca). – Constraint solving over semirings. *In : Proc. of IJCAI-95*, pp. 624–630. – Montréal, Canada, 1995.

[Bod91] Boddy (M.). – *Solving Time-Dependent Problems: A Decision-Theoritic Approach to Planning in Dynamic Environments.* – Providence, RI, Thèse de PhD, Brown University, Department of Computer Science, 1991.

[Bon97] Boniol (F.). – *Etude d'une sémantique de la réactivité : variations autour du modèle synchrone et application aux systèmes embarqués.* – Toulouse, France, Thèse de PhD, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, décembre 1997.

[BR96] Bessière (Christian) et Régin (Jean-Charles). – Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. *Proc. of CP-96*, 1996.

[Bre96] Bresina (John L.). – Heuristic-Biased Stochastic Sampling. *In : Proc. of AAAI-96*, pp. 271–278. – Portland, OR, 1996.

[BSV95] Bensana (Eric), Schiex (Thomas) et Verfaillie (Gérard). – *Radio Link Frequency Assignment Problem: a CSP approach.* – Rapport technique, ONERA-CERT, 1995.

[BVA+96] Bensana (E.), Verfaillie (G.), Agnèse (J.C.), Bataille (N.) et Blumstein (D.). – Exact and approximate methods for the daily management of an earth observation satellite. *In : Proc. of the 4th International Symposium on Space Mission Operations and Ground Data Systems (SpaceOps96).* – Munich, Germany, 1996.

[Cab96] Cabon (Bertrand). – *Problèmes d'optimisation combinatoire: évaluation de méthodes de la physique statistique.* – Toulouse, France, Thèse de PhD, ENSAE, 1996.

[CB94a] Charpillet (François) et Boyer (Anne). – Incorporating AI techniques into predictable real-time systems: Reakt outcome. *In : Avignon'94*, pp. 121–134. – Avignon, 1994.

[CB94b] Crawford (James M.) et Baker (Andrew B.). – Experimental results on the application of satisfiability algorithms to scheduling problems. *In : Proc. of AAAI-94*, pp. 1092–1097. – Seattle, WA, 1994.

[CB95] Coustet (C.) et Boniol (F.). – Un apport du modèle fonctionnel à la définition de systèmes temps-réel. *In : 6ème Journées Francophones des Langages Applicatifs, JFLA'95.* – Bois d'Amont, France, 30-31 janvier 1995.

[CdGL+98] Cabon (B.), de Givry (S.), Lobjois (L.), Schiex (T.) et Warners (J.P.). – Benchmark Problems: Radio Link Frequency Assignment. *Submited to Constraints*, 1998.

[CdGV98] Cabon (Bertrand), de Givry (Simon) et Verfaillie (Gérard). – Anytime Lower Bounds for Constraint Optimization Problems. *In : Submitted to ECAI-98.* – Brighton, England, 1998.

[CHPR91] Caspi (P.), Halbwachs (N.), Pilaud (P.) et Raymond (P.). – The synchronous dataflow programming language Lustre. *Proceedings of IEEE, Another Look at Real-time programming*, vol. 79, n° 9, September 1991, pp. 1305–1319.

[Chu94] Chu (Lon-Chan). – *Algorithms for Combinatorial Optimization in Real Time and their Automated Refinements by Genetics-Based Learning.* – Urbana-Champaign, IL, Thèse de PhD, Department of Electrical and Computer Engineering, University of Illinois, 1994.

[CK95] Crescenzi (Pierluigi) et Kann (Viggo). – *A compendium of* NP *optimization problems (ftp://ftp.nada.kth.se/Theory/Viggo-Kann/compendium.ps.Z)*. – Rapport technique, Stockholm, Sweden, Royal Institute of Technology, 1995.

[CLR94] Cormen (Thomas), Leiserson (Charles) et Rivest (Ronald). – *Introduction à l'algorithmique*. – Dunod, 1994.

[Coo89] Cooper (Martin C.). – An optimal k-consistency algorithm. *Artificial Intelligence*, vol. 41, 1989, pp. 89–95.

[CPL95] CPLEX Optimization, Inc. – *CPLEX Linear Optimizer 4.0.2*, 1989-1995.

[CVMB96] Cabon (Bertran), Verfaillie (Gérard), Martinez (David) et Bouret (Paul). – Using Mean Field Methods for Boosting Backtrack Search in Constraint Satisfaction Problems. *In : Proc. of ECAI-96*, pp. 165–169. – Budapest, Hungary, 1996.

[CW91] Chu (Lon-Chan) et Wah (Benjamin W.). – Optimization in real time. *In : Proc. of the Twelfth Real Time Systems Symposium*, pp. 150–159. – Washington, D.C., 1991.

[Dag97] Dago (Pierre). – *Extension d'algorithmes dans le cadre des Problèmes de Satisfaction de Contraintes valués. Application aux systèmes satellitaires d'observation de la Terre*. – Toulouse, France, Thèse de PhD, ENSAE, 1997.

[Dan74] Dano (Sven). – *Linear Programming in Industry*. – Springer-Verlag, 1974, 4 édition.

[DB88] Dean (Thomas) et Boddy (Mark). – An analysis of time-dependent planning. *In : Proc. of AAAI-88*, pp. 49–54. – St. Paul, MN, 1988.

[DD88] Dechter (R.) et Dechter (A.). – Belief Maintenance in Dynamic Constraint Networks. *In : Proc. of AAAI-88*, pp. 37–42. – St. Paul, MN, 1988.

[Dec90] Dechter (R.). – Enhancement Schemes for Constraint Processing : Backjumping, Learning and Cutset Decomposition. *Artificial Intelligence*, vol. 41, n° 3, 1990, pp. 273–312.

[DFP93] Dubois (D.), Fargier (H.) et Prade (H.). – The Calculus of Fuzzy Restrictions as a Basis for Flexible Constraint Satisfaction. *In : Proc. of the 2nd IEEE Conference on Fuzzy Sets*, pp. 1131–1136. – San Francisco, CA, 1993.

[DGP94] Dekneuvel (Eric), Ghallab (Malik) et Philippe (Hervé). – Distributed inference on compiled knowledge for real time distributed systems. *In : Proc. of ECAI-94*, pp. 719–723. – Amsterdam, The Netherlands, 1994.

[dGV97a] de Givry (Simon) et Verfaillie (Gérard). – Bounding the Optimum of Constraint Optimization Problems. *In : Proc. of CP-97*, pp. 405–419. – Schloss Hagenberg, Austria, October 29 - November 1 1997.

[dGV97b] de Givry (Simon) et Verfaillie (Gérard). – Optimum Anytime Bounding for Constraint Optimization Problems. *In : Proc. of the AAAI97 workshop on "Building Resource-Bounded Reasoning Systems"*. – Providence, RI, July 27 1997.

[dGV97c] de Givry (Simon) et Verfaillie (Gérard). – Problèmes d'optimisation sous contraintes : Encadrement ANYTIME de l'optimum. *In : Actes des 3ièmes Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets (JNPC-97)*, pp. 33–39. – Rennes, France, 21-22 avril 1997.

[dK89] de Kleer (J.). – A Comparison of ATMS and CSP Techniques. *In : Proc. of IJCAI-89*, pp. 290–296. – Detroit, MI, 1989.

[DL93a] Decker (Keith) et Lesser (Victor). – *Quantitative Modeling of Complex Computational Task Environments.* – Rapport technique n° CS Technical Report 93-21, University of Masssachusetts, 1993.

[DL93b] Decker (Keith) et Lesser (Victor). – Quantitative modeling of complex computational task environments. *In : Proc. of AAAI-93*, pp. 217–224. – Washington, DC, 1993.

[DMG97] Dauchez (Jean-François), Mouaddib (Abdel-Illah) et Grégoire (Éric). – Primes: Progressive reasoning and intelligent multiple methods system. *In : Proc. of the AAAI97 workshop on "Building Resource-Bounded Reasoning Systems".* – Providence, RI, July 27 1997.

[DP87] Dechter (Rina) et Pearl (Judea). – The cycle-cuset method for improving search performance in AI. *In : Proc. of the 3rd IEEE Conference on AI Applications*, pp. 224–230. – Orlando, FL, 1987.

[DP88] Dechter (Rina) et Pearl (Judea). – Network-based heuristics for constraint-satisfaction problems. *In : Artificial Intelligence*, pp. 1–38.

[DSBL93] Drummond (Mark), Swanson (Keith), Bresina (John) et Levinson (Richard). – Reaction-first search. *In : Proc. of IJCAI-93*, pp. 1408–1414. – Chambery, France, 1993.

[Dv95] Dechter (Rina) et van Beek (Peter). – Local and global relational consistency. *In : PC.*

[DV96] Dago (Pierre) et Verfaillie (Gérard). – Nogood Recording for Valued Constraint Satisfaction Problems. *In : Proc. of ICTAI-96.* – Toulouse, France, 1996.

[Dzi90] Dzierzgowski (D.). – Quatre exemples de langages ou environnements pour le développement de programmes où le temps intervient. *Technique et Science Informatiques*, vol. 9, n° 4, 1990, pp. 289–312.

[FD94] Frost (Daniel) et Dechter (Rina). – In search of the best constraint satisfaction search. *In : Proc. of AAAI-94*, pp. 301–306. – Seattle, WA, 1994.

[FDP95] Fargier (H.), Dubois (D.) et Prade (H.). – Problèmes de satisfaction de contraintes flexibles: une approche égalitariste. *Revue d'Intelligence Artificielle*, vol. 9, n° 3, 1995, pp. 311–354.

[FG95] Feige (Uriel) et Goemans (Michel). – Approximating the value of two prover proof systems, with applications to MAX 2SAT and MAX DICUT. *In : Proc. of 3rd Israel Symp. on Theory of Computing and Systems.* – IEEE Computer Society.

[Fis81] Fishburn (P.). – Subjective expected utility: A review of normative theories. *Theory and Decision*, vol. 13, 1981, pp. 139–199.

[For82] Forgy (Charles L.). – Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, vol. 19, n° 1, 1982, pp. 17–37.

[Fre78] Freuder (E.). – Synthesizing Constraint Expressions. *Communications of the ACM*, vol. 21, 1978, pp. 958–966.

[Fre82] Freuder (E.). – A sufficient Condition for Backtrack-free Search. *Journal of the ACM*, vol. 29, n° 1, 1982, pp. 24–32.

[Fre91] Freuder (Eugene C.). – Eliminating Interchangeable Values In Constraint Satisfaction Problems. *In: Proc. of AAAI-91*, pp. 227–233. – Anaheim, CA, 1991.

[FS95] Freuder (Eugene C.) et Sabin (Daniel). – Interchangeability Supports Abstraction and Reformulation for Constraint Satisfaction. *In: Proc. of the Symposium on Abstraction, Reformulation, and Approximation (SARA-95)*, pp. 62–68.

[FW92] Freuder (Eugene C.) et Wallace (Richard J.). – Partial constraint satisfaction. *Artificial Intelligence*, vol. 58, 1992, pp. 21–70.

[Gar96] Garvey (Alan). – *Design-to-time Real-time Scheduling.* – Amherst, Thèse de PhD, University of Massachusetts, 1996.

[Gas79] Gaschnig (J.). – *Performance measurement and analysis of certain Search Algorithms.* – Pittsburgh, PA, Thèse de PhD, Carnegie-Mellon University, 1979.

[GFN89] Gupta (Anoop), Forgy (Charles) et Newell (Allen). – High-speed implementations of rule-based systems. *ACM Transactions on Computer Systems (TOCS)*, vol. 7, n° 2, 1989, pp. 119–146.

[GH92] Ginsberg (M.L.) et Harvey (W.D.). – Iterative broadening. *Artificial Intelligence*, vol. 55, 1992, pp. 367–383.

[GHHY96] Goldschmidt (Olivier), Hochbaum (Dorit S.), Hurkens (Cor) et Yu (Gang). – Approximation algorithms for the k-clique covering problem. *SIAM Journal on Discrete Mathematics*, vol. 9, n° 3, August 1996, pp. 492–509.

[Gin93] Ginsberg (M.). – Dynamic Backtracking. *Journal of Artificial Intelligence Research*, vol. 1, 1993, pp. 25–46.

[GL93a] Garvey (Alan) et Lesser (Victor). – *A Survey of Research in Deliberative Real-Time Artificial Intelligence.* – Rapport technique n° CS Technical Report 93-84, University of Masssachusetts, 1993.

[GL93b] Garvey (Alan J.) et Lesser (Victor R.). – Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Planning, Scheduling and Control*, vol. 23, n° 6, 1993, pp. 1491–1502.

[GL93c] Glover (F.) et Laguna (M.). – Tabu Search. *In: Modern Heuristic Techniques for Combinatorial Problems*, pp. 70–141. – Oxford, Blackwell Scientific Publishing, 1993.

[GL94] Garvey (Alan) et Lesser (Victor). – A survey of research in deliberative real-time artificial intelligence. *Real-Time Systems*, vol. 6, n° 2, 1994.

[GL95a] Garvey (Alan) et Lesser (Victor). – Design-to-time scheduling and ANYTIME algorithms. *In : IJCAI'95 Workshop on Anytime Algorithms and Deliberation Scheduling.* – Montreal, Canada, August 21-25 1995.

[GL95b] Garvey (Alan) et Lesser (Victor). – *Design-to-time Scheduling with Uncertainty.* – Rapport technique n° CS Technical Report 95-03, University of Massachusetts, 1995.

[Gol89] Goldberg (D.E.). – *Genetic Algorithms in Search Optimisation & Machine Learning.* – Addison-Wesley Publishing Company, 1989.

[GP89] Ghallab (Malik) et Philippe (Hervé). – A compiler for real-time knowledge-base. *In : Proc. Workshop on Artificial Intelligence for Industrial Applications*, pp. 387–393.

[GR96] Génisson (R.) et Rauzy (A.). – Sur les relations algorithmiques entre classes polynomiales du problème SAT et des problèmes de satisfaction de contraintes. *actes de RFIA'96*, 1996, pp. 97–107.

[Gro88] Groupe de réflexion temps réel. – Le temps réel. *Technique et Science Informatiques*, vol. 7, n° 5, 1988, pp. 493–500.

[GS84] Gurari (E.) et Sudborough (I.). – Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem. *Journal of Algorithms*, vol. 5, 1984, pp. 531–546.

[GW93a] Gent (I.) et Walsh (T.). – Towards an Understanding of Hill-Climbing Procedures for SAT. *In : Proc. of AAAI-93*, pp. 28–33. – Washington, DC, 1993.

[GW93b] Goemans (Michel X.) et Williamson (David P.). – A new 3/4-approximation algorithm for MAX SAT. *In : 3rd Conf. on Integer Programming and Combinatorial Optimization.*

[Har95] Harvey (William D.). – *NONSYSTEMATIC BACKTRACKING SEARCH.* – Thèse de PhD, Stanford University, March 1995.

[HE80] Haralick (Robert M.) et Elliott (Gordon L.). – Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, vol. 14, 1980, pp. 263–313.

[HFL96] Hull (David), Feng (Wu-chun) et Liu (Jane W.S.). – Operating system support for imprecise computation. *In : AAAI Fall Symposium on Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*, p. 4p. – Cambridge, Massachusetts, 1996.

[HG95] Harvey (William D.) et Ginsberg (Matthew L.). – Limited discrepancy search. *In : Proc. of IJCAI-95*, pp. 607–613. – Montréal, Canada, 1995.

[HK95] Horvitz (Eric) et Klein (Adrian). – Flexible logical inference : Analysis of beliefs and actions in theorem proving under limited resources. *In : IJCAI'95 Workshop on Anytime Algorithms and Deliberation Scheduling.* – Montreal, Canada, August 21-25 1995.

[HN89] Hérault (L.) et Niez (J.J.). – Neural networks and graph k-partitioning. *Complex Systems*, no3, 1989, pp. 531–575.

[Hor90] Horvitz (Eric). – *Computation and action under bounded resources.* – California, Thèse de PhD, Department of Computer Science and Medecine, Stanford University, 1990.

[HP85] Harel (D.) et Pnueli (A.). – On the development of reactive systems. *In : Logic and Models of Concurrent Systems.* Proc NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems (NATO ASI Series F vol. 13).

[HS95] Hamidzadeh (Babak) et Shekhar (Shashi). – Deadline compliance, predictability, and on-line optimization in real-time problem solving. *In : Proc. of IJCAI-95*, pp. 220–226. – Montréal, Canada, 1995.

[HZ96] Hansen (Eric A.) et Zilberstein (Shlomo). – Monitoring the progress of anytime problem-solving. *In : Proc. of AAAI-96.* – Washington, DC, 1996.

[Ist97] Istrate (Gabriel). – Counting, Structure Identification and Maximum Consistency for Binary Constraint Satisfaction Problems. *In : Proc. of CP-97*, pp. 136–149. – Schloss Hagenberg, Austria, 1997.

[Joh74] Johnson (D.S.). – Optimization algorithms for combinatorial problems. *Computing and Sys. Sci.*, no9, 1974, pp. 256–279.

[K9̈4] Kökény (Tibor). – *Satisfaction de Contraintes dans un Environnement Orienté Objets.* – Montpellier, France, Thèse de PhD, Université Montpellier II, Sciences et Techniques du Languedoc, 1994.

[Kau95] Kautz (Henry). – Efficient propositional reasoning. – 1995. Draft.

[KD95] Kask (Kalev) et Dechter (Rina). – Gsat and local consistency. *In : Proc. of IJCAI-95*, pp. 616–622. – Montréal, Canada, 1995.

[Knu75] Knuth (D.). – Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, vol. 29, n° 129, 1975, pp. 121–136.

[Kol96] Kolen (Antoon). – Results on infeasible instances of the rlfap. – 1996. Personal communication.

[Kor85] Korf (Richard). – Depth-First Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, vol. 27, 1985, pp. 97–109.

[Kor90] Korf (Richard E.). – Real-time heuristic search. *Artificial Intelligence*, vol. 42, 1990, pp. 189–211.

[Kor93] Korf (Richard E.). – Linear-space Best-first Search. *Artificial Intelligence*, vol. 62, 1993, pp. 41–78.

[Kor95] Korf (Richard E.). – Space-efficient search algorithms. *ACM Computing Surveys*, vol. 27, n° 3, september 1995.

[Kra87] Krakowiak (S.). – *principes des systèmes d'exploitation des ordinateurs.* – Bordas, 1987.

[KY89] Keng (Naiping) et Yun (David Y. Y.). – A planning/scheduling methodology for the constrained resource problem. *In: Proc. of IJCAI-89*, pp. 998–1003. – Detroit, MI, 1989.

[Lan92] Langley (Pat). – Systematic and nonsystematic search strategies. *In: Proc First Int. Conf. AI Planning Systems*, pp. 145–152.

[Lau96] Lau (Hoong Chuin). – *Simple Yet Effective Algorithms for Constraint Satisfaction and Related Problems.* – 2-12-1 Ookayama, Meguro-ku, Tokyo 152, Japan, Thèse de PhD, Dept. of Computer Science, Tokyo Institute of Technology, August 28 1996.

[LCH92] Lalanda (P.), Charpillet (F.) et Haton (J.-P.). – Real-time blackboard-based architecture. *In: Proc. of ECAI-92*, pp. 262–266. – Vienna, Austria, 1992.

[Lem97a] Lemaître (Michel). – *Problèmes d'affectation de fréquences à des liens radio (CELAR).* – Rapport technique, ftp://ftp.cert.fr/pub/lemaitre/LVCSP/Pbs/CELAR, ONERA-CERT/DCSDV, 1997.

[Lem97b] Lemaître (Michel). – *Problèmes de prises de vue du satellite SPOT5.* – Rapport technique, ftp://ftp.cert.fr/pub/lemaitre/LVCSP/Pbs/SPOT5, ONERA-CERT/DCSDV, 1997.

[Les94] Lesaint (D.). – Maximal Sets of Solutions for Constraint Satisfaction Problems. *In: Proc. of ECAI-94*, pp. 110,114. – Amsterdam, The Netherlands, 1994.

[Les95] Lesaint (D.). – *Calcul d'ensembles de solutions pour problèmes de satisfaction de contraintes.* – Toulouse, France, Thèse de doctorat, ENSAE, 1995.

[LGGLBLM91] Le Guernic (P.), Gauthier (T.), Le Borgne (M.) et Le Maire (C.). – Programming real time application with Signal. *Proceedings of IEEE, Another Look at Real-time programming*, vol. 79, n° 9, September 1991, pp. 1321–1336.

[LL73] Liu (C. L.) et Layland (J. W.). – Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, vol. 20, n° 1, 1973, pp. 46–61.

[LL97a] Lemaître (Michel) et Lobjois (Lionel). – *A Library of Algorithms for solving (Valued) Constraint Satisfaction Problems.* – Rapport technique n° version 2h, ftp://ftp.cert.fr/pub/lemaitre/LVCSP/Sources/2h.tgz, ONERA-CERT/DCSDV, 1997.

[LL97b] Lobjois (L.) et Lemaître (M.). – Coopération entre méthodes complètes et incomplètes pour la résolution de (V)CSP: une tentative d'inventaire. *In: Actes des 3ièmes Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets (JNPC-97)*, pp. 67–73. – Rennes, France, 1997.

[LL97c] Lobjois (Lionel) et Lemaître (Michel). – *Prédiction de la complexité de problèmes d'optimisation combinatoire.* – Rapport technique, Toulouse, ONERA-CERT, 1997.

[LL98] Lobjois (L.) et Lemaître (M.). – Branch and Bound Algorithm Selection by Performance Prediction. *In: soumis à AAAI-98.*

[LLD+96] Liu (J. W. S.), Liu (C. L.), Deng (Z.), Tia (T. S.), Sun (J.), Storch (M.), Hull (D.), Redondo (J. L.), Bettati (R.) et Silberman (A.). – Perts: A prototyping environment for real-time systems. *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, n° 2, 1996, pp. 161–177.

[LLS+91] Liu (J.), Lin (K.), Shih (W.), Yu (A.), Chung (J.) et Zhao (W.). – Algorithms for Scheduling Imprecise Computations. *IEEE Computer*, vol. 24, n° 5, 1991, pp. 58–68.

[LM95] Larrosa (J.) et Meseguer (P.). – Optimization-based Heuristics for Maximal Constraint Satisfaction. *In: Proc. of CP-95 (LNCS 976)*, pp. 103–120. – Springer, 1995.

[LM96a] Larrosa (J.) et Meseguer (P.). – Expoiting the Use of DAC in MAX-CSP. *In: Proc. of CP-96*, pp. 308–322. – Cambridge, MA, USA, 1996.

[LM96b] Larrosa (J.) et Meseguer (P.). – Phase transition in MAX-CSP. *In: Proc. of ECAI-96*, pp. 1990–1994. – Budapest, Hungary, 1996.

[LMSV98] Larrosa (J.), Meseguer (P.), Schiex (T.) et Verfaillie (G.). – Reversible dac and Other Improvements for Solving Max-CSP. *In: Submitted to ECAI-98*. – Brighton, England, 1998.

[Lob96] Lobjois (Lionel). – *Coopération entre méthodes exactes et inexactes pour la résolution de problèmes de satisfaction de contraintes valuées.* – Rapport technique n° Mémoire de DEA, Toulouse, France, ENSAE, 1996.

[LV96] Lemaître (Michel) et Verfaillie (Gérard). – Russian doll search for solving constraint optimization problems. *In: Proc. of AAAI-96*, pp. 181–187. – Portland, OR, 1996.

[Mac92] Mackworth (A.). – Constraint Satisfaction. *In: The Encyclopedia of Artificial Intelligence*, éd. par Shapiro (S.), pp. 285–293. – John Wiley & Sons, 1992.

[Man95] Manzini (Giovanni). – Bida*: an improved perimeter search algorithm. *Artificial Intelligence*, vol. 75, n° 2, 1995, pp. 347–360.

[MCH92] Mouaddib (Abdel-Illah), Charpillet (François) et Haton (Jean-Paul). – Progressive reasoning and approximation. *In: Proc. AAAI Workshop on Imprecise and Approximate Computation*, pp. 166–170. – San Jose, CA, 1992.

[MGS96] Mazure (B.), Grégoire (E.) et Saïs (L.). – Deux Approches Pour La Résolution du Problème SAT. *In: Actes de la Seconde Conférence Nationale sur la Résolution pratique de Problèmes* NP*-complets (CNPC96)*. – Nancy, France, 1996.

[MH95] McDermott (Drew) et Hendler (James). – Planning: What it is, what it could be, an introduction to special issue on planning and scheduling. *Artificial Intelligence*, vol. 76, 1995, pp. 1–16.

[MHA+95] Musliner (David J.), Hendler (James A.), Agrawala (Ashok K.), Durfee (Edmund H.), Strosnider (Jay K.) et Paul (C.J.). – The challenges of real-time AI. *IEEE Computer*, vol. 28, n° 1, January 1995, pp. 58–66.

[MJPL90] Minton (Steve), Johnston (Mark D.), Philips (Andrew B.) et Laird (Philip). – Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. *In: Proc. of AAAI-90*, pp. 17–24. – Boston, MA, 1990.

[MJPL92] Minton (Steve), Johnston (Mark D.), Philips (Andrew B.) et Laird (Philip). – Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, vol. 58, 1992, pp. 161–205.

[Mon74] Montanari (U.). – Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, vol. 7, 1974, pp. 95–132.

[Mou93] Mouaddib (Abdel-Illah). – *Contribution au raisonnement progressif et temps réel dans un univers multi-agents.* – Thèse de PhD, Université de Nancy I, octobre 1993.

[MV96] Martinez (David) et Verfaillie (Gérard). – Echange de nogoods pour la résolution coopérative de problèmes de satisfaction de contraintes. *In: 2ème Conférence Nationale sur la Résolution Pratique de Problèmes* NP*-complets.* Université de Bourgogne. – Dijon, France, 1996.

[MZ95] Mouaddib (Abdel-Illah) et Zilberstein (Shlomo). – Knowledge-based anytime computation. *In: Proc. of IJCAI-95*, pp. 775–781. – Montréal, Canada, 1995.

[MZ97] Mouaddib (Abdel-Illah) et Zilberstein (Shlomo). – Handling duration uncertainty in meta-level control for progressive reasoning. *In: Proc. of IJCAI-97.* – Japon, 1997.

[NW88] Nemhauser (G.L.) et Wolsey (L.A.). – *Integer and Combinatorial Optimization.* – New York, John Wiley & Sons, 1988.

[Pea84] Pearl (J.). – *HEURISTICS, Intelligent Search Strategies for Computer Problem Solving.* – Addison-Wesley Publishing Company, 1984.

[Pea86] Pearl (J.). – Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, vol. 29, 1986, pp. 241–288.

[Pem95] Pemberton (Joseph C.). – k-best: A new method for real-time decision making. *In: Proc. of IJCAI-95*, pp. 227–233. – Montréal, Canada, 1995.

[Pro93] Prosser (Patrick). – Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 1993, pp. 268–299.

[Pro94] Prosser (Patric). – *An empirical study of phase transitions in binary constraint satisfaction problems.* – Rapport technique, University of Strathclyde, 1994.

[Pro95] Prosser (Patrick). – *MAC-CBJ: maintaining arc consistency with conflict-directed backjumping.* – Rapport technique, Department of Computer Science Research Report 95/177, 1995.

[PS89] Peterson (C.) et Söderberg (B.). – A New Method for Mapping Optimization Problems onto Neural Networks. *International Journal of Neural Systems*, vol. 1, n° 1, 1989, pp. 3–22.

[Ré95] Régin (Jean-Charles). – *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* – Montpellier, France, Thèse de PhD, Université Montpellier II, Sciences et Techniques du Languedoc, 1995.

[RB95] Régin (Jean-Charles) et Bessière (Christian). – *ftp://ftp.lirmm.fr/pub/CSP/mac.tar.gz.* – Rapport technique, 161 rue Ada, 34392 Montpellier Cedex 5, France, LIRMM, 1995.

[RHZ76] Rosenfeld (A.), Hummel (R.) et Zucker (S.). – Scene Labeling by Relaxation Operations. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 6, n° 6, 1976, pp. 173–184.

[Rus92] Russell (Stuart). – Efficient memory-bounded search methods. *In : Proc. of ECAI-92*, pp. 1–5. – Vienna, Austria, 1992.

[RW91] Russel (Stuart) et Wefald (Eric). – *Do the RIGHT Thing. Studing in limited rationality*, chap. 5, pp. 119–157. – MIT Press, 1991.

[Sch92] Schiex (Thomas). – Possibilistic constraint satisfaction problems or "how to handle soft constraints ?". *In : Proceedings of the 8th International Conference on Uncertainty in Artificial Intelligence*, pp. 269–275. – Stanford, 1992.

[Sch94] Schiex (T.). – *Préférences et Incertitudes dans les Problèmes de Satisfaction de Contraintes.* – Rapport technique n° 2/7899 DERA, CERT, 1994.

[Sch97] Schiex (Thomas). – *Les problèmes d'affectation de fréquences du CELAR.* – Rapport technique, http://www-bia.inra.fr/T/schiex/Doc/CELAR.html, Institut National de la Recherche Agronomique, Département de Biométrie et Intelligence Artificielle, 1997.

[SF94] Sabin (Daniel) et Freuder (Eugene C.). – Contradicting conventional wisdom in constraint satisfaction. *In : Proc. of ECAI-94*, pp. 125–129. – Amsterdam, The Netherlands, 1994.

[SFV95] Schiex (Thomas), Fargier (Hélène) et Verfaillie (Gérard). – Valued constraint satisfaction problems: Hard and easy problems. *In : Proc. of IJCAI-95*, pp. 631–637. – Montréal, Canada, 1995.

[SFV97] Schiex (T.), Fargier (H.) et Verfaillie (G.). – Problèmes de satisfaction de contraintes valués. *Revue d'Intelligence Artificielle*, vol. 11, n° 3, 1997, pp. 339–373.

[SH93a] Shekhar (Shashi) et Hamidzadeh (Babak). – DynoraII: A Real-Time Planning Algorithm. *Intl. Journal on Artificial Intelligence Tools (Special issue on real-time AI)*, vol. 2, n° 1, 1993, pp. 93–115.

[SH93b] Shekhar (Shashi) et Hamidzadeh (Babak). – Specification and analysis of real time problem solvers. *IEEE Trans. on Knowledge and Data Engineering*, 1993.

[SH97] Shekhar (Shashi) et Hamidzadeh (Babak). – Sarts: A dependable real time search algorithm. *IEEE Trans. on Knowledge and Data Engineering*, 1997.

[Shi92] Shih (Wei-Kuan). – *Scheduling in Real-Time Systems to ensure graceful degradation: the Imprecise-Computation and the Deferred-Deadline Approaches.* – Urbana-Champaign, IL, Thèse de PhD, University of Illinois, 1992.

[SK93] Selman (Bart) et Kautz (Henry). – Domain-independent extensions to gsat: Solving large structured satisfiability problems. *In: Proc. of IJCAI-93*, pp. 290–295. – Chambery, France, 1993.

[SL92] Shih (Wei-Kuan) et Liu (Jane W.S.). – On-line scheduling of imprecise computations to minimize error. *In: IEEE Real-Time Systems Symposium.* – Los Alamitos, California, 1992.

[SM96] Shrag (R.) et Miranker (D.). – Abstraction and the CSP Phase Transition Boundary. *In: Proc. of the Fourth International Symposium on Artificial Intelligence and Mathematics*, pp. 138–141. – Fort Lauderdale, FL, USA, 1996.

[Smi94] Smith (B.). – Phase Transition and the Mushy Region in Constraint Satisfaction Problems. *In: Proc. of ECAI-94*, pp. 100–104. – Amsterdam, The Netherlands, 1994.

[SP94] Strosnider (J.) et Paul (C.). – A Structured View of Real-Time Problem Solving. *Artificial Intelligence*, vol. 15, n° 2, 1994, pp. 45–66.

[SV94] Schiex (Thomas) et Verfaillie (Gérard). – Nogood Recording for Static and Dynamic Constraint Satisfaction Problem. *International Journal of Artificial Intelligence Tools*, vol. 3(2), 1994, pp. 187–207.

[THL95] Tiourine (S.), Hurkens (C.) et Lenstra (J.). – *A review of algorithmic approaches to frequency assignment problems.* – Rapport technique n° 3507.00/W.E 2.5.4, Eindhoven University of Technology, CERT, 1995.

[Tsa93] Tsang (Edward). – *Foundations of Constraint Satisfaction.* – Academic Press, 1993.

[TV97] Tsang (E.P.K.) et Voudouris (C.). – Fast local search and guided local search and their application to british telecom's workforce scheduling problem. *In: Operations Research Letters*, pp. 119–127. – Amsterdam, Elsevier Science Publishers, March 1997.

[VAL95] Vaessens (R.J.M.), Aarts (E.H.L.) et Lenstra (J.K.). – *A Local Search Template.* – Rapport technique, ftp.win.tue.nl, Eindhoven University, 1995.

[VAL96] Vaessens (R.J.M.), Aarts (E.H.L.) et Lenstra (J.K.). – Job shop scheduling by local search. *INFORMS Journal on Computing*, vol. 8, 1996, pp. 302–317.

[Van89] Van Hentenryck (Pascal). – *Constraint Satisfaction in Logic Programming.* – The MIT Press, 1989.

[van94] van Beek (Peter). – *ftp://ftp.cs.unh.edu/pub/csp/archive/code/library/std_algo_vanBeek/libraries-constraints-in-c.zip.* – Rapport technique, Edmonton, Alberta, Canada T6G 2H1; vanbeek@cs.ualberta.ca, Department of Computing Science, University of Alberta, 1994.

[Van95] Van Hentenryck (Pascal). – Constraint solving for combinatorial search problems: A tutorial. *In: Proc. of CP-95.* – Cassis, France, 1995.

[VB95] Verfaillie (Gérard) et Bensana (Eric). – *Evaluation d'algorithmes sur le problème de programmation journalière des prises de vue du satellite d'observation SPOT5.* – Rapport technique, ONERA-CERT, 1995.

[vD94] van Beek (Peter) et Dechter (Rina). – Constraint tightness versus global consistency. *In: KR.*

[VdG97] Verfaillie (Gérard) et de Givry (Simon). – Algorithmic problems and solutions in the valued constraint satisfaction problem framework. *In: Proc. of the Fifth European Congress on Fuzzy and Intelligent Technologies (EUFIT-97).* – Aachen, Germany, 1997.

[Ver95] Verfaillie (Gérard). – *Intelligence Artificielle et Temps Contraint.* – Rapport technique, CERT, 1995.

[vNM47] von Neumann (J.) et Morgenstern (O.). – *Theory of Games and Economic Behavior.* – Princeton University Press, 1947.

[Vou97] Voudouris (Christos). – *Guided local search for combinatorial optimisation problems.* – Colchester, UK, Thèse de PhD, Department of Computer Science, University of Essex, July 1997.

[VS94] Verfaillie (Gérard) et Schiex (Thomas). – Dynamic backtracking for dynamic constraint satisfaction problems. *In: Proc. of the ECAI94 Workshop on "Constraint Satisfaction Issues Raised by Practical Applications".* – Amsterdam, The Netherlands, 1994.

[VS95] Verfaillie (Gérard) et Schiex (Thomas). – Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes: bilan de quelques approches. *In: Revue d'Intelligence Artificielle*, pp. 269–309. – 1995.

[VT96] Voudouris (C.) et Tsang (E.P.K.). – Partial constraint satisfaction problems and guided local search. *In: Practical Application of Constraint Technology (PACT'96)*, pp. 337–356. – London, UK, April 1996.

[Wal94] Wallace (R.). – Directed Arc Consistency Preprocessing. *In: Proc. of the ECAI-94 Workshop on Constraint Processing (LNCS 923)*, pp. 121–137. – Springer, 1994.

[Wal96a] Wallace (R.). – Analysis of Heuristic Methods for Partial Constraint Satisfaction Problems. *In: Proc. of CP-96*, pp. 482–496. – Cambridge, MA, USA, 1996.

[Wal96b] Wallace (R.). – Enhancements of branch and bound methods for maximal constraint satisfaction problem. *In: Proc. of AAAI-96*, pp. 188–195. – Portland, OR, 1996.

[WB93] Wilson (M.) et Borning (A.). – Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, vol. 16, n° 3, 1993, pp. 277–318.

[WC97] Wah (Benjamin W.) et Chu (Lon-Chan). – Tcgd: A time-constrained approximate guided depth-first search algorithm. *Intl. Journal on Artificial Intelligence Tools*, 1997.

[Wei92] Weiss (Mark Allen). – *Data Structures and Algorithm Analysis.* – Benjamin/Cummings, 1992.

[WF95] Wallace (Richard J.) et Freuder (Eugene C.). – Anytime algorithms for constraint satisfaction and SAT problems. *In : IJCAI-95 Workshop on Anytime Algorithms and Deliberation Scheduling*, pp. 99–103.

[XP93] Xu (Jia) et Parnas (David Lorge). – On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, vol. 19, n° 1, January 1993, pp. 70–84.

[Xuo92] Xuong (Nguyen Huy). – *Mathématiques discrètes et informatique.* – Paris, France, Masson (Logique Mathématique Informatique), 1992. ISBN : 2-225-82621-8.

[Zab90] Zabih (R.). – Some Applications of Graph Bandwidth to Constraint Satisfaction Problems. *In : Proc. of AAAI-90*, pp. 46–51. – Boston, MA, 1990.

[Zil93] Zilberstein (Shlomo). – *Operational Rationality through Compilation of Anytime Algorithms.* – Berkeley, Thèse de PhD, University of California, 1993.

[Zil96] Zilberstein (Shlomo). – Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, vol. 17, n° 3, 1996, pp. 73–83.

Index