Designing limited search algorithms for time constrained combinatorial optimization problems

Jean Jourdan and Simon de Givry and Pierre Savéant
Thomson-CSF, Corporate Research Laboratory
Domaine de Corbeville, 91404 Orsay Cedex
email {jourdan,degivry,saveant}@thomson-lcr.fr

April 29, 1999

Abstract

The objective of our work is the design of search algorithms for combinatorial optimization problems with respect to time constraints. We first analyse the different ways in which a search can be limited in order to identify the parameters for controlling the complexity of the algorithm. These parameters fix the trade-off between speed and completeness. Different strategies are then suggested in order to tune these parameters according to the time left. From a software engineering point of view a modeling language is required. We show the relevance of SaLSA by first rewriting a number of limited search algorithms from the literature and then presenting our self-adjusting strategy in this language. Finally we report on experiments with a real-life application borrowed from the military domain.

1 Introduction

Combinatorial optimization problems encountered in industry are normally tackled with approximate resolution methods which give good results within reasonable running times. These methods fall into two broad categories: constructive and local search. In this paper, we focus on the constructive method which, in the light of constraint programming, can be seen as a collaboration between a thinking process and a guessing process.

The thinking process is responsible for local consistency maintenance performed by deductive reasoning and implemented by a propagation algorithm. The guessing process divides the problem into subproblems by generating hypotheses, thus performing abductive reasoning. Hypothesis generation amounts to creating a search tree and exploring parts of it usually in the prefix order and with the guidance of heuristics.

In addition, the operational context might impose a time limit for solving the problem, particularly in the case of on-line systems and interactive decision support systems.In such contexts, not only should the limit not be exceeded but it is also expected that the solver take maximum advantage of the contract duration. The first point is the problem of estimating the running time of a non-deterministic algorithm. The second point gives rise to the following questions: what are the possible ways to limit the search to a small fraction of the space? given a time limit what strategy will lead to the best solution? Previous work has addressed this point, in particular iterative methods which apply the same algorithm with choice points enlarged at each iteration.

Our objective is to produce from a single scheme an incomplete search algorithm which falls between the greedy and complete extremities and which adapts dynamically to the time left. Clearly, the design and implementation of such an algorithm is going to be highly experimental and from a software engineering point of view a modeling language is needed. Constraint programming is well known for its declarative nature in problem modeling but it has lacked until recently the same feature for resolution. In 1997 Localizer [23] was proposed for local search algorithms followed in 1998 by SaLSA [18] for hybrid algorithms (global and local search).

After having identified the parameters which tune the incompleteness of the algorithm, we suggest a strategy to limit the search to the time left. In order to illustrate SaLSA expressiveness we have rewritten a number of limited search algorithms from the literature. Finally, the paper reports on experiments with a real-life application borrowed from the military domain.

2 SaLSA

SaLSA is a modeling language dedicated to supporting the design and implementation of search algorithms (the guessing machine in the introduction) [3, 18]. Choice points have been reified and can be composed by a set of operators to form regular expressions. These terms are then given to a compiler that generates Claire code, which in turn is compiled into C++ code.

A choice point generates a set of decisions either empty or non empty. Two composition operators are considered: leaf-composition (`*`) and fixpoint-composition (`+`). For instance CXC' is an algorithm where first a decision of C is considered followed by a decision of C'. Power notation is offered to state bounded composition. $\wedge\wedge$ will stands for infinite composition. Two predefined constants serve as terminators: `cont` to keep going for other solutions and `exit` to escape from the search procedure. In order to allow side effects, procedure calls can be composed with terms. `T where f` calls `f` at each internal node of the tree and discards the sub-trees where this evaluation has returned false, or where it has led to a failure. `T` and `f` can also be composed with the `*` and `+` operators.

In the following, a finite domain solver, denoted FD, is assumed to play the role of the thinking machine.

The standard assignment choice point will be used as a common pattern of decisions generator. Any other disjunctive enumerator could be defined such as the dichotomic splitter. The standard assignment choice point comes out as follows:

```
L ::  choice(
  moves post(FD, x = v)
         ordered by increasing heuristic(x,v)
  with x = some(x in Variable | unknown?(value,x)),
       v in domain(x)
  until timeOut?  )
```

A choice definition contains two parts. The first part follows the `moves` keyword and defines the set of transitions representing all branches of the choice. The second part follows the `with` keyword and defines local variables. Such variables may either be bound to a value or be assigned to a set of possible values. Transitions can be conditioned by the `such that` keyword and sorted by the `ordered by` keyword. The `until` keyword applied to a boolean function stops the choice point when the function returns true. In our case, all the algorithms stop at the deadline (`timeOut?` function).

The `post` instruction tells the FD solver a new assignment. In further examples CONTEXT instead of FD will stand for the environment responsible for keeping global backtrackable variables. The above choice point generates as many branches as there are possible values to be substituted for the pair (x, v) in the expression $post(FD, x == v)$. In addition these hypotheses are ordered for each variable x by a heuristic function H.

In order to define the standard depth first branch and bound algorithm, a global variable (`UB`) is needed to store the current value of the cost criterion (`OBJECTIVE`). The `solution` procedure updates the current upper bound of the problem whereas the `prune?` procedure posts the constraint that the cost is now strictly smaller than the best upper bound found so far.

```
prune?() : boolean -> post(FD, Objective < UpperBound), true

solution() -> UpperBound := Objective.value
```

The following expression encodes the algorithm:

```
( ( L where prune?  )∧∧
) + solution + cont
```

Note that we always use an incomplete depth first branch and bound as the search procedure in further examples.

3 Controlling algorithm incompleteness

Multiple ways for limiting the search can be imagined. Given the standard depth first branch and bound algorithm, we suggest three main approaches:

- for a given variable, restrict value enumeration to a subset of its domain according to some heuristic,
- define global limitations relative to a set of variables,
- enforce arbitrary smaller upper bounds.

In the following sections, an attempt to identify parameters to control incompleteness is made. The respective SaLSA expressions given beside will be useful for the writing of parametrized algorithms.

3.1 Incomplete enumeration

Since the heuristic is assumed to sort values in a best order, a subset of it can be extracted either by:

- a cardinality condition (n best ranked): `L / n`,
- a metric condition (maximum distance to the best value):
  such that `heuristic(x,v) <= dist * min({heuristic(x,w) | w in domain(x)})` ,
- clustering values in similarity classes according to some similarity treshold:
  with `v in some(w in cluster / true) / cluster in similarityPartition(x, treshold)` .

3.2 Global limitations

Global conditions can be stated to limit the search:

1. topological conditions: maximum number of nodes: `(T) ! NDMAX`, maximum number of backtrackings: `T bk (incNbBk until endNbBk)`, maximum number of leaves: `T / NBSOLMAX`,
2. cardinality or metric condition on the number of best transitions after a lookahead search: `smallest(T, H, NBSOL, dist)` [18],
3. cumulated number of heuristic discrepancies (see further for the LDS algorithm).

The third condition is at the heart of the Limited Discrepency Search Algorithm [13]. With this scheme heuristic discrepancies for the first nodes are tried earlier than with the standard branch and bound.

3.3 Upper bound convergence speed

Let ϵ $(0 < \epsilon \leq 1)$ denote the upper bound convergence speed in:

```
hprune?() : boolean − >
post(FD, Objective < UpperBound * Inc / MaxDepth), true
```

This condition anticipates on future upper bounds jumping intermediate steps.

4 Tuning incompleteness parameters

Incomplete constructive methods have been developed in order to produce better solutions than a complete method interrupted before its end. Complete and incomplete methods produce better solutions as time passes. Between two incomplete methods, within the same contract duration, we empirically observe that a more incomplete method which ends just in time generally produces better solutions than a less incomplete method which is interrupted before its end.

For an incomplete method, we suggest that the quality of its solutions is related to the degree of incompleteness it reaches (without being interrupted). The degree of incompleteness is controlled by some parameters (cf. section 3) that may be fixed before the search (cf. sections 4.1 and 4.2) or dynamically adjusted during the search (cf. section 4.3).

4.1 Problem-dependent tuning

The problem-dependent tuning of an incomplete algorithm consists in fixing its parameters so that the algorithm will perform well on average for a specific problem and contract duration. The values of these parameters are learned by doing experiments on a benchmark of the problem. This experimentation allows us to assess the relevance of the value heuristics, in order to tune the related parameters (cf. section 3.1 and 3.2).

This can be done by collecting information on good solutions obtained by an incomplete algorithm using iterative tuning (cf. section 4.2) and a large contract duration. This information may be the rank of the chosen values in their sorted domains. If the position for a certain kind of variable or for the variables below a certain depth in the search tree is on average equal to one, then the algorithm could be greedy on those variables. On the other hand, a cardinality condition for incomplete enumeration could be used. The main drawback of problem-dependent tuning is its inability to adapt to a particular problem instance. For example, the algorithm can end long before the deadline, i.e. the end of the contract duration, or it can be interrupted long before its solving time. The problem-dependent tuning can be combined with the methods described in the following paragraphs, by giving maximum values to the parameters.

4.2 Iterative methods and problem-independent tuning

Iterative methods apply the same constructive algorithm with different heuristics or different parameters of incompleteness at each iteration. The parameters are tuned before each search. The first search starts with a large degree of incompleteness, in order to be sure that it will end before the deadline whatever the problem instance is. Like interruptible or anytime algorithms [32, 30], iterative methods use entirely the contract duration, produce better solutions as time passes, until they are stopped at the deadline. In other words, iterative methods have the ability to adapt themselves to any kind of problem instances or contract durations, without doing any prior experimentation.

4.2.1 Iterative methods using several different heuristics

The idea is to rapidly try several different value and variable heuristics by doing greedy searches or very incomplete searches. A way to obtain new heuristics consists in randomly biasing an already existing heuristic [2]. A simpler approach consists in using random value selection [20, 7, 12]. The main drawback of these methods is their inability to improve the quality of the solutions if more time is allowed. This is due to their large degree of incompleteness and their blind searches, in case of random selection. We give two examples of iterative methods using random selection.

Iterative Sampling [20, 7]

Iterative sampling iterates greedy searches with random selection. First, we define an "iterator" choice point, called `FOR(s,e)`, which increments a global variable, called `Inc`, from an initial value `s` to a final value `f`. The instruction `post(Context, Xx := vv)` means that the value of the global variable `Xx` will be restore upon backtrack.

```
Inc := 0

FOR(s,e) :: choice(
         moves post(Context, Inc := i)
               ordered by increasing i
         with i in (s ... e)
         until timeOut?  )
```

`LIS` is a new labelling choice point, like `L` (cf. section 2), except that the domain is randomly ordered.

```
LIS :: choice(
    moves post(FD, x = v)
          ordered by random
    with x = some(x in Variable | unknown?(value,x)),
         v in domain(x)
    until timeOut?  )
```

The algorithm is encoded as:

```
FOR(1,MaxTries)
*
( ( ( LIS / 1 ) where prune?  )∧∧
) + solution + cont
```

Restart Depth First Search [12]

Restart Depth First Search iterates bounded node searches (with a `MaxNd` limit) with random selection.

```
FOR(1,MaxTries)
*
( ( ( LIS where prune?  )∧∧
  ) + solution + cont
) !  MaxNd
```

4.2.2 Iterative methods using a decreasing degree of incompleteness

These methods use a single parametrized search algorithm. They successively apply this algorithm, configured such that to perform first a greedy search and last a complete search. So, the successive searches follow an order of increasing complexity. The last non interrupted search acheives the smallest degree of incompleteness. The main drawback of these methods is the fact that each iteration revisits all the interior[1] nodes of the previous iteration, except when a better upper bound has been found. We give four examples of iterative methods using a decreasing degree of incompleteness.

Iterative Broadening [9]

Iterative Broadening controls the search algorithm incompleteness by using a cardinality condition (`Inc`), limiting domain enumeration. This condition evolves from one to the maximum domain size (`MaxCard`).

```
FOR(1,MaxCard)
*
( ( ( L / Inc ) where prune?  )∧∧
) + solution + cont
```

[1] Some iterative methods do not revisit the leaf nodes of the previous iteration [16, 31].

Limited Discrepancy Search [13]

Each tree search visits all the branches where the cumulated number of heuristic discrepancies is lower or equal to a discrepancy limit (`Inc`). The search makes first discrepancies at the top of the tree. The discrepancy limit evolves form zero to the sum of the domain sizes minus one (`MaxDiscr`).

First, we define a new choice point, called `LLDS`, which visits the first `(Inc + 1)` best values in reverse order (**decreasing** keyword). The discrepancy limit is updated inside the choice point, depending on the position of the chosen value in the sorted domain (fonction `pos(v)`, returns 1 for the best value).

```
LLDS ::  choice(
     moves post(Context, Inc := Inc - pos(v) + 1),
           post(FD, x = v)
           ordered by decreasing heuristic(x,v)
     with x = some(x in Variable | unknown?(value,x)),
          l = sort(domain(x), heuristic(x)),
          i in (1 ... Inc + 1),
          v = l[i]
     until timeOut?  )
```

Then, the following expression encodes the *Limited Discrepancy Search* algorithm:

```
FOR(0,MaxDiscr)
*
( ( LLDS where prune?  )∧∧
) + solution + cont
```

Depth-bounded Discrepancy Search [31]

Depth-bounded Discrepancy Search modifies the iterative deepening approach [15] by adding a greedy search from the nodes at the depth bound. Each search does a complete enumeration for all the variables above the depth bound and follows the heuristic to a leaf node, below the depth bound. At the depth bound, the enumeration does not try the best value because it has been tried at the previous iteration. The depth bound evolves from zero to the number of variables (`MaxDepth`). The following expression encodes the algorithm[2]:

```
FOR(0,MaxDepth)
*
( ( ( L ∧ Inc )
    *
```

[2]For the sake of simplicity, the expression does not take into account the improvement which consists in not re-visiting any nodes at the depth bound.

```
    ( L / 1 )∧∧
  ) where prune?
) + solution + cont
```

Iterative Approximating [24]

Iterative Approximating, which is referred as A^*_ϵ in [24], replaces the backtracking condition of the classical *Depth First Branch and Bound* (OBJECTIVE $\geq$ UpperBound) by a weaker condition (OBJECTIVE $\geq \epsilon$.UpperBound, $0 < \epsilon \leq 1$). At the end of each search, we obtain a problem lower bound (LowerBound) equal to ϵ.UpperBound. We chose $\epsilon = \frac{\texttt{Inc}}{\texttt{MaxDepth}}$ with Inc increasing from one to MaxDepth.

```
hprune?() : boolean − >
post(FD, Objective < UpperBound * Inc / MaxDepth), true
```

We define a new choice point, called FORIA, which reuses the FOR choice point, by using the **explore** keyword, and adds an update to the lower bound after each backtracking of the FOR choice point, by using the **on failure** keyword.

```
FORIA(s,e) ::  choice(
           moves explore FOR(s,e)
           on failure LowerBound := UpperBound * Inc / MaxDepth )
```

The algorithm is encoded as:

```
FORIA(1,MaxDepth)
*
( ( L where hprune?  )∧∧
) + solution + cont
```

4.3 Dynamic tuning

The idea behind dynamic tuning is to adjust the algorithm parameters during the search in order to achieve, without being interrupted, a search with the smallest degree of incompleteness. Therefore, dynamic tuning takes into account the problem instance and the contract duration in order to produce better solutions than the previous static tuning approaches (cf. sections 4.1 and 4.2).

Dynamic tuning looks for a specific set of parameter values implying a search duration shorter than the time left. This means that we must be able to estimate the duration of a search algorithm applied to a particular problem instance. Such an empirical estimation is studied in section 4.4.

Dynamic tuning can be applied to a single or an iterative search algorithm. We present both strategies in section 4.5.

Note that related work considers the selection from a candidate list of the best complete search algorithm for a particular optimization problem instance [21] or for a particular satisfaction problem instance [1]. The candidate search algorithms may use different propagation mechanisms or different value and variable heuristics.

4.4 Search time estimation

We have identified two methods to estimate the number of nodes developed by a search algorithm with a particular set of parameter values, for a particular problem instance:

- estimation of the size of a search tree, based on Knuth's sampling method [14], by iteratively generating random paths in the search tree ; Knuth's method has been improved in different ways by [26] and [5] ; initially applied to satisfaction problems, it has been adapted to optimization problems in [22, 21] ;
- during the search, assessment of the mean depth (n) and the mean number of kept values per domain (d) ; then, computation of the size of a perfect-balanced tree of heigth n and branching factor d explored by the search algorithm without any node pruning; in the case of a simple *Depth First Branch and Bound* with incomplete enumeration, this size is equal to $\frac{d^n-d}{d-1}$; this size is an approximation of the size of the tree effectively explored ; this makes the hypothesis of a uniform node pruning.

The first method gives us an unbiased estimator. Its main drawback is the possible computation time overhead, especially when the contract duration is short and when many sets of parameter values have to be tested. We will use the second simpler method in section 4.5.2. Both methods provide an estimation of the search duration by multiplying the number of nodes by the mean time to explore a node. We make the hypothesis that node propagation is done in near constant time, which is not true when there are global constraints.

A third method has been identified. During the search, we compute the current percentage of the search space already explored compared to the total search space:

```
%OfSearchDone() : float -> (
   s := 0,
   p := 1,
   for x in {x in Variable | known?(value,x)}
       ( p := p * actualDomainSize(x),
         s := s + ( min( pos(x.value), actualDomainSize(x) ) - 1 ) / p
   s + 1 / p )
```

The function `actualDomainSize(x)` returns the number of values that will be tested for the variable `x` by the incomplete search algorithm. The result of this function depends on the current domain size and the parameter values. The function `pos(x.value)` returns the rank of the current chosen value in the sorted domain of `x` (cf. section 4.2.2). At any time, with the current percentage of search done (r) and the current consumed time (t), we can deduce the estimated remaining time: $\frac{t.(1-r)}{r}$. This makes also the hypothesis of a uniform node pruning. This last method is fast enough (linear in the number of variables) to be regularly called during the search. A bad estimation done for a while can possibly be corrected by better estimations done later.

4.5 Tuning strategies

We have experimented two different tuning strategies. The first strategy dynamically adjusts the size of a single search tree by adapting the incompleteness parameters during the search (cf. section 4.5.1). The second strategy skips some unnecessary steps of iterative methods using a decreasing degree of incompleteness (cf. section 4.5.2).

Related work: [17, 27, 28] dynamically adjust the depth of a look-ahead search, inserted in a *Best First Search* algorithm ; [6, 29] dynamically adjust the convergence speed parameter of the *Iterative Approximating* algorithm.

4.5.1 Self-adjusting Depth First Branch and Bound Algorithm

Given a set of parameters, we must define a strategy to dynamically select the best set of parameter values in order to end the search near the deadline. The number of possible parameter values may be huge. It is impossible to test all the combinations. Some different combinations may imply the same search duration. Moreover, some combinations may be better than others, because the resulting search algorithm produces on average better solutions for a set of problem instances. Thus, we propose to establish a list of interesting sets of parameter values, by doing experiments. We define a tuning strategy, named S, as an ordered array containing this list of sets.

In order to speed up the tuning process, this array must respect the following property:

$$\forall I,\ \ i < j \quad \Rightarrow$$
$$\textsc{estimatedRemainingTime}(I, S[i]) \leq \textsc{estimatedRemainingTime}(I, S[j])$$

The function $\textsc{estimatedRemainingTime}(I, S[i])$ returns an estimation of the remaining search duration (cf. third method in section 4.4) with $S[i]$, the selected set of parameter values and I, a problem instance. In practice, the above property will be respected if the list S contains monotonicaly increasing parameter values. Such an ordered list S allows a dichotomic (in $\log_2 m$ steps, with m, the length of S) approach in order to find the best set of parameter values.

Self-Adjusting Depth First Branch and Bound Algorithm [10]

For example, we present a self-adjusting *Depth First Branch and Bound* algorithm. The two incompleteness parameters are a cardinality (`Inc`) and a metric (`Dist`) conditions. The tuning process is activated at each backtracking (`adjustParameters`).

```
LA :: choice(
   moves post(FD, x = v)
         such that heuristic(x,v) <= Dist * hmin
         ordered by increasing heuristic(x,v)
   with x = some(x in Variable | unknown?(value,x)),
        hmin = min({ heuristic(x,w) | w in domain(x)}),
        v in domain(x)
   on failure adjustParameters
   until timeOut?  )
```

The following expression encodes the algorithm:

```
( ( ( LA / Inc ) where prune?  )^^
) + solution + cont
```

4.5.2 Self-adjusting Limited Discrepancy Search

Self-Adjusting Limited Discrepancy Search skips a *Limited Discrepancy Search* iteration when a condition (`ldsJump?`) becomes true. This condition is regularly tested during each iteration, but also from one iteration to an other. Thus, some intermediate discrepancy levels will be interrupted or not executed. The condition estimates the time to end the next iteration (with a discrepancy equal to the current discrepancy level plus one) and return true if this time is lower than the remaining time to deadline.

```
ldsJump?() : boolean ->
estimateLds(Inc + 1, DepthSum / NbBk) * elapsedTime() / NbNd <= remainingTime()

noLdsJump?() : boolean -> post(Context, Inc := Inc + 1), not(ldsJump?())
```

The function `estimateLds(d,n)` returns the maximum number of nodes that a *Limited Discrepancy Search* algorithm could explore with a tree of height n and branching factor d (without any node pruning, cf. second method in section 4.4). In order to speed up this function, we pre-compute its values and use an array to access them. The function `elapsedTime` returns the current consumed time. The function `remainingTime` returns the current remaining time to deadline.
We define a new choice point, called `LALDS`, which reuse the `LLDS` choice point and maintains the number of visited nodes (`NbNd`), the number of backtrackings (`NbBk`), the current tree depth (`Depth`) and the current mean depth (`DepthSum / NbBk`):

```
DepthSum := MaxDepth
NbBk := 1
NbNd := MaxDepth

LALDS ::  choice(
      moves NbNd := NbNd + 1,
             post(Context, Depth := Depth + 1),
             explore LLDS
      on failure DepthSum := DepthSum + Depth, NbBk := NbBk + 1
      until ldsJump?  or timeOut?  )
```

The following expression encodes the *Self-Adjusting Limited Discrepancy Search* algorithm[3]:

```
( ( L / 1 )∧∧    U    ( ( FOR(1,MaxDiscr) where noLdsJump?  )
                        *
                        ( LALDS where prune?  )∧∧
                      )
) + solution + cont
```

5 Experimentation

Several experiments have been performed on a weapon allocation problem. Given a stock of available missiles and a set of reachable threats, the system repeatedly computes an engagement plan, i.e. a set of threat-missile assignments each stamped with a shooting time. Missile activation is made up of different steps and requires specific resources. A missile can be assigned to a threat if the launching time is within the temporal window in which the threat is reachable. Optimization criterion minimizes the number of non assigned threats.
The model has been implemented with the Eclair finite domain constraint solver [19]. We have experimented some of the parametrized algorithms described in previous sections. Tuning applies on missile assignment and resource allocation (cardinality and metric conditions).

A global comparison between different algorithms has been performed. The figure 1 shows the criterion obtained by five incomplete search algorithms for different contract durations. The mean criterion has been computed after two thousand problem instances. The greedy algorithm goes straight on with heuristics. The Iterative Broadening algorithm is complete on the resource choice point but open progressively the missile choice point with a cardinality condition. The Self-Adjusting Branch and Bound algorithm applies cardinality and metric conditions on the missile choice point and a metric condition on the resource choice point. The Limited Discrepancy Search and Self-Adjusting Limited Discrepancy Search algorithms consider the missile and resource choice points.

[3]The U binary operator applied to two SaLSA terms means that both terms will be sequentially explored.

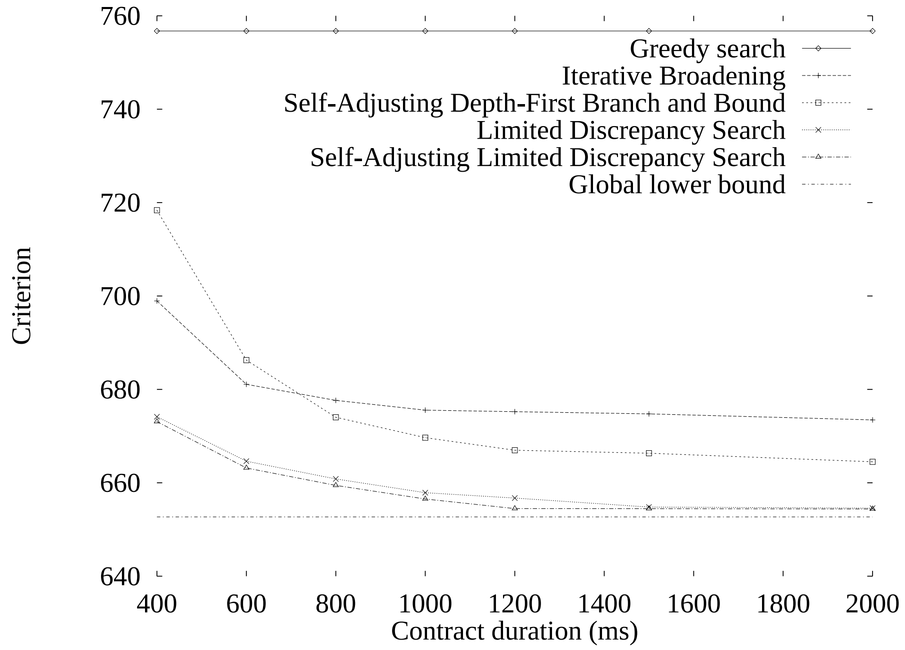

Figure 1: Global comparison

The analysis of experimentation results shows:

- the greedy algorithm always ends before 400 milliseconds, thus does not benefit from longer contract duration;
- the *Self-Adjusting Branch and Bound* algorithm performs better than the *Iterative Broadening* algorithm, although both algorithms are using quite the same kind of incompleteness parameters (incomplete enumeration); the *Self-Adjusting Branch and Bound* algorithm opens choice points more selectively than the *Iterative Broadening* algorithm because more parameters are under control;
- the *Limited Discrepancy Search* and *Self-Adjusting Limited Discrepancy Search* algorithms are performing very well; after a second and a half, their results are closed to the global lower bound; before that time, the dynamic tuning procures only a small improvement compared to the standard *Limited Discrepancy Search* algorithm.

6 Conclusions

Finally, a pragmatic approach to time constrained combinatorial optimization is the following:

1. produce an application domain dependent tuning by learning on a testbed,
2. refine and validate the previous tuning with an iterative strategy,
3. then try a self-adjusting strategy.

The static tuning give indications on which choice points should be opened or not. The resulting algorithm takes into account the application domain, the time limit and the problem instance.

In this paper, only depth first search algorithms have been considered. The reason is to avoid memory allocation problems encountered with breadth first search algorithms. On the other hand, parametrizing lookahead search is of course a possibility we did not try yet. Such a scheme will enable best first search by better informing the heuristics [8]. As a matter of fact, in [17, 11, 25] real time planning problems are addressed with a lookahead search which depth depend on a time contract.

In this paper, we have proposed a strategy to tune dynamically the complexity of a global search algorithm according to the time left. The same approach could be considered for local search algorithms as in [1] for tabu search and as in [4] for simulated annealing. We would then be in position to propose self-adjusting hybrid algorithms within the SaLSA modeling language.

References

[1] J. Allen and S. Minton. Selecting the right heuristic algorithm: Runtime performance predictors. In *Proceedings of the Canadian AI Conference*, 1996.

[2] John L. Bresina. Heuristic-Biased Stochastic Sampling. In *Proc. of AAAI-96*, pages 271–278, Portland, OR, 1996.

[3] Yves Caseau and François Laburthe. Salsa: A specification language for search algorithms. Technical Report LIENS report 97-11, Ecole normale supérieure, Paris, France, 1997.

[4] Irène Charon and Olivier Hudry. Réglage totalement automatique du bruitage. In *Proc. of ROADEF-99*, pages 120–121, Autrans, France, 1999.

[5] P. Chen. Heuristic Sampling : a Method for Predicting the Performance of Tree Searching Programs. *SIAM Journal on Computing*, 21(2):295–315, 1992.

[6] Lon-Chan Chu. *Algorithms for Combinatorial Optimization in Real Time and their Automated Refinements by Genetics-Based Learning*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, IL, 1994.

[7] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of AAAI-94*, pages 1092–1097, Seattle, WA, 1994.

[8] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proc. of IJCAI-95*, pages 572–578, Montréal, Canada, 1995.

[9] M.L. Ginsberg and W.D. Harvey. Iterative broadening. *Artificial Intelligence*, 55:367–383, 1992.

[10] Simon de Givry, Pierre Savéant, and Jean Jourdan. Optimisation combinatoire en temps limité : Depth first branch and bound adaptatif. In *Proc. of JFPLC-99*, Lyon, France, 1999.

[11] Babak Hamidzadeh and Shashi Shekhar. Deadline compliance, predictability, and on-line optimization in real-time problem solving. In *Proc. of IJCAI-95*, pages 220–226, Montréal, Canada, 1995.

[12] William D. Harvey. *NONSYSTEMATIC BACKTRACKING SEARCH*. PhD thesis, Stanford University, March 1995.

[13] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proc. of IJCAI-95*, pages 607–613, Montréal, Canada, 1995.

[14] D. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[15] Richard Korf. Depth-First Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

[16] Richard Korf. Improved limited discrepancy search. In *Proc. of AAAI-96*, Portland, OR, 1996.

[17] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[18] François Laburthe. SaLSA: a language for search algorithms. In *Proc. of CP-98*, pages 310–324, Pisa, Italy, October 26-30 1998.

[19] François Laburthe, Pierre Savéant, Simon de Givry, and Jean Jourdan. Eclair: A library of constraints over finite domains. Technical Report technical report ATS 98-2, Thomson-CSF LCR, Orsay, France, 1998.

[20] Pat Langley. Systematic and nonsystematic search strategies. In *Proc First Int. Conf. AI Planning Systems*, pages 145–152, 1992.

[21] L. Lobjois and M. Lemaître. Branch and Bound Algorithm Selection by Performance Prediction. In *Proc. of AAAI-98*, pages 353–358, Madison, WI, 1998.

[22] L. Lobjois and M. Lemaître. Adaptation de l'estimateur de knuth aux problèmes d'optimisation combinatoires. In *Proc. of RJCIA-98*, pages 171–179, Toulouse, France, 1998.

[23] Laurent Michel and Pascal Van Hentenryck. Localizer: A modeling language for local search. In *Proc. of CP-97*, pages 237–251, Schloss Hagenberg, Austria, 1997.

[24] J. Pearl. *HEURISTICS, Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company, 1984.

[25] Joseph C. Pemberton. k-best: A new method for real-time decision making. In *Proc. of IJCAI-95*, pages 227–233, Montréal, Canada, 1995.

[26] P. Purdom. Tree Size by Partial Backtracking. *SIAM Journal on Computing*, 7(4):481–491, 1978.

[27] Stuart Russel and Eric Wefald. *Do the RIGHT Thing. Studing in limited rationality*, chapter 5, pages 119–157. MIT Press, 1991.

[28] Shashi Shekhar and Babak Hamidzadeh. Sarts: A dependable real time search algorithm. *IEEE Trans. on Knowledge and Data Engineering*, 1997.

[29] Benjamin W. Wah and Lon-Chan Chu. Tcgd: A time-constrained approximate guided depth-first search algorithm. *Intl. Journal on Artificial Intelligence Tools*, 1997.

[30] Richard J. Wallace and Eugene C. Freuder. Anytime algorithms for constraint satisfaction and SAT problems. In *IJCAI-95 Workshop on Anytime Algorithms and Deliberation Scheduling*, pages 99–103, 1995.

[31] Toby Walsh. Depth-bounded discrepancy search. In *Proc. of IJCAI-97*, Nagoya, Japan, 1997.

[32] Shlomo Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3):73–83, 1996.