Proceedings CPAIOR'03

ToOLS : A Library for Partial and Hybrid Search Methods

Simon de Givry[1] **and Laurent Jeannin**[2]
[1]INRA Biometrics and Artificial Intelligence
Chemin de Borde Rouge, BP 27, 31326 Castanet-Tolosan cedex France
degivry@toulouse.inra.fr
[2]Thales Research & Technology
Domaine de Corbeville, 91404 Orsay cedex France
laurent.jeannin@thalesgroup.com

Abstract

We present a library called ToOLS for the design of customized tree search algorithms in a Constraint Programming (CP) framework. We separate the description of search algorithms into three parts: a refinement-based search scheme which defines a complete search tree, a set of conditions for visiting nodes which specifies a parameterized partial exploration, and a temporal strategy for combining several partial explorations. This library allows to express most of the partial, i.e. non systematic backtracking, search methods and also, a specific class of hybrid local/global search methods called Large Neighborhood Search, which is very naturally suited to CP technology. Variants of those methods are easy to implement with the ToOLS primitives. We demonstrate the expressiveness and efficiency of the library by solving a mission management problem which is a mix between a traveling salesman problem with time windows and a knapsack problem. Several partial and hybrid search methods are compared. The best results we get are close to the ones obtained by a dedicated algorithm and drastically outperform CP approaches based on classical depth-first search methods.

Introduction

The purpose of ToOLS [1] (Templates of On-Line Search) is twofold: (i) to help a constraint programmer to build complex customized search algorithms and (ii) to offer ready-made search components for engineers, improving algorithm reuse and capitalization. This paper concerns the first point. ToOLS is part of a finite-domain constraint solver library

[1]This work was partially funded by the RNRT EOLE project [10].

called Eclair [29] developed in the high-level language Claire [2] [7]. Our main contribution is to propose a global approach for the design of partial and hybrid search algorithms. This approach allows to propose a set of search primitives.

Due to the exponential complexity of many combinatorial optimization problems, the size of a complete search tree is often intractable. In practice, in a limited amount of time, a search algorithm explores a subpart of its complete tree only. Depth-first search explores the bottom-left part only. *Partial search methods*, as introduced in [4], explore other parts of the tree, by diversifying their exploration. In most cases, partial search methods provide better results than depth-first search for a given time limit. We distinguish four approaches:

- *Iterative weakening methods* solve the same problem repeatedly with some search restrictions progressively relaxed at each iteration. See, for instances, *iterative broadening* (IB) [11] which uses an artificial breadth cutoff, *limited discrepancy search* (LDS) [17] which uses a maximum number of discrepancies along all the search paths and *depth-bounded discrepancy search* (DDS) [32] which allows discrepancies high in the tree by means of an iteratively increasing depth bound.

- *Real-time heuristic search methods* adapt some cutoff parameters depending on a given time limit. For instance, [9] dynamically adjust the approximation degree of an approximate branch and bound algorithm.

- *Iterative sampling methods* perform a sequence of greedy searches by randomizing their value heuristic [6] or their variable heuristic [14]. In order to improve the quality of the solutions, [14] uses a limited amount of backtrack in every search.

- *Interleaving methods* solve simultaneously different parts of a single search tree, as in *interleaved depth-first search* [25], or different search trees, as in *algorithm portfolios* [13].

Most partial search methods apply conditions for visiting nodes, they change the exploration order and they perform several explorations. A very interesting research direction is the establishment of links between partial search methods and local search methods to eventually hybridize both methods. In particular, a promising hybrid approach is called *large neighborhood search* (LNS) [30]. It consists in a local search method whose neighborhood is explored by a partial search method. Large neighborhoods diminish the risk of being stuck in local minima. The authors of [28] were the first to show the interest of performing a large neighborhood search in a constraint programming framework. The neighborhood space is explored in an efficient way thanks to constraint propagation, cost pruning and partial exploration techniques. LNS applies the branch and bound principle during the neighborhood search. It implies a local descent strategy allowing only better solutions to be found from one neighborhood search to another. A technique called *variable neighborhood search* [15] is used to escape from local minima produced by this descent strategy. The main observation is that partial and hybrid search methods can be characterized by the way they generate search trees and by the way they explore them.

Constraint programming is well known for its declarative nature in problem modeling but it has lacked until recently the same feature for the design of search procedures. First

[2] Claire is copyright of Yves Caseau. ToOLS and Eclair are copyright of Thales.

Localizer [26] was proposed for local search algorithms. It is based on invariants instead of constraints, and does not use constraint propagation. SaLSA [21] was a first attempt to provide a language that unifies global and local search methods extending the concept of choice point. A first implementation was made at Thales. The complexity of the language and its lack of operators for building iterative or LNS methods induced us to restrict the scope of the search functionalities and to focus this scope on a specific class of local/global hybridization. OPL [20] is a major step towards the proposal of a high-level language for global search. The readability of the language, due to its imperative programming approach, which is very important from a software engineering point of view, convinced us to follow the same approach. The expressiveness of OPL is very high but it is a "closed" language (the interface with other general-purpose languages such as C/C++ is at the compiled level) difficult to extend. ToOLS is an object-oriented library part of Eclair . Adding new primitives and new templates (encapsulating parts of search algorithms) is easy. ToOLS and Eclair are written in the Claire [7] programming language which offers most of the programming facilities given by OPL: objects (classes are objects also), efficient set operators, associative arrays (for sparse arrays), garbage collecting and support for backtracking. [27] introduced a unified approach for the design of partial search methods based on a priority queue used to store the current set of open search nodes (as in best-first search). This approach has a potential risk of memory explosion that becomes problematic in case of large scale combinatorial optimization problems. ToOLS keeps the depth-first search principle always. Only the current search path is stored, avoiding any memory problems. For iterative weakening methods, ToOLS will revisit search nodes while [27] will perform state recomputation.

Compared to the previously cited approaches, the main advantages of ToOLS are:

- **Expressiveness**. A unified approach for the design of partial and hybrid search methods based on the notion of partial exploration.
- **Adaptability**. A single search scheme can perform a variety of different searches from a greedy search to a complete search depending on an explicit tuning strategy of cutoff parameters.
- **Readability**. A set of primitives to express complex partial explorations in a declarative way.

The rest of this paper is organized as follows. Section 1 presents the way of designing partial and hybrid search algorithms in ToOLS . Section 2 describes the experiments we made on a benchmark in order to validate our approach.

1 Designing complex tree search algorithms in ToOLS

The novelty of this library is to separate the design of a search algorithm into three distinct components. The first one defines a complete search tree. The second one explores the tree partially. The third one defines a temporal strategy for making several partial explorations. Each component can be reused separately. A search algorithm is a Claire object created by a functional composition of constructors called "ToOLS primitives". This form of nested constructors defines a simple functional language which is easy to parse and interpret. A special function (*solve*, *solveAll* or *minimize*) specifies

the goal of the search (satisfaction or optimization) and is applied to a single algorithm object. The complete syntax is given in the annex.

1.1 Primitives for defining a complete search tree based on refinements

Tree search methods decompose a problem into some simpler problems until a solution is reached. The simplification, called a refinement, consists in reducing the number of solutions by adding some constraints. For this, we use only primitive constraints [5] of Eclair , which have a direct impact on the constraint store maintained by Eclair . For instance, the primitive constraint $x \leq v$ reduces the domain of a variable x to the values lower than v. The primitive constraint $settle(c_1 \, or \, c_2, left)$ replaces in the constraint store the *logical disjunctive constraint* [18] $c_1 \, or \, c_2$ by its left part, i.e. the constraint c_1. In order to get a complete search tree, we restrict the problem decomposition process to a set of predefined complete choice points. The availability of user-defined choice points, like in OPL, would not ensure the property of search completeness. For instance, $splitleq(x, v)$ decomposes a problem into two subproblems: the first one having the constraint $x <= v$ and the other one having the constraint $x > v$. $enum(x)$ enumerates all the values in the current domain of x (with $n = |dom(x)|$). See the following semantic description for the other choice points:

splitleq(x, v)	:	$x <= v \quad \mid \quad x > v$
splitlt(x, v)	:	$x < v \quad \mid \quad x >= v$
setval(x, v)	:	$x == v \quad \mid \quad x! = v$
enum(x)	:	$x == dom(x)[1] \quad \mid \quad x == dom(x)[2] \quad \mid \quad \cdots \quad \mid \quad x == dom(x)[n]$
setdisj$(c_1 \, or \, c_2)$	:	$settle(c_1 \, or \, c_2, left) \quad \mid \quad settle(c_1 \, or \, c_2, right)$

All the choice points are binary choice points, except for *enum* which is nary. In every choice point, a specific heuristic can be used to order choices. Heuristics are Claire functions that can easily access to the constraint store. We combine choice points using an imperative programming approach. For instance, the term $while(x, l, enum(x))$ defines a classical search tree by enumeration. It repeatedly performs the choice point $enum(x)$ (which could be another combination of choice points also) until all the variables in the list l are assigned. Here x is a local variable that is used by the choice point. By default, x is assigned to the first unassigned variable in l before each exploration of $enum(x)$. The correct value of x is restored upon backtracking. The leaves of the tree are the solutions. In optimization, a basic branch and bound method is used, an improvement on the best solution cost found so far is enforced at each node. In real-life applications, classical variable enumeration can be very inefficient. Other search schemes are needed. We show this on two famous examples. The *bridge scheduling problem* [18] is solved by the following search algorithm:

```
do(
   while(d, Disjunctions,
      setdisj(d)),
   while(x, Variables,
      enum(x)))
```

The primitive $do(term_1, term_2)$ connects the subtree $term_2$ at every leaf of the subtree $term_1$. In this example, all the disjunctive constraints are simplified first, then a classical enumeration is performed on the variables. Dealing with disjunctive constraints in an explicit way is a specificity of Eclair , which is useful for scheduling problems, but also for managing complex choice points. It avoids to create new constraints, such as $t_x \geq t_y + d_y$, during the search.

A more complex combinatorial problem, the *perfect square placement problem* [19], is solved by the following search algorithm:

```
do(
   while(x, list{s.xorigin | s in Squares}, smallestVar,
      let(xinf, delay(inf, x),
         splitleq(x, xinf))),
   while(y, list{s.yorigin | s in Squares}, smallestVar,
      let(yinf, delay(inf, y),
         splitleq(y, yinf))))
```

$smallestVar$ is a heuristic that returns the first unassigned variable with the smallest value in its domain. The *let* primitive defines a local variable and computes its value once only, before entering into the subtree (defined by *splitleq* in this example). The *delay* primitive is used to perform a function call with optional parameters during the search. Here local variable $xinf$ is equal to the smallest value in the current domain of x, produced by calling the function inf applied to x. Remember that a ToOLS term is an object that will be interpreted during the search. Any call to a Claire function has to be encapsulated into an object, this is what the primitive *delay* does. The primitive *while* interacts with the constraint store in an elegant way: the number of iterations depends on the list of variables and the propagations. At the leaves of the first *while*, all the x-axis coordinates of the packed squares are assigned. The second *while* assigns all the y-axis coordinates. A complete search using this search algorithm finds a first solution in a second and ends in one minute on a modern computer, finding 8 symmetrical solutions for the 21-square problem. The same search algorithm written in Claire takes about the same time. The ToOLS terms, which are interpreted during the search, induce a small overhead. The implementation of ToOLS relies on polymorphic mechanisms: every search primitive has an associated class constructor and an associated search function. All the search functions have the same interface: a global context (to store the best solution and some printing information), a list of pending search terms (used by the primitive *do*) where the search will continue after having explored the leaves of the current search term, and a list of active search limits (see section 1.2). The interpretor overhead is mainly due to the management of these lists, the polymorphic function calls and the indirection toward Claire functions and variables (primitive *delay*). However in many applications, the time taken by the search part is negligible compared to the time taken by constraint propagation.

1.2 Primitives for partial exploration

We define some primitives to control the size of the explored part of a given search tree. The primitives specify some conditions for visiting nodes. A condition is a formula, $expression \leq threshold$, that must hold at any node (before posting a primitive

constraint). *expression* defines a way to evaluate a node (primitive **nodelimit**), a path (primitive **pathlimit**), or a subtree (primitives **treelimit** and **globallimit**) during the search. When a search algorithm exceeds the threshold of a *treelimit*, the exploration of the subtree is stopped and the search backtracks towards the last choice point outside the subtree. In case of a *globallimit*, the search ends definitively (*globallimit* has the same effect as a conditional breakpoint). We found the following useful expressions:

- **order**: rank of a node (*nodelimit*). The alternative choices (child nodes) of a choice point (parent node) are sorted according to a given heuristic, and each alternative is assigned a rank based on this sort. The first choice starts at rank zero.
- **distance**: distance of a node to the preferred node (*nodelimit*). A special heuristic is used to mark each alternative of a choice point. All the alternatives are evaluated by the heuristic before a choice is made. The preferred alternative corresponds to the one having the highest mark value, and it has a distance equal to zero. See figure 1.
- **sum(order)**: sum of all the node ranks in a search path (*pathlimit*)
- **sum(distance)**: sum of all the node distances in a search path (*pathlimit*)
- **nbbacktracks**: number of backtracks in a subtree (*treelimit* and *globallimit*)
- **nbnodes**: number of nodes in a subtree (*treelimit* and *globallimit*)
- **nbleaves**: number of leaves in a subtree (*treelimit* and *globallimit*)

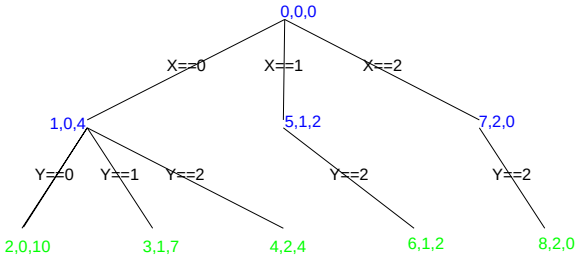

Figure 1: A search tree with 5 leaves/solutions, 4 backtracks and 8 nodes corresponding to the maximization of $2X + 3Y$ where the variables X and Y belong to $[0, 2]$, using the following search algorithm: $do(enum(X, increasing, mark_X), enum(Y, increasing, mark_Y))$. At each node, the exploration order, the evaluation of $sum(order)$ and the evaluation of $sum(distance)$ are given. The domain values of X and Y are explored in the initial order, expressed by the heuristic *increasing*. Let $mark_X(v) \mapsto 2v$ and $mark_Y(v) \mapsto 3v$ be two heuristic functions that return a mark for any assignment of X and Y. For instance, the third explored node corresponds to the assignment $X == 0$ and $Y == 1$. Here, $sum(order)$ evaluates to $0 + 1 = 1$ and $sum(distance)$ evaluates to $(2 * 2 - 2 * 0) + (3 * 2 - 3 * 1) = 7$.

Every condition applies to a given subtree. Let *st* be any subtree produced by a ToOLS term as defined in the previous section. Then $nodelimit(0, order, st)$ implements

a greedy search. Each visited node correspond to the first alternative choice, and has a rank equal to zero. The search algorithm $pathlimit(1, sum(order), st)$ explores all the paths with zero or one discrepancy. Note that best alternative choices and second best alternatives are visited only, other alternatives having a discrepancy greater than one.

Let $while_1$ and $while_2$ correspond to both *while* terms in the perfect square example. Then,

```
do(
   pathlimit(3, sum(order), while_1),
   pathlimit(1, sum(order), while_2))
```

applies two different discrepancy limits for each subtree.

$treelimit(100, nbnodes, pathlimit(1, sum(order), st))$ explores at most 100 nodes of a partial search tree. The order of condition primitives applied on the same subtree does not matter. When several limits occur at the same time, the search backtracks to the closest choice point from the root. In case of *nodelimit* and *pathlimit*, the scope of these conditions can be restricted by an additional argument (**relDepth**), specifying that the conditions are only active inside a depth interval of a given subtree. The bounds of this interval are tunable thresholds. A positive bound means the depth is relative to the root of the subtree. A negative bound means the depth is relative to the current deepest leave. For instance $nodelimit(0, order, relDepth(3, -1), st)$ will explore the first two choice points completely, the best alternative of the other choice points is visited only, except for the last choice point which is completely explored. Negative depth bounds implement *Bounded Backtrack Search* (BBS) [16, 32]. The **distance** keyword is used when "it is not the number of discrepancies that matter, but rather the quality of the discrepancies" [3]. A *mark heuristic* is a function that returns a *signal* representing the value of expanding a node. For instance, it can be the expected cost value in optimization. In the example of figure 1, we project the linear objective function $2X + 3Y$ on each variable X and Y. [3] shows that "signal strength plays a more significant role than discrepancies in determining where the search effort should be spent". Further investigations need to be done in this direction.

In condition formulae, **threshold** is a cutoff value that tunes the degree of incompleteness of the exploration. Decreasing values imply decreasing sizes of the explored part of a given search tree. We call the cutoff values the incompleteness parameters. These parameters are clearly exhibited in ToOLS . A static value for these parameters can be used as in the previous examples. But a more general approach consists in defining a ***tuning policy***. A tuning policy restricts the space of the possible combinations of parameter values to the "relevant" combinations and sorts these combinations by an order of increasing complexity[3]. The first combination should correspond to a greedy search and the last one to a complete search. A well-known tuning strategy consists in performing a sequence of partial explorations based on the same search tree following the ordered tuning policy from the first greedy combination to the last complete one. The primitive **increasedScope** implements this strategy. We describe several iterative weakening methods using this primitive:

- **IB**: *iterative broadening* [11]

[3] In practice, the increasing property is verified if the policy contains monotonically increasing parameter values.

$increasedScope(p, list(0, 1, 2, \ldots), nodelimit(p, order, st))$

- **LDS**: *limited discrepancy search* [17]
  $increasedScope(p, list(0, 1, 2, \ldots), pathlimit(p, sum(order), st))$
- **LDS-BBSk**: *LDS & bounded backtrack search* [16]
  $increasedScope(p, list(0, 1, 2, \ldots), pathlimit(p, sum(order), relDepth(1, -k), st))$
- **DDS**: *depth-bounded discrepancy search* [32]
  $increasedScope(p, list(1, 2, \ldots), nodelimit(0, order, relDepth(p, \infty), st))$[4]
- **DDS-BBSk**: *DDS & bounded backtrack search*[32]
  $increasedScope(p, list(1, 2, \ldots), nodelimit(0, order, relDepth(p, -k), st))$
- **DBDFSk**: *discrepancy-bounded depth first search* [2]
  $increasedScope(p, list(k-1, 2k-1, 3k-1, \ldots), pathlimit(p, sum(order), st))$

The relevant combinations are easy to be found when there is a single integer parameter. This is not the case with float parameters, when mark heuristics are normalized, or combinations of parameters. In that case, 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 tree. 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 relevant combinations by doing experiments. [3] learns the *optimal cutoff policy* for a single float parameter of the *weighted discrepancy search* method from a model of the value heuristic. When the time limit is known, one should take maximum advantage of this information. This is the purpose of real-time heuristic search methods that use a dynamic tuning strategy [12].

1.3 Primitives for combining several partial explorations

The temporal combination is described by two common primitives which are the basis of several algorithms given as examples:

- **sequence**$(hs_1, hs_2, \ldots)$: a sequence of several search algorithms (hs_i can be any ToOLS term). hs_{i+1} is performed when the search hs_i is finished[5]. Examples are iterative sampling methods [6, 14] and large neighborhood search methods [30, 8, 24].
- **interleave**$(hs_1, hs_2, \ldots)$: an interleaving of several search algorithms. All the hs_i are performed at the same time. Examples are *interleaved depth-first search* [25] and *algorithm portfolios* [13] .

All the searches are completely independent (interleaved searches use distinct copies of the constraint store), except for the solutions which are stored in a common pool and the best cost bound in optimization which is shared. The search process stops before

[4] Without the improvement which consists in not re-visiting any nodes at the depth bound p.

[5] Note the difference with the primitive $do(s_1, s_2, \ldots)$ of section 1.1, where s_{i+1} is explored at every leave of s_i.

the end of all its searches if a solution is found in satisfaction or the optimum has been proved in optimization (a partial search is complete if no threshold has been overcome during its search). A simple *sequence* example is to perform a greedy search, a partial search and a complete search sequentially, each search using a different search scheme. Iterative weakening methods follow this approach but use only **one** scheme of problem decomposition based on the **same** heuristics. The *sequence* primitive overcomes this limitation. ToOLS lets the user define its own *generator* function in Claire that will generate a sequence of partial search algorithms dynamically. This feature is used to implement large neighborhood search methods, see section 2.2 for a concrete example. In addition, we can specify how to distribute a global time limit to all the searches by the notion of a *time-sharing policy* [12].

2 Experiments

2.1 A mission management benchmark for agile satellites

The benchmark[6] consists in solving a simplified version of a problem of selecting and scheduling observations for agile satellites. See [31, 23] for a complete description. The satellite has a pool of candidate photographs to take, and must select and schedule a subset of them, at each pass above a strip of the earth territory. The satellite can only take one photograph at a time (disjunctive scheduling). A photograph can only be taken during a given time window, depending on its coordinates on the earth surface. Minimal time manoeuvers are required between two consecutive photographs. All physical constraints (time windows and time manoeuvres) must be met, and the sum of the revenues of the selected photographs must be maximized (linear objective criterion). This problem is a mix between a *Traveling Salesman Problem* with time windows, and a *Knapsack problem*. The decision variables are grouped in two sets: the first one for the *selection* (binary variables) and the second one for the acquisition *starting times* of the selected photographs. The refinement-based search scheme we used corresponds to a general search procedure for scheduling problems as described in [1]:

```
ST :: while(t, StartTimes, mostUrgent,
         let(tinf, delay(inf, t),
            do( splitleq(t, tinf, bestChoice),
                if(delay(?>?, t, tinf),
                   tell(t, >=, delay(postponeRule, StartTimes, t))))))
```

The *while* loop iterates until all the photographs have their start times fixed (selected photographs) or postponed outside their time windows (rejected photographs). Let $StartTimes$ be the list of start time variables t_i sorted by their associated revenue (highest first). The Claire function $mostUrgent$ returns the first unassigned start time variable with the smallest value in its domain which is compatible with the time window constraint, or returns a special value indicating the end of the *while* loop. The *splitleq* choice point implements a schedule or postpone strategy. We use the value heuristic proposed in [31] to decide whether the start time is fixed to its minimum value or postponed first. This heuristic approximates the future gain by making the first alternative compared to the one by selecting and fixing another available photograph. The available photographs

[6] This benchmark is available in the free constraint solver *choco* [22].

are restricted to the ones whose starting time is reduced if the first alternative is chosen. Last a redundant constraint is added, using the primitive *tell*, when a start time t_i is postponed (test performed by the primitive if, with $? >?(t, tinf) = inf(t) > tinf$):

$$t_i \geq \min_{j \neq i}(inf(t_j) + D_j + M_{j,i})$$

This redundant constraint, called a delaying constraint in [1], avoids the search algorithm to enumerate the possible start time values.

We solved to optimality problem instances with less than thirty candidate photographs. For larger instances, we developed a specific hybrid search algorithm.

2.2 Design of a hybrid search algorithm

The algorithm implements a large neighborhood search method [30]. Its main core is a local descent search method. Limited Discrepancy Search [17] with Bounded Backtrack Search [16, 32] is applied to perform an efficient partial exploration of a large neighborhood in a constraint programming framework. The neighborhood is built by keeping a subset of the selected photographs in the best solution found so far (selection variables are assigned to one) and by enforcing the previous sequencing order of the selected photographs (disjunctive scheduling constraints are *settled*). We use a *Variable Neighborhood Descent* (VND) method [15] to escape from local minima. The method starts with rejecting only one photograph. And a partial search is done for all the neighborhoods with one photograph rejected. If no improving solution is found, the number of rejected photographs is increased by one, and so on. If a better solution is found, this number is reset to one. The rejected photographs are chosen in a deterministic way using a sliding window of consecutive photographs. An analog search strategy is used in *VNS/LDS+CP* [24]. The corresponding ToOLS code is:

```
VND/LDSk-BBSl :: sequence(
                    globallimit(1, nbleaves, ST),
                    sequence(delay(GenerateNeighborhood,
                                   pathlimit(k,
                                             sum(order),
                                             relDepth(1, -l),
                                             ST))))

GenerateNeighborhood(Choice st)
 do( let(list(selectedPhotoSelections, selectedDisjunctions), delay(vndStrategy),
        do( forall(x, selectedPhotoSelections,
               tell(x, ==, delay(getLastSolution, x))),
            forall(d, selectedDisjunctions,
               tell(d, delay(getLastOrder, d))))),
    st)
```

2.3 A comparative analysis of partial and hybrid search algorithms

We generated several random instances of the above described problem for different numbers of candidate photographs. We compared different versions of partial search methods

as described in sections 1.2 and 1.3, and our hybrid search algorithm. All the algorithms are based on the same refinement-based search scheme ST using the same heuristics (except for the iterative sampling methods). We also implemented the sequence-based greedy algorithm (*Greedy*) and the dynamic programming algorithm (DPA) described in [31, 23]. DPA provides the best (non optimal) results and is very fast (less than a second for 200 photographs). But it corresponds to a very specialized algorithm which cannot cope with new constraints. In fact the benchmark corresponds to the simpliest problem defined in [31] (with a linear criterion) and the dynamic programming approach is not applicable to the more complex problem (with non linear criterion and stereoscopic constraints). The constraint programming approach is applicable to both versions. Our goal is to assess and compare the quality of the results obtained by search methods which use constraint programming. DPA results are used as reference values.

The results are presented in table 1.

Problem size	50		100		200	
Cpu time	30 sec.		1 min.		5 min.	
	Mean	%	Mean	%	Mean	%
Greedy	**3600**	0 %	**4205**	0 %	**4732**	0 %
DFBB	**3718**	51 %	**4271**	28 %	**4788**	20 %
DDS	3609	3 %	4211	2 %	4733	0 %
DDS-BBS1	3709	47 %	4280	31 %	4784	19 %
DDS-BBS2	3711	48 %	4283	33 %	4785	19 %
DDS-BBS4	**3715**	50 %	**4290**	36 %	**4789**	21 %
LDS	3772	74 %	4331	53 %	4837	39 %
LDS-BBS1	3774	75 %	**4333**	54 %	4840	40 %
LDS-BBS2	**3774**	75 %	4332	54 %	**4841**	40 %
LDS-BBS4	3772	74 %	4330	53 %	4838	39 %
DBDFS2	3766	72 %	4321	49 %	4826	35 %
DBDFS4	3765	71 %	4314	46 %	4815	30 %
ISamp	3733	57 %	4339	57 %	4860	47 %
ISamp/BBS4	3746	63 %	4344	59 %	4865	49 %
ISamp/BBS8	3753	66 %	4347	60 %	**4866**	50 %
ISamp/BBS16	3760	69 %	4345	59 %	4859	47 %
ISamp/LDS1-BBS1	**3779**	77 %	**4353**	62 %	4858	47 %
VND/LDS1-BBS1	3813	92 %	4396	81 %	4929	73 %
VND/LDS2-BBS2	**3815**	93 %	**4399**	82 %	4929	73 %
VND/LDS4-BBS4	3811	91 %	4393	80 %	4923	71 %
VND/LDS8-BBS8	3809	90 %	4395	80 %	4916	68 %
VND/DFBB	3809	90 %	4394	80 %	4911	66 %
VND/DFBB1000	3812	92 %	4396	81 %	4913	67 %
VND/DFBB100	3806	89 %	4398	82 %	**4929**	73 %
DPA	**3830**	100 %	**4440**	100 %	**5000**	100 %

Table 1: A comparative analysis of partial and hybrid search algorithms. $Mean$ is the mean value of the best solution found after a given cpu time for 100 randomly-generated instances. The percentage is equal to $100 \times \frac{mean(Algorithm)-mean(Greedy)}{mean(DPA)-mean(Greedy)}$.

The analysis of experimentation results shows:

- The hybrid search algorithms obtain the best CP results, close to the DPA results. We compared different partial search methods for the neighborhood search. A complete neighborhood search ($VND/DFBB$) saturates when the size of the problem increases. LDS provides us slightly better results than using a restricted number of backtracks ($VND/DFBB1000$ and $VND/DFBB100$).

- The iterative sampling approach ($ISamp$) reaches the second position. The difficulty is how to add randomness. We replaced the value heuristic by a biased one which performs a random choice if the current selected photograph is close to the best unselected available photograph. We tried different partial exploration methods instead of the greedy search used by $ISamp$. Surprisingly, $LDS1 - BBS1$ provides us the best combination for 50 and 100 photographs. This shows the interest of building customized search algorithms.

- The iterative weakening approach comes in the third position. LDS outperforms DDS. Our explanation is that the quality of the value heuristic does not depend on the search depth. Adding a limited amount of backtrack at the leaves improves the results for LDS.

[31] gave the results obtained by a constraint programming algorithm (CPA) using classical depth-first search. For two problems with 106 and 147 photographs respectively, they report a relative distance from CPA to DPA ($100\times\frac{(DPA-CPA)}{DPA}$) of 26.7% and 13.3%. We drastically reduce the mean relative distance ($100\times\frac{(mean(DPA)-mean(VND/LDS2-BBS2))}{mean(DPA)}$) to respectively 0.9% and 1.4% for 100 and 200 photographs by using a hybrid search method.

Conclusion

Complex tree search algorithms are separated into three parts: the definition of a complete search tree, a set of conditions for visiting nodes and a temporal strategy for combining several partial explorations. In a few number of lines of code, ToOLS allows to experiment a large variety of partial and hybrid search methods. An interesting result is the good performance of a large neighborhood search method based on limited discrepancy search and bounded backtrack search. ToOLS makes the design of such new mixtures easier. Recently, a more complex version of the satellite benchmark was used for a challenge in the French Operations Research community. We obtained competitive results by reusing the same large neighborhood search algorithm. The main conclusion is that partial search methods integrate easily in a constraint programming framework and are the basis of powerful hybrid search methods.

Other experiments on a military application showed that partial search methods significantly improve the solution quality compared to an existing customized greedy algorithm and also demonstrated the gain in development time of new customized search algorithms. The code is clearer and more concise when using the search primitives. Efficiency issues were taken into account during all the design process and implementation of ToOLS (search limits are computed incrementally, the code to interpret predefined choice points is optimized, etc.). The whole framework, Claire + Eclair + ToOLS , has

been integrated with success in an operational on-board hard real-time system of Thales.

We thank Michel Lemaître for providing the benchmark on agile satellites. Thanks also to the anonymous reviewers for their useful comments.

References

[1] P. Baptiste and C. Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1-2):119–139, 2000.

[2] J.C. Beck and L. Perron. Discrepancy-bounded depth first search. In *Proc. of CP-AI-OR'2000*, Paderborn, Germany, March 8-10 2000.

[3] T. Bedrax-Weiss. *Optimal Search Protocols*. PhD thesis, University of Oregon, 1999.

[4] N. Beldiceanu, E. Bourreau, P. Chan, and D. Rivreau. Partial search strategy in chip. In *Proc. of 2nd Int. Conf. on Meta-Heuristics*, Sophia-Antipolis, France, 1997.

[5] A. Bockmayr and T. Kasper. Branch-and-infer: A unifying framework for integer and finite domain constraint programming. *INFORMS J. Computing*, 10(3):287 – 300, 1998.

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

[7] Y. Caseau, F.X. Josset, and F. Laburthe. Claire: Combining sets, search and rules to better express algorithms. In *Proc. of ICLP'99*, pages 245–259, Las Cruces, New Mexico, 1999.

[8] Y. Caseau and F. Laburthe. Effective forget-and-extend heuristics for scheduling problems. In *Proc. of CP-AI-OR'1999*, Ferrara, Italy, February 1999.

[9] Lon-Chan Chu and Benjamin W. Wah. Optimization in real time. In *Proc. of the Twelfth Real Time Systems Symposium*, pages 150–159, Washington, D.C., 1991.

[10] EOLE consortium. EOLE project: On-line optimization framework for telecom. http://www.lcr.thomson-csf.com/projects/www_eole (in french), 2000.

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

[12] S. de Givry, Y. Hamadi, J. Mattioli, P. Gérard, M. Lemaître, G. Verfaillie, A. Aggoun, I. Gouachi, T. Benoist, E. Bourreau, F. Laburthe, P. David, S. Loudni, and S. Bourgault. Towards an on-line optimisation framework. In *CP-2001 Workshop on On-Line combinatorial problem solving and Constraint Programming (OLCP'01)*, pages 45–61, Paphos, Cyprus, December 1 2001.

[13] C. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[14] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of AAAI-98*, Madison, WI, 1998.

[15] P. Hansen and N. Mladenovic. *Handbook of Metaheuristics*, chapter Variable Neighborhood Search. Kluwer Academic Publisher, f. glover and g. kochenberger edition, 2002.

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

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

[18] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, 1989.

[19] P. Van Hentenryck. *Intelligent Scheduling*, chapter Scheduling and Packing in the Constraint Language cc(FD). Morgan Kaufmann, 1994.

[20] P. Van Hentenryck. *OPL: The Optimization Programming Language*. The MIT Press, Cambridge, Mass., 1999.

[21] F. Laburthe and Y. Caseau. Salsa: A language for search algorithms. *Constraints*, 7(3):255–288, 2002.

[22] F. Laburthe and the OCRE project team. Choco: implementing a cp kernel. In *CP-2000 Workshop on Techniques for Implementing Constraint Programming Systems (TRICS)*, Singapore, 2000. http://www.choco-constraints.net/.

[23] M. Lemaître, G. Verfaillie, F. Jouhaud, J-M. Lachiver, and N. Bataille. Selecting and scheduling observations of agile satellites. *Aerospace Sciences and Technology*, 6:367–381, 2002.

[24] S. Loudni and P. Boizumault. Vns/lds+cp: A hybrid method for constraint optimization in anytime contexts. In *Proc. of 4th Metaheuristics International Conference*, pages 761–765, Porto, Portugal, 2001.

[25] P. Meseguer. Interleaved depth-first search. In *Proc. of IJCAI-97*, pages 1382–1387, Nagoya, Japan, 1997.

[26] L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5(1-2):43–84, 2000.

[27] L. Perron. Search procedures and parallelism in constraint programming. In *Proc. of CP-99*, pages 346–360, Alexandria, Virginia, October 11-14 1999.

[28] G. Pesant and M. Gendreau. A constraint programming framework for local search methods. *Journal of Heuristics*, 5:255–279, 1999.

[29] PLATON Team. *Eclair reference manual*. PLATON, THALES Research & Technology, Orsay, France, version 6.0 edition, 2001.

[30] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proc. of CP-98*, pages 417–431, Pisa, Italy, 1998.

[31] G. Verfaillie and M. Lemaître. Selecting and scheduling observations for agile satellites: some lessons from the constraint reasoning community point of view. In *Proc. of CP-01*, Paphos, Cyprus, 2001.

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

ToOLS syntax chart

```
<Run> ->  solve( <HybridSearch> [, time] )
      ->  solveAll( <HybridSearch> [, time] )
      ->  minimize( variable, <HybridSearch> [, time] )
      ->  maximize( variable, <HybridSearch> [, time] )

<HybridSearch> ->  sequence( [timeSharingPolicy,] {<HybridSearch>}+ )
               ->  sequence( [timeSharingPolicy,] generator )
               ->  interleave( [timeSharingPolicy,] {<HybridSearch>}+ )
               ->  interleave( [timeSharingPolicy,] generator )
               -> <PartialSearch>

<PartialSearch> ->  increasedScope( thresholds, tuningPolicy, <Choice> )
                ->  decreasedScope( thresholds, tuningPolicy, <Choice> )
                ->  fixedScope( thresholds, tuningPolicy[i], <Choice> )
                -> <Choice>

<Choice> ->  do( {<Choice>}+ )
         ->  while( variable, <Choice> )
         ->  while( identifier, variables [, heuristic], <Choice> )
         ->  while( identifiers, tuplesOfVariables [, heuristic], <Choice> )
         ->  while( identifier, disjunctions [, heuristic], <Choice> )
         ->  case( identifier, expression, {setOfAny, <Choice>}+ [, <Choice>] )
         ->  if( expression, <Choice> [, <Choice>] )
         ->  let( identifier, expression, <Choice> )
         ->  splitleq( variable, integer [, heuristic [, markheuristic]] )
         ->  splitlt( variable, integer [, heuristic [, markheuristic]] )
         ->  setval( variable, integer [, heuristic [, markheuristic]] )
         ->  enum( variable [, heuristic [, markheuristic]] )
         ->  setdisj( disjunction [, heuristic [, markheuristic]] )
         ->  tell( variable, {<= | < | >= | > | == | !=}, integer )
         ->  tell( disjunction, { left |  right } )
         -> <Limit>

<Limit> ->  nodelimit( threshold, { order |  distance }, [<Scope>,] <Choice> )
        ->  pathlimit( threshold,  sum( order [, weights]), [<Scope>,] <Choice> )
        ->  pathlimit( threshold,  sum( distance [, weights]), [<Scope>,] <Choice> )
        ->  treelimit( threshold, { nbbacktracks |  nbnodes |  nbleaves }, <Choice> )
        ->  globallimit( threshold, { nbbacktracks |  nbnodes |  nbleaves }, <Choice> )

<Scope> ->  relDepth( threshold, threshold )
```