A unified framework for partial and hybrid search methods in constraint programming

Simon de Givry, Laurent Jeannin

▶ To cite this version:

Simon de Givry, Laurent Jeannin. A unified framework for partial and hybrid search methods in constraint programming. Computers and Operations Research, 2006, 33 (10), pp.2805-2833. 10.1016/j.cor.2005.01.012 . hal-02659665

HAL Id: hal-02659665
https://hal.inrae.fr/hal-02659665
Submitted on 30 May 2020

ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT

Computers & Operations Research 33 (2006) 2805–2833

computers & operations research

www.elsevier.com/locate/cor

A unified framework for partial and hybrid search methods in constraint programming

Simon de Givry[a,*], Laurent Jeannin[b]

[a]*INRA Unité de Biométrie et Intelligence Artificielle, Chemin de Borde Rouge, BP 27, 31326 Castanet-Tolosan Cedex, France*
[b]*Thales Research & Technology, Domaine de Corbeville, 91404 Orsay Cedex, France*

Available online 19 February 2005

Abstract

We present a library called ToOLS for the design of complex tree search algorithms in constraint programming (CP). We separate the description of a search algorithm into three parts: a refinement-based search scheme that defines a complete search tree, a set of conditions for visiting nodes that specifies a parameterized partial exploration, and a strategy for combining several partial explorations. This library allows the expression of most of the partial, i.e. nonsystematic backtracking, search methods, and also a specific class of hybrid local/global search methods called large neighborhood search, which are very naturally suited to CP. Variants of these methods are easy to implement with the ToOLS primitives. We demonstrate the expressiveness and efficiency of the library by solving a satellite mission management benchmark that is a mix between a traveling salesman problem with time windows and a Knapsack problem. Several partial and hybrid search methods are compared. Our results dramatically outperform CP approaches based on classical depth-first search methods.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Design; Languages; Constraint programming; Combinatorial optimization; Search strategies; Tree search; Large neighborhood search

1. Introduction

Constraint programming (CP) is a declarative language that allows combinatorial problems to be modeled and solved. It is an open paradigm that is well suited for the integration of techniques from

* Corresponding author.
E-mail addresses: simon.degivry@toulouse.inra.fr (S. de Givry), laurent.jeannin@thalesgroup.com (L. Jeannin)
URL: http://www.inra.fr/bia/T/degivry/ (S. de Givry).

Artificial Intelligence and Operations Research. CP solvers provide efficient algorithms through the use of global constraints. A constraint model has modularity properties, i.e. adding/removing a constraint is easy, which enables an incremental development process, reducing the development time and effort. The declarative nature of CP enables the programmer to focus on the application requirements rather than on debugging low-level programming errors. Validated CP models can be reused in a product line approach. The technology has spread to many markets such as manufacturing, transportation, telecommunications and building. Hundreds of industrial applications based on CP are used on a daily basis all over the world. Many of them use robust off-the-shelf CP solvers, including ILOG Solver [1] and CHIP [2].

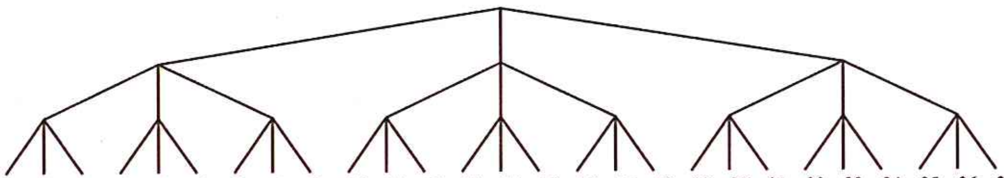
CP was specifically designed for tree search methods, in particular depth-first search (DFS), which is best suited to incremental constraint propagation. Many combinatorial optimization problems are NP-hard and therefore intractable due to the size of a complete search tree. Given a limited amount of time, a tree search algorithm explores a subpart of its complete tree only. DFS explores the bottom-left part only. Partial search methods, as introduced in [3], explore other parts of the tree, by diversifying their exploration. In most cases, partial search methods provide better results than DFS for a given time limit. 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, the large neighborhood search (LNS) [4] is a promising hybrid approach. LNS consists in a local search method whose neighborhood is explored by a complete or a partial search method. Large neighborhoods diminish the risk of being stuck in a local optimum.

CP is well known for its declarative nature in problem modeling but, until recently, it has lacked the same feature for the design of partial and hybrid search methods. Localizer [5] for local search methods, OPL [6] for tree search methods and Salsa [7] for both local and tree search methods were major attempts to define high-level languages for the search. However, none of these languages offers a unified framework for the design of partial and hybrid search methods in CP. As far as we know, ToOLS (Templates of On Line Search[1]) is the first concrete proposition in this direction. ToOLS divides the description of a search algorithm into three parts: a complete search tree defined by a refinement-based search scheme, a set of conditions restricting the exploration of the tree, and a combination of several partial explorations. This approach allows a set of search *primitives* to be proposed. ToOLS has been implemented and integrated on top of Eclair [9], the Thales operational finite domain constraint solver based on the Claire [10] language. 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, also used to explore neighborhoods in hybrid search.
- *Adaptability.* A single search scheme can be used to perform a variety of different searches, from a greedy search to a complete search, depending on an explicit tuning strategy of dynamically adjusted cutoff parameters. With these parameters, it is possible to implement automatic tuning strategies that are not discussed in this paper (for that, see [11–13]).
- *Readability and modularity.* A set of primitives to express complex partial explorations in a declarative and modular way.

The rest of this paper is organized as follows. Section 2 gives an overview of existing partial and hybrid search methods that will be used in subsequent sections. Section 3 shows how to design partial and hybrid

[1] ToOLS was initially designed for hard and soft real-time applications [8]. Templates of search algorithms provide ready-made search components for engineers, improving algorithm reuse and capitalization.



DFS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Discrepancy	0	1	2	1	2	3	2	3	4	1	2	3	2	3	4	3	4	5	2	3	4	3	4	5	4	5	6
IB	1 2 14	3 15	16	4 17	5 18	19	20	21	22	6 23	7 24	25	8 26	9 27	28	29	30	31	10 32	11 33	34	12 35	13 36	37	38	39	40
LDS	1 2 6	3 7	8	4 9	10		11			5 12	13		14						15								
DBDFS2	1	2	5	3	6	7	8	9	18	4	10	11	12	13	19	14	20	21	15	16	22	17	23	24	25	26	
DDS	1			4			5			2			6		3	7						8			9		

Fig. 1. Visiting orders of leaf nodes for various iterative weakening methods.

search algorithms in ToOLS. Section 4 describes the experiments we made on a mission management benchmark for agile satellites in order to show the expressiveness and readability of our framework, and to give a comparison of various partial and hybrid search algorithms.

2. Background

2.1. Partial search methods

The main idea is to diversify the search by avoiding the thrashing phenomenon of systematic backtracking methods. Systematic backtracking methods can spend a very long time to explore a subtree containing no feasible solution or only suboptimal solutions. See [14,15] for an analysis of this phenomenon. We give a classification of the existing partial search algorithms:

- *Iterative weakening methods* solve the same problem repeatedly with some search restrictions progressively relaxed at each iteration. The successive searches are of increasing complexity, until optimality is proved or the deadline is reached. For instance, iterative broadening (IB) [16] uses an artificial breadth cutoff, with a restriction on the number of explored values in each domain. Limited discrepancy search (LDS) [17] uses a maximum number of cumulated discrepancies along all the search paths. Depth-bounded discrepancy search (DDS) [18] allows discrepancies high in the tree by means of an iteratively increasing depth bound. Discrepancy-bounded depth first search (DBDFS) [19] uses a minimum and maximum number of discrepancies along all the search paths. Fig. 1 shows the behavior of several iterative weakening methods, in particular the way they change the exploration order and perform multiple explorations.
  The main drawback of these methods is that each new iteration revisits all the nodes of the previous iteration, except when a tighter upper bound has been found in minimization or, for some iterative

methods, they revisit neither leaf nodes [18], nor interior nodes [19].[2] The semantic decomposition method in [20] revisits subproblems rather than search nodes. The ability to find decreasing upper bounds quickly during the first iterations helps the Depth First Branch and Bound algorithm to cut branches earlier in subsequent iterations. This fact alone justifies iterative methods.

- *Real-time heuristic search methods* adapt some cutoff parameters depending on a given time limit. For instance, Ref. [21] dynamically adjusts the approximation degree of an approximate branch and bound algorithm. In best first search, Ref. [22] dynamically adjusts the depth of a look-ahead search. The self-adjusting depth first branch and bound algorithm in [11,12] dynamically tunes two branch factor thresholds in order to end the search near a given deadline.
- *Iterative sampling methods* try several different value and variable ordering heuristics rapidly by doing greedy searches or very incomplete searches. One way to obtain new heuristics is to bias a given heuristic randomly (see [15] for biased variable ordering heuristics and [23] for biased value ordering heuristics). The main drawback of these methods is the difficulty of improving the solution quality when a large amount of time is allocated. This is due to a large degree of incompleteness and also to blind searches in case of a random selection. A solution proposed by [15] is to increase the search effort every n searches.
- *Interleaving methods* simultaneously examine different parts of a single search tree, as in interleaved depth-first search [24], or different search trees, as in Algorithm Portfolios [25]. The interleaving methods should be used when there are different search algorithms or search heuristics that perform well on different instances of a given problem. When interleaving several searches, the most promising ones can get more computational resources as time passes. When solving constraint satisfaction problems, Ref.[26] choose to allocate more CPU time to the search that reaches the deepest node in the search tree. When solving optimization problem, [27] applies a reinforcement learning strategy to focus on the best algorithms in an algorithm portfolios approach.

2.2. *Hybrid search methods and large neighborhood search*

The exploration of local search neighborhoods using CP was initially proposed by Pesant and Gendreau [28]. They transformed the k-interchange neighborhood for the travelling salesman problem into a CP model with k variables specifying the removed edges and a set of interface constraints linking these k variables with the variables of the original problem. Constraint propagation and cost pruning were able to discard infeasible or uninteresting sets of neighbors, accelerating the neighborhood exploration relative to classic neighbor enumeration for $k \geqslant 4$ and a number of cities greater than sixty. The speed-up can be even greater if side-constraints are added to the original TSP.

A common neighborhood is obtained by freezing a part of the current solution and by relaxing the rest. This defines a new subproblem in which a better solution is sought, by using tree search. In n-jobs$\times m$-machines job-shop scheduling, [29] kept the job sequences on all machines but one. The scheduling of n tasks on the one remaining machine defined the neighborhood, which was completely explored by branch and bound. This method is called *shuffle*. In vehicle routing, Shaw [4] relaxed a set of *related* visits to be reinserted at a different place in the current solution. Related visits were geographically close

[2] The ones which conduct to a search tree with every node having a number of cumulated discrepancies strictly lower than the current iteration limit.

to one another or from the same route. Note that the selection of the related visits was not included in the neighborhood definition but were randomly chosen at each local move. The neighborhood subproblem was created by fixing some variables of the original problem only (the frozen visits), without adding new variables and constraints as in [28]. The neighborhood can be very large (up to 30% of the visits were relaxed in [4]), hence the term large neighborhood search. Partial search has been used to guide the search towards good neighbors rapidly. Different partial search methods have been incorporated into LNS by several authors [4,27,30–33]. For instance, LDS with a constant discrepancy limit was used in [4,32]. For a network design problem, Perron [27] showed that the addition of a constant discrepancy limit and a limit on the number of backtracks provided results that were more robust than using just one of these two search limits.

All the previously cited methods perform a local descent strategy, permitting only better solutions to be found from one neighborhood search to another. A simple strategy to escape from local optima is to use variable neighborhoods. This is the variable neighborhood search (VNS) principle introduced by Mladenovic and Hansen [34], who proposed ordering neighborhoods by increasing sizes. If we let parameter k control the size of a neighborhood, e.g. k-interchange in TSP, VNS starts its first neighborhood search with $k = 1$, and then increases k by one in each iteration, until there is no better solution in the current neighborhood. If a better solution is found, then k is reset to one. VNS will spend more time on the smallest neighborhoods, getting better solutions faster than using a fixed k. This fact was verified by Loudni and Boizumault [32] on a radio-link frequency assignment problem. A variant of VNS is to use completely different neighborhoods. In [35], two different neighborhood operators were used to solve the vehicle routing problem with time windows. The original VNS [34] does not explore each neighborhood completely, but starts with a random point in the current neighborhood and then performs a fast local search method (e.g. 2-opt in TSP) that tries to improve this starting point and is allowed to exit the neighborhood. By contrast, Refs. [4,30–32] use deterministic partial search methods, without any randomization. This approach is called variable neighborhood descent (VND) in [36].

A technique related to VNS is called variable depth search, including the Lin–Kernighan heuristic [37] which controls the size of a neighborhood based on gain criteria.

As pointed out by Hansen et al. [36], when the neighborhood size becomes very large, the local search inside VNS becomes too slow. These authors proposed an approximation scheme that restricts the local search to move inside the current neighborhood and not in the whole problem space. This is called variable neighborhood decomposition search (VNDS). By definition, LNS applies this strategy. Moreover, VNS applied on very large neighborhoods tends to degenerate into multistart [36]. The same conclusion was drawn by Loudni and Boizumault [32] for VND with partial search. One way to improve performance is to use VNS again for the neighborhood exploration as proposed in [36]. We will show the relative performances of these different LNS-based hybrid search methods in Section 4, on a satellite mission management benchmark.

3. Designing complex tree search algorithms in ToOLS

A search algorithm in ToOLS is a Claire object created by a functional composition of constructors called ToOLS *primitives*. This form of nested constructors defines a simple language which is easy to parse and interpret. There are three possible goals that can be applied to a search algorithm object: to search for one solution or to prove there is no solution (goal function `solve`), to search for all the solutions or to prove there is no solution (goal function `solveAll`), and to search for one optimal solution with respect to an objective function represented by a CP variable and to prove its optimality (goal functions `minimize` and `maximize`). The complete syntax is given in Appendix A. We now describe the ToOLS primitives.

3.1. Primitives for expressing a complete search tree based on refinements

Tree search methods divide a problem into simpler problems until a solution is reached. The simplification, called a refinement, consists in reducing the search space by adding some constraints. For this purpose, we use only *primitive constraints* [38] of the Eclair solver, which have a direct impact on the constraint store. For instance, the primitive constraint `x`$\leqslant$`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* [39] 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. If user-defined choice points were available, as in OPL, a complete search would not be ensured. An example of a predefined choice point is `splitleq(x, v)`, which divides 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 $n = domainsize(x)$). Below is a semantic description of the predefined
choice points:

splitleq `(x, v)  : x <= v | x > v`
splitlt `(x, v)  : x < v | x >= v`
setval `(x, v)  : x == v | x != v`
enum `(x)  : x == dom(x)[1] | x == dom(x)[2] | ··· | x == dom(x)[n]`
setdisj `(` c_1 `or` c_2 `)  : settle (`c_1 `or` c_2 `,left) | 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 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 leaf nodes of the search tree are the solutions. In optimization, a basic[3] 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* [39] is best solved by the following search algorithm:

[3] The objective function is represented by a CP variable. Thanks to constraint propagation, a simple lower-bound computation is performed in minimization.

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

The primitive `do(`$term_1$`, `$term_2$`)` creates the subtree $term_2$ at every leaf node 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 capability of Eclair. This is useful for scheduling problems, but also for expressing and managing complex choice points. It avoids creating new constraints, such as precedence constraints, during the search.

A more complex combinatorial problem, the *perfect square placement problem* [40], is best tackled 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 possible parameters *during the search.* The first argument of `delay` is a Claire function name. The subsequent arguments are passed as parameters when the function is called (they must correspond to the function interface). Here, local variable `xinf` is assigned to the smallest value in the current domain of `x`, obtained 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 to be done during the search has to be encapsulated into an object and this is precisely what the primitive `delay` does. The `delay` primitive acts as closures in functional programming languages, i.e. a piece of code (represented by a function name) together with its environment (represented by a list of parameters). 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 leaf nodes 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 1 min on a modern computer, finding eight symmetrical solutions for the 21-square problem.[4] The same search algorithm written in Claire takes the same time. The ToOLS terms, which are interpreted during the search, induce a negligible overhead on this example.

[4] Our constraint model uses logical constraints and linear equations only. Better results were obtained using the CHIP global constraints `diffn` and `cumulative` in [41].

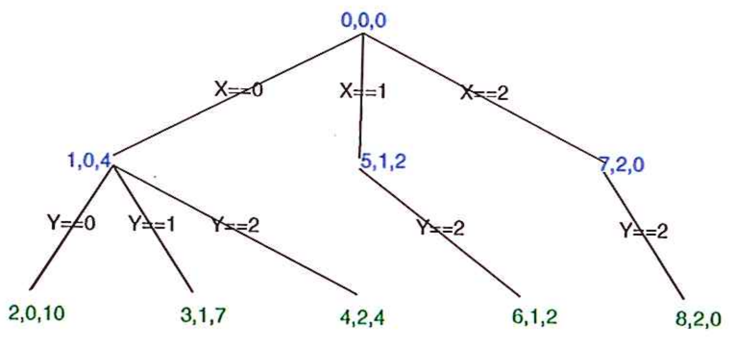


Fig. 2. A search tree with 5 solutions (leaf nodes), 4 backtracks and 8 nodes corresponding to the maximization of $2X + 3Y$ where the variables X and Y belong to [0,2], using the following search algorithm: `do(enum(`X`, increasing, `$mark_X$`), enum(`Y`, increasing, `$mark_Y$`))`. At each node, the 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 $(2*2 - 2*0) + (3*2 - 3*1) = 7$.

3.2. *Primitives for partial exploration*

We define several primitives to control the size of the explored part of a given search tree. The primitives specify conditions under which nodes may be visited. A condition is a formula, *expression* $\leqslant$ *threshold*, that must hold at any node (before posting a primitive constraint). *expression* defines a function that is used to evaluate a node, path or subtree during the search. A node is explored only when the function value is less than or equal to the given *threshold* (primitive `nodelimit`). The primitives `pathlimit`, `treelimit`, and `globallimit` are also provided. 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 evaluation functions to be useful:

- `order`: rank of a node (restricted by `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`: rating difference of a node from the preferred node (`nodelimit`). A special heuristic is used to mark each alternative of a choice point with a rating, and the one with the highest rating is designated as the preferred alternative (thus having a null rating difference). See Fig. 2.
- `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 leaf nodes in a subtree (`treelimit` and `globallimit`)

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 corresponds 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 only the best and second best alternative choices are visited, since the other alternatives have a discrepancy greater than one.

Let $while_1$ and $while_2$ correspond to both `while` terms in the perfect square example. This code applies two different discrepancy limits for each subtree:

```
do(
  pathlimit(3, sum(order), while₁),
  pathlimit (1, sum(order), while₂))
```

Another example is `treelimit(100, nbnodes, pathlimit(1, sum(order), st))` that 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 to the root. Thanks to the functional composition of primitives, we cannot have two conditions on two overlapping search trees.

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 currently deepest leaf node. For instance `nodelimit(0, order, relDepth(3, -1), st)` will explore the first two choice points completely but only the best alternative choice of the other choice points, except for the last choice point, which is completely explored. Negative depth bounds implement bounded backtrack search (BBS) [42,18]. The `distance` keyword is used when "it is not the number of discrepancies that matter, but rather the quality of the discrepancies'' [43]. 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 Fig. 2, we project the linear objective function $2X + 3Y$ onto each variable X and Y. Ref. [43] 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. Smaller values indicate smaller sizes of the explored part of a given search tree. We call the cutoff values the incompleteness parameters. 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 induced search complexity.[5] 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, beginning with the first (greedy) combination and concluding with the last (complete) one. The primitive `increasedScope` implements this strategy. We describe several iterative weakening methods using this primitive:

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

- `IB`: Iterative Broadening [16]
  increasedScope`(p, list(0,1,2, ...),`
  nodelimit`(p, order, st))`
- `LDS`: Limited Discrepancy Search [17]
  increasedScope`(p, list(0,1,2,...),`
  pathlimit`(p, sum(order), st))`
- `LDS-BBSk`: LDS & Bounded Backtrack Search [42]
  increasedScope`(p, list(0,1,2,...),`
  pathlimit`(p, sum(order), relDepth(1,-k), st))`
- `DDS`: Depth-Bounded Discrepancy Search [18][6]
  increasedScope`(p, list(1,2,...),`
  nodelimit`(0, order, relDepth(p,∞), st))`
- `DDS-BBSk`: DDS & Bounded Backtrack Search [18]
  increasedScope`(p, list(1,2,...),`
  nodelimit`(0, order, relDepth(p,-k), st))`
- `DBDFSk`: Discrepancy-Bounded Depth First Search [19]
  increasedScope`(p, list(k−1, 2k−1, 3k−1,...),`
  pathlimit`(p, sum(order), st))`

The relevant combinations are easily found when there is a single integer-valued parameter. This is not the case when there are floating point parameters, when mark heuristics are normalized, or when there are several parameters. In these cases, 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. Ref. [43] learns the *optimal cutoff policy* for a single float parameter of the *weighted discrepancy search* method from a model of the value ordering 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 [22,21,11–13].

3.3. Primitives for combining several partial explorations

The combination is described by the sequence primitive, which is 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). On the contrary to the do primitive which chains searches by hooking up the tree of hs_{i+1} at each leaf node of hs_i, the `sequence` primitive explores the tree of hs_i modulo the limits in place and proceeds to apply a new search hs_{i+1} *from the initial computation state* unless hs_i terminated without encountering any limit (complete search). A simple `sequence` example is to perform a greedy search, a partial search, and then a complete search sequentially, each search using a different search scheme. Iterative weakening methods (primitive `increasedScope`) follow this approach but use only *one* scheme

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

of problem decomposition based on the *same* heuristics. The `sequence` primitive overcomes this limitation. Typical examples are iterative sampling methods [23,15] and large neighborhood search methods [4,27,30–32].

All the searches are completely independent, except for the solutions, which are stored in a common pool, and the best cost bound in optimization, which is shared. The goal of the search, which can be `solve`, `solveAll`, `minimize`, or `maximize`, is the same for all the searches defined by a search algorithm object. Therefore, the whole search process stops as soon as a search included in the `sequence` process ends and is complete, or there is no more search to be done. As said before, ToOLS considers a search to be complete if no threshold limits were exceeded during the search, or if a solution was found for a constraint satisfaction problem (goal `solve`).

Another important primitive is needed to express the interleaving of several search algorithms, dealing with parallelism issues, including load-balancing and adaptive resource allocation depending on the effectiveness of the various search algorithms. Examples are interleaved depth-first search [24,26] and algorithm portfolios [25,27]. This primitive is not addressed in the sequel of this paper and is left for future work.[7]

ToOLS lets the user define its own *generator* function in Claire that builds a sequence of partial search algorithms dynamically. The `sequence` process will call this function at runtime in order to get the next search algorithm to be executed or a special value (`unknown`) that expresses the end of the sequence. The generator function will receive three mandatory input parameters, `step` (the search number in the sequence), `time` (the remaining CPU time), and `trace` (information about search history such as the number of solutions found so far/by the previous search or the number of nodes done so far/by the previous search), provided by the ToOLS runtime system. Because the function is encapsulated into a `delay` primitive, it can also receive optional input parameters provided by the programmer. For instance, let `hs` defines any search scheme that uses a randomized value ordering heuristic. Then the following generator function `iterator` repeats hs N times:

```
[iterator(step:integer, time:integer, trace:list, nb:integer,
              hs:HybridSearch)
 -> if (step <=nb) hs
    else unknown]
```

This function can be used to implement various iterative sampling methods [23,15] depending on which search limits are used (`randomst` corresponds to any randomized search scheme):

- `ISamp/DFSk`: iterative sampling with depth-first search limited on the number of backtracks
  sequence`(delay(iterator, N,`
    globallimit`(k, nbbacktracks, randomst)))`
- `ISamp/BBSk`: iterative sampling & bounded backtrack search
  sequence`(delay(iterator, N,`
    nodelimit`(0, order, relDepth(1,-k), randomst)))`

[7] The interleaving mechanism could rely on the multi-threading capacity of unix-like operating systems. Each search algorithm will correspond to a unix thread and will use its own distinct copy of the constraint store. Future versions of the Claire language would enable this multi-threading approach in ToOLS.

• `ISamp/LDSd-BBSk`: iterative sampling & limited discrepancy search & bounded backtrack search

```
    sequence(delay(iterator, N,
      pathlimit(d, sum(order), relDepth(1,-k), randomst)))
```

This feature is also used to implement LNS methods. When using the goal functions `minimize` or `maximize`, only the best solution is stored. In the case of LNS, the best solution found by one search is partially reused for the next search. The information which is reused can be a partial assignment or a partial schedule, as it is done in Section 4.4. The function `getLastSolution` accesses the value of the variables saved in the last solution found, i.e. the current best solution in optimization. Let `choose` be a Claire function that returns a set of variables belonging to a given problem. And let `st` be a search scheme represented by a ToOLS object. Then, the following generator function `lns` applies `st` on a restriction of the original problem, such that the variables returned by `choose` are assigned to the values contained in the last solution:

```
[lns(step:integer,time:integer,trace:list,choose:function,
st:Choice)
    -> let FrozenVariables := choose(step) in
        if (length(FrozenVariables) > 0)
            do( nodelimit(0, order,splitleq(Zero, 0)),
                forall(x, FrozenVariables,
                    tell(x, ==, delay(getLastSolution, x))),
                st)
        else unknown]
```

When the neighborhood search is complete, we cannot deduce the whole search is complete. We add a dummy choice point `splitleq` with a greedy search limit before the neighborhood definition in order to be sure that the sequence will continue to the next neighborhood even if the current neighborhood search is complete. The `choose` function manages the size of the neighborhood depending on the number of frozen variables. It can change the size by following the VNS [34] strategy, see Section 2.2. The function `vns(s,e)` does this, where `s` is the initial minimum size, and `e` is the final maximum size. Here are two examples of LNS using different partial search methods applied on a given search scheme `st`:

• `VNDSs,e/LDSd-BBSk`: one-level variable neighborhood decomposition search with limited discrepancy search & bounded backtrack search[8]

```
        sequence(
            // Compute an initial solution
            globallimit(1, nbleaves, st),
            // Improve this solution by using one-level VNDS
            // VNDSs,e
            sequence(delay(lns, vns(s,e),
                // LDSd-BBSk
                pathlimit(d,sum(order), relDepth(1, -k),st))))
```

[8] Following the terminology in [36], LNS combining with VNS is called VNDS. But, because VNS is not again used inside VNDS, we call our algorithm *one-level* VNDS, as opposed to the *two-level* VNDS initially proposed in [36].

• `VNDSs,e/DFSk:` one-level variable neighborhood decomposition search with depth-first search limited on the number of backtracks

```
sequence(
    // Compute an initial solution
    globallimit(1, nbleaves, st),
    // Improve this solution by using one-level VNDS
    // VNDSs,e
    sequence(delay(lns, vns(s,e),
        // DFSk
        globallimit(k, nbbacktracks, st))))
```

Note that the first sequence is needed in order to produce an initial solution for LNS first. Moreover, the branch and bound principle is enforced for all the searches included in the sequence process: each search cannot find a solution worse than the ones found by the previous searches. Thus, an optimization goal applied on a ToOLS algorithm implies a local descent strategy. If a non descent strategy has to be expressed, then another goal is required. We use the goal function `solveAll` that will not enforce any improvement constraint on the objective variable as `minimize` or `maximize` does. First, we describe a generic function `vnds` that creates a VNDS algorithm that uses `hs`, which can be any search algorithm, for its neighborhood exploration:

```
[vnds(s:integer, e:integer, hs:HybridSearch)
  -> sequence(delay(iterator, N,
       sequence(delay(lns, vns(s,e),
        let(solution, delay( minimize, Objective, hs),
         if(delay(isbetter, solution),
          forall(x, ProblemVariables,
           tell(x, ==, delay(getValue, x, solution))),
          // else
          FAILURE))))))]
```

Let `solveAll` be the (first) goal applied on an algorithm produced by this function. The first sequence repeats N times the LNS scheme defined in the second sequence. The neighborhood search in LNS starts by fixing some variables as it is defined by the `lns` generator function. From this subproblem, a second goal is executed using `hs` to search for the best solution that minimizes the global variable `Objective`. This second goal returns a solution which is compared with the last solution found in the master scheme (Claire function `isbetter`). If this new solution is strictly better, then it is rebuilt in the master scheme (`forall` applied on `ProblemVariables`, which contains the list of problem variables, and `getValue` that returns the value assigned to variable `x` in this solution), so it becomes a solution for the master scheme. Otherwise, the current neighborhood search stops (ToOLS primitive `FAILURE`), and `lns` generates a new neighborhood.

The following code represents a two-level variable neighborhood decomposition search algorithm using VNDS again as a local search procedure to explore the neighborhoods [36]:

`VNDSs,e/Random+VNDS1,2/LDSd-BBSk`: two-level VNDS with random initial solutions, and LDS and BBS. Let `st` be any search scheme. `randomst` can be the same search scheme as `st`, but adds randomness in the value ordering heuristics.

```
// Master VNDSs,e
vnds(s, e,
    sequence(
        // Compute a random initial solution inside the neighborhood
        // defined by vnds
        globallimit(1, nbleaves, randomst),
        // Slave VNDS1,2
        sequence(delay(lns, vns(1,2),
            // LDSd-BBSk
            pathlimit(d, sum(order), relDepth(1, -k), st)))))
```

Because we apply the `solveAll` goal on this algorithm, each neighborhood search of the master scheme starts with an initial solution the quality of which can be worse than the best solution found so far. The same goal could be used to implement population-based local search methods, by keeping a set of (diversified) solutions.

In addition, we can specify how to distribute a global time limit to all the searches included in a sequence by the notion of a *time-sharing policy* [13]. By default, the `sequence` primitive allocates the total time to the current search. An example using a static policy is `sequence(list(1,0.5),` hs_1, hs_2, hs_3`)`, that allocates the total time to the first search hs_1. Then, if all the time has not been exhausted, half of the remaining time is allocated to hs_2. Similarly, if time remains, the rest is allocated to hs_3. A dynamic policy is achieved by replacing the list of constant values by a `delay` primitive. Static and dynamic time-sharing policies, including [26,27], may improve the performance of the search. This is why the third dimension of search algorithm description in ToOLS (first one defines a complete search tree, second one a partial search) is also called *temporal strategy*, which performs several partial explorations and stipulates how to distribute the global time limit to these explorations. Dealing with real-time aspects is further discussed in [13].

4. Experiments

We present a mission management benchmark for agile satellites in order to show the expressiveness and readability of our framework, and to give a comparison of various partial and hybrid search algorithms. This section also emphasizes how important it is for performance issue to develop problem-dependent search procedures.

4.1. A mission management benchmark for agile satellites

We solved a simplified version of a problem of selecting and scheduling earth observations for agile satellites.[9] See [44,45] for a complete description. The satellite has a pool of candidate photographs to take. It must select and schedule a subset of them on each pass above a certain strip of territory.

[9] This benchmark is available in the free constraint solver choco [46].

The satellite can only take one photograph at a time (disjunctive scheduling). A photograph can only be taken during a time window that depends on the location photographed. Minimal repositioning times are required between two consecutive photographs. All physical constraints (time windows and repositioning times) 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 of the candidate photographs (binary variables) and the second one for the acquisition starting times of the selected photographs (integer variables, time accuracy in milliseconds).

4.2. The constraint model

Let N be the number of candidate photographs. For each photograph with number i ($i \in [1, N]$), let E_i be its earliest starting time, L_i its latest starting time, D_i its duration, x_i its selection variable, t_i its starting time, and r_i its revenue. Let $M_{i,j}$ be the repositioning time between any pair of photographs (i, j). M is a symmetrical matrix. The Eclair constraints are:

$$\forall i \in [1, N] : t_i \geqslant E_i, \tag{1}$$

$$\forall i \in [1, N] : t_i \leqslant L_i \quad \text{or} \quad x_i = 0, \tag{2}$$

$$\forall i \in [1, N] : t_i > L_i \quad \text{or} \quad x_i = 1, \tag{3}$$

$$\forall (i, j) \in [1, N] \times [1, N] : t_i \geqslant t_j + D_j + M_{j,i} \quad \text{or} \quad t_j \geqslant t_i + D_i + M_{i,j}. \tag{4}$$

The constraints 1,2 and 3 are time window constraints. Instead of writing $(x_i = 1) \Rightarrow (E_i \leqslant t_i \leqslant L_i)$, we assume that every candidate photograph must start after its earliest starting time but must finish before its latest starting time only if it is selected. Rejected photographs will have their starting time greater than their latest starting time. This is a valid model if the initial domains of the t_is are large enough. The constraints 2 and 3 express the link between the selection variables and the starting time variables. The constraint 4 represents minimum transition time constraints. Although the physical constraint on repositioning times concerns pairs of *consecutive selected* photographs only, constraint 4 concerns *any* pair of candidate photographs. Due to the way repositioning times are computed, based on the Euclidean distance between two points, the matrix M satisfies the triangle inequality ($\forall (i, j, k) \in [1, N]^3$, $M_{i,j} \leqslant M_{i,k} + M_{k,j}$). Therefore, if i and j are non consecutive photographs, then the repositioning time between i and j is less than the sum of the repositioning times between the consecutive photographs from i to j. Thus the constraints do not exclude any solution. The objective function is just a linear combination of the selection variables and the photograph revenues, $Gain = \sum_{i=1}^{N} r_i \times x_i$, to be maximized. This basic constraint model allows a minimum amount of constraint propagation. Eclair performs arc-bound consistency on the variables. Moreover, let c_1 `or` c_2 be a disjunctive constraint, if constraint c_1 becomes false then c_2 is enforced, and vice versa. A lesson from [44] is that more complex constraint propagation mechanisms such as global scheduling constraints are not powerful enough for this problem because of the mix between planning (selection of photographs) and scheduling.

4.3. Refinement-based search scheme

The refinement-based search scheme we used corresponds to a general search procedure for scheduling problems as described in [47]:

```
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 ordering heuristic proposed in [44] to decide whether the start time is fixed to its minimum value or postponed first. The heuristic approximates the future gain of a photograph in the following way. Let glb be the gain of the best solution found so far (recall that this is a maximization problem). Then,

$$r_j + glb\frac{\max_k L_k - \inf(t_j) - D_j}{\max_k L_k - \min_k E_k}$$

is the approximate future gain for photograph j. If the future gain for the photograph returned by `mostUrgent` is greater than the future gain for any other *related* photograph, then the start time is fixed to its minimum value first, otherwise it is postponed first. The related photographs are restricted to the ones not selected yet and whose starting time is reduced if the first alternative choice is performed. We also developed a randomly biased version of this heuristic that performs a random choice only if the absolute difference between the approximate gain of the current selected photograph and the best approximate gain of the related photographs is strictly lower than K, otherwise it acts the same as the original heuristic. We set $K = 15$ in our experiments.

Finally, a redundant constraint is added, using the `tell` primitive, when a start time t_i is postponed. The test is performed by an `if` primitive and uses a predefined function `>?` (`>?(t,tinf) : inf(t) > tinf`).

$$t_i \geqslant \min_{j \neq i} (\inf(t_j) + D_j + M_{j,i}). \tag{5}$$

This redundant constraint, called a delaying constraint in [47], 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.

4.4. Customized hybrid search algorithms

For the benchmark, we made some modifications to the generic LNS algorithms described in Section 3.3. The neighborhood subproblem was defined by fixing a subset of the photographs already selected in the last solution (x_i assigned to one), and by enforcing the same temporal order as the one given in the last solution between pairs of consecutive photographs belonging to this subset. The neighborhood generator function `lns` was the following:

```
[lns(step:integer,time:integer,trace:list,choose:function,st:Choice)
  -> let list(FrozenPhotographs,FrozenDisjunctions)
  := choose(step) in
     if (length(FrozenPhotographs) > 0)
        do(nodelimit(0, order,splitleq(Zero, 0)),
           forall(x, FrozenPhotographs,
              tell(x, ==, 1)),
           forall(d, FrozenDisjunctions,
              tell(d, delay(getLastOrder, d))),
           st)
     else unknown]
```

The function `getLastOrder` in the `tell` primitive is used to enforce the left (respectively, the right) part of constraint 4 (defined in Section 4.2) if $t_i > t_j$ (resp. $t_j > t_i$) in the last solution. In our experiments, this neighborhood scheme performed much better than just freezing the start times of a subset of the previously selected photographs.

We tried three different strategies for choosing a subset of the photographs which belongs to the last solution (function `choose`). The first basic approach, called `LNSs,e` in the experiments, chooses the photographs at random. The number of *relaxed* photographs, belonging to the last solution and not inserted in that subset, is randomly chosen between `s` and `e` before each neighborhood search. The second approach, called `LNSs,e`*sw*, adds the restriction that the relaxed photographs are temporally consecutive. The third approach, called `VNDSs,e,` is an extension of the previous one. In both cases, the number of different neighborhoods is quadratic in the size of the last solution rather than exponential as in the `LNSs,e` approach.[10] For this reason, we chose in the third approach to enumerate the neighborhoods from the smallest to the largest in a systematic way rather than randomly, by following the Variable Neighborhood Search principle. So, the first neighborhood has all the previously selected photographs except `s` photographs. If there is no improvement, the number of relaxed photographs is increased by one, and so on, until the maximum value `e` is overcome (then, `VNDSs,e` stops). If a better solution is found, this number is reset to `s`. This reseting feature improved the results. The relaxed photographs are chosen in a deterministic way using a sliding window of consecutive photographs. Therefore, if there are n selected photographs in the last solution, and the current number of relaxed photographs is equal to k, then `VNDSs,e` will generate $n-k+1$ neighborhoods containing $n-k$ frozen photographs before increasing k. In practice, this deterministic strategy was always more effective than choosing the bounds of the sliding window at random as it was done in `LNSs,e`*sw*.

[10] If there are n photographs selected in the last solution, then `LNS1,n` has up to $2^n - 1$ different neighborhoods, and `LNS1,n`*sw* and `VNDS1,n` have $(n(n+1))/2$ only.

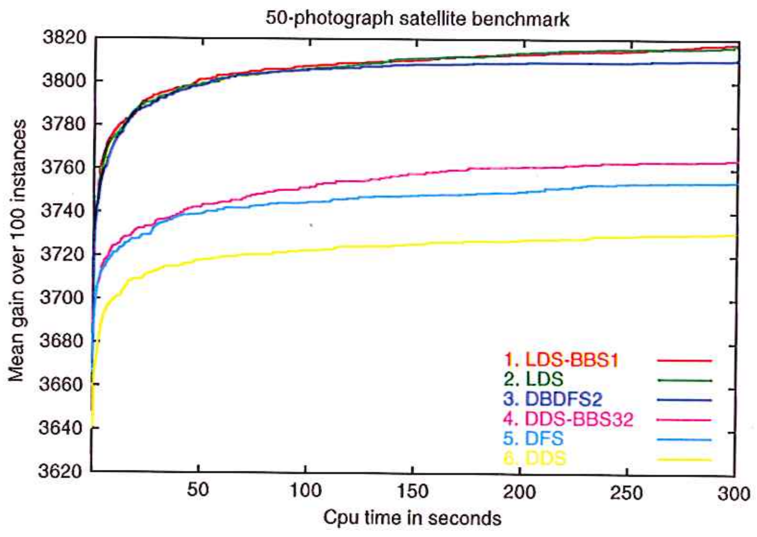


Fig. 3. Comparison of iterative weakening methods.

4.5. A comparative analysis of partial and hybrid search algorithms

We generated 100 random instances for three different numbers of candidate photographs (50, 100, and 200).[11] All the algorithms are using the same refinement-based search scheme defined in Section 4.3. Iterative sampling algorithms and LNS-based methods whose name ends with the "`(random)`" keyword use the randomized value ordering heuristic as described in Section 4.3. We also implemented the sequence-based greedy algorithm (`Greedy`) and the dynamic programming algorithm (`DPA`) described in [44,45]. `DPA` provided very good (nonoptimal) results and was really fast (less than a second for 200 photographs). However, this is a very specialized algorithm which cannot cope with new constraints. In fact the benchmark corresponds to the simplest problem defined in [44] (with a linear criterion) and the dynamic programming approach is not applicable to the more complex problem (with a nonlinear criterion and stereoscopic constraints). The constraint programming approach is applicable to both versions. Our goal was to assess and compare the quality of the results obtained by CP algorithms, and `DPA` results were used as reference values.

We tried different settings for the algorithm parameters. The complete results are presented in Tables 1 and 2. These results were obtained on a 2.4 GHz Intel XEON with Linux 2.4. Figs. 3–7 show the solution quality (mean gain over 100 problems) as time passes for problems with 50 candidate photographs (200 for Fig. 6). In the legends, algorithms are sorted by decreasing efficiency (the first one produced the best solution quality after 5 min of CPU time).

The analysis of experimental results shows:

- Fig. 3 compares iterative weakening methods. DFS is classical DFS without any search limit, except for the time limit. LDS clearly outperformed DFS and DDS. Our explanation is that the quality of

[11] These instances are available at *http://www.inra.fr/bia/ftp/T/bep/instances.tar.gz*

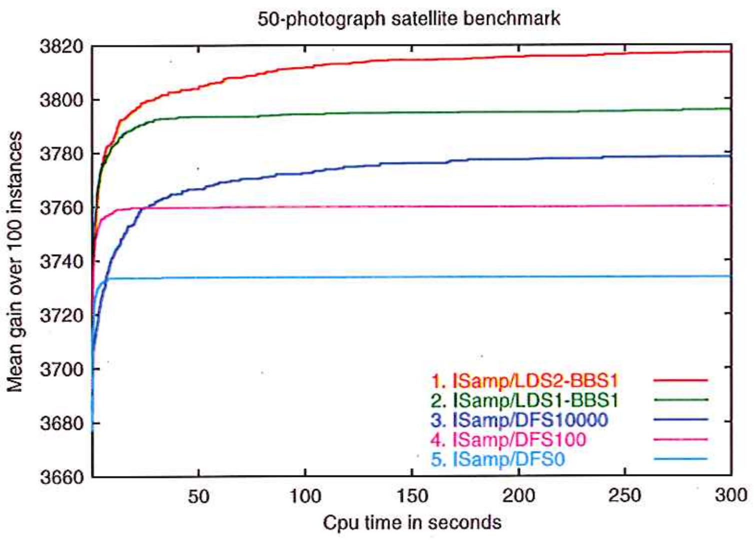


Fig. 4. Comparison of iterative sampling methods.

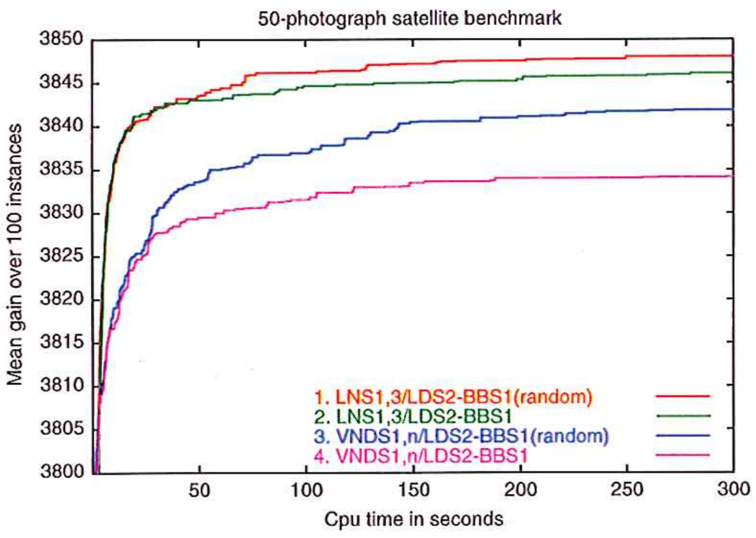


Fig. 5. Comparison of LNS-based methods.

the value ordering heuristic does not depend on the depth of the search. As it has been previously observed for scheduling problems in [42], adding a limited amount of backtracks near the leaf nodes (`LDS-BBS1`) slightly improved the results for LDS. DDS with a large amount of backtracks near the leaf nodes (`DDS-BBS32`) was also a bit better than DFS. Other LDS strategies for increasing the discrepancy more rapidly (`DBDFS2`) were counterproductive.

- Fig. 4 compares iterative sampling methods. Again, limiting the number of discrepancies (`ISamp/LDSk-BBS1`) was better than limiting the number of backtracks (`ISamp/DFSk`). Moreover,

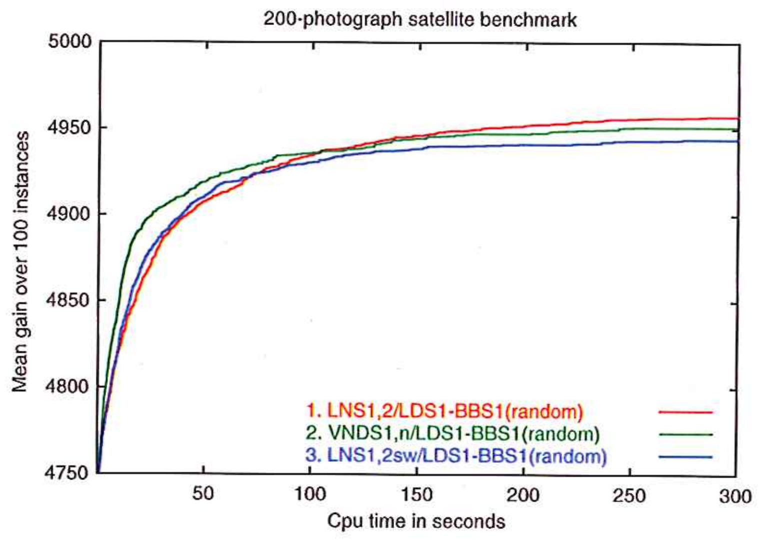


Fig. 6. Comparison of LNS-based methods on very large problems.

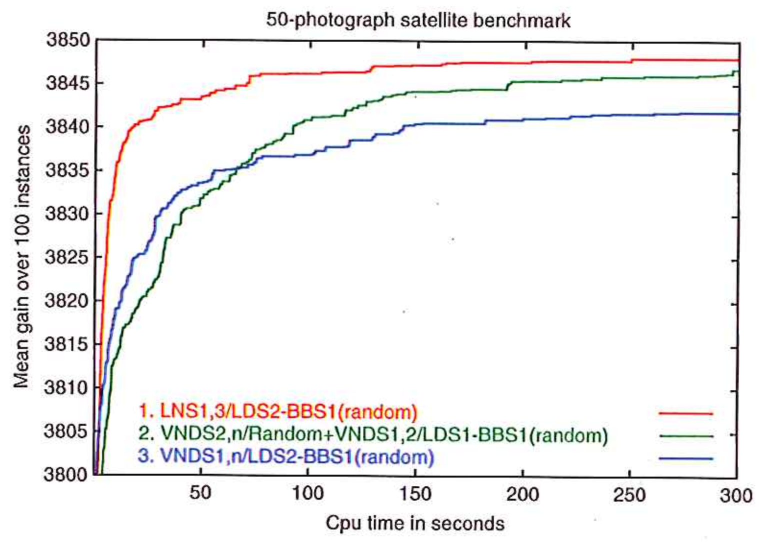


Fig. 7. Comparison of one-level and two-level LNS-based methods.

the limit was more readily ascertainable. Here, `ISamp/DFS0` corresponds to iterative sampling with a greedy search, but DFS got better results after 30 s only. Note that LDS, which is potentially a complete method, was as good as `ISamp/LDS2-BBS1` after about 5 min for 50-photograph instances.

- Fig. 5 compares LNS-based methods on problems with 50 photographs. All these methods are clearly superior to iterative weakening or sampling methods. LDS as a partial search method for the neighborhood search is reported only, other combinations produced worse results as it was the case for `ISamp`.

As shown in Table 2 on large scale problems, using a complete neighborhood search rather than a partial search performed badly when the size of the neighborhood is large. After a short period, choosing the relaxed photographs at random (`LNS1,3/LDS2-BBS1` and `LNS1,3/LDS2-BBS1(random)`) was better than choosing temporally consecutive photographs (`VNDS1,n/LDS2-BBS1` and `VNDS1,n/LDS2-BBS1(random)` where n is the maximum number of selected photographs). Surprisingly, LNS and especially VNDS both using LDS with a randomized value ordering heuristic performed better than without any randomization. Randomized VNDS was able to find solutions in larger neighborhoods on the average than without randomization. Moreover, the randomized algorithm does not stop when the maximum neighborhood size is reached (after around 230 s for `VNDS1,n/LDS2-BBS1`), but continues by reseting the neighborhood size to the minimum size. As for iterative sampling methods, randomization helps the search to escape from local optima obtained by using the same value ordering heuristic at every search. The number of neighborhood searches is significantly lower for VNDS (350 iterations and 527 iterations for randomized VNDS) than for LNS (5523 iterations) due to the difference in neighborhood sizes.[12]
If we increase the problem size or reduce the CPU time, then choosing temporally consecutive photographs can be a good strategy. Fig. 6 gives an example before 100 s of CPU time. This can be explained by the fact that the sliding window approach plus the VNS principle focussed the beginning of the search on small neighborhoods in a systematic way which is not the case for LNS and that it can reschedule consecutive photographs in order to reduce repositioning times with the opportunity to add new photographs elsewhere in the solution. In addition, the figure shows the advantage of using the VNS principle rather than using a random neighborhood size as in `LNS1,2sw`. Besides, tuning the neighborhood size for LNS is a critical step (see Tables 1 and 2), whereas VNDS has an automatic tuning.
If the time limit is large enough, after 82 s for 50-photograph instances, two-level VNDS can outperform one-level VNDS, as it is shown in Fig. 7. Two-level VNDS was able to find solutions in larger neighborhoods than one-level VNDS. The number of master/slave neighborhood searches was 626/13169 on 50-photograph problems. Tuning the slave VNDS parameters was based on the fact that more than 80% of the solutions found by one-level VNDS have one or two relaxed photographs only. As far as we know, this is the first published experimental results for two-level VNDS [36] in CP.

Verfaillie and Lemaître [44] 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(DPA − CPA)/DPA) of 26.7% and 13.3%. We dramatically reduce the main relative distance

$$\left(100\frac{(\mathrm{mean(DPA)} - \mathrm{mean(LNS1,2/LDS2-BBS1(random))})}{\mathrm{mean(DPA)}}\right)$$

to respectively, −0.05% and 0.82% for 100 and 200 photographs by using a hybrid search method.

[12] There are 19 photographs per solution on the average. Each new solution implies up to $\frac{19\times20}{2}=190$ different neighborhoods for VNDS and $C^1_{19}+C^2_{19}+C^3_{19}=19+171+969=1159$ different neighborhoods for `LNS1,3`.

Table 1
A comparative analysis of partial and hybrid search algorithms for 50 candidate photographs in 3 s, 30 s, and 5 min

Problem size CPU time	50 3 s		50 30 s		50 5 min	
	Mean	(%)	Mean	(%)	Mean	(%)
Greedy	3600	0	3600	0	3600	0
DFS	3711	48	3734	58	3754	66
DDS	3685	36	3713	49	3731	56
DDS-BBS8	3711	48	3735	58	3752	66
DDS-BBS16	3716	50	3739	60	3756	67
DDS-BBS32	3709	47	3736	59	3764	71
LDS	3757	68	3793	83	3818	94
LDS-BBS1	3759	69	3795	84	3818	94
LDS-BBS2	3756	67	3793	83	3817	94
LDS-BBS4	3754	66	3789	82	3814	93
DBDFS2	3751	65	3792	83	3811	91
DBDFS4	3747	63	3790	82	3806	89
ISamp/DFS0	3730	56	3734	58	3734	58
ISamp/DFS100	3753	66	3760	69	3760	69
ISamp/DFS1000	3750	65	3766	72	3770	73
ISamp/DFS10000	3720	52	3762	70	3779	77
ISamp/LDS1-BBS1	3768	73	3793	83	3796	85
ISamp/LDS2-BBS1	3768	73	3800	86	3817	94
ISamp/LDS4-BBS1	3729	56	3771	74	3815	93
LNS1,2/DFS	3792	83	3841	104	3843	105
LNS1,2/LDS1-BBS1	3817	94	3833	101	3833	101
LNS1,2/LDS2-BBS1	3812	92	3837	103	3837	103
LNS1,2/LDS2-BBS1 (random)	3815	93	3842	105	3844	106
LNS1,3/DFS	3750	65	3829	99	3847	107
LNS1,3/LDS1-BBS1	3811	91	3838	103	3840	104
LNS1,3/LDS2-BBS1	3806	89	3842	105	3846	106
LNS1,3/LDS4-BBS1	3776	76	3837	103	3847	107
LNS1,3/LDS2-BBS1 (random)	3809	90	3842	105	3848	107
LNS1,5/DFS	3712	48	3758	68	3828	99
LNS1,5/LDS1-BBS1	3808	90	3834	101	3839	103
LNS1,5/LDS2-BBS1	3791	83	3837	103	3844	106
LNS1,5/LDS4-BBS1	3738	60	3820	95	3847	107
LNS1,2sw-LDS2-BBS1	3799	86	3801	87	3801	87
LNS1,2sw-LDS2-BBS1 (random)	3796	85	3800	86	3800	86
LNS1,3sw-LDS2-BBS1	3805	89	3813	92	3813	92
LNST1,3sw-LDS2-BBS1 (random)	3804	88	3816	93	3816	93

Table 1 (*continued*)

Problem size CPU time	50 3 s		50 30 s		50 5 min	
	Mean	(%)	Mean	(%)	Mean	(%)
LNST1,5sw-LDS2-BBS1	3794	84	3827	98	3828	99
LNST1,5sw-LDS2-BBS1 (random)	3785	80	3827	98	3829	99
VNDS1,n/DFS	3802	87	3818	94	3825	97
VNDS1,n/LDS1-BBS1	3805	89	3822	96	3822	96
VNDS1,n/LDS2-BBS1	3807	90	3828	99	3834	101
VNDS1,n/LDS4-BBS1	3804	88	3822	96	3834	101
VNDS1,n/LDS2-BBS1 (random)	3807	90	3830	100	3842	105
VNDS2,n/Random+VNDS1,2/LDS1-BBS1	3781	78	3827	98	3844	106
VNDS2,n/Random+VNDS1,2/LDS1-BBS1 (random)	3791	83	3823	96	3847	107
DPA	3830	100	3830	100	3830	100

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 (mean(Algorithm) − mean(Greedy))/(mean(DPA) − mean(Greedy)).

5. Related work

Localizer [5] was proposed for local search algorithms. It is based on invariants instead of constraints, and does not use constraint propagation. SaLSA [7] was a first attempt to provide a language that unifies global and local search methods extending the concept of choice point into a versatile *move* operator. Partial search methods are expressed by using the filtering predicates for moves; several examples are described in [12]. 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 on a specific class of local/global hybridization. OPL [6] was 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) and difficult to extend. ToOLS is an extensible 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 [10] 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. Calling any Claire functions during the search is possible thanks to the `delay` primitive. Pure local search could be inserted into a tree search by calling an external procedure.

Perron [48] introduced a unified approach for the design of partial search methods, later included in OPL [49], based on a priority queue used to store the current set of open search nodes (as in best-first search).

Table 2
A comparative analysis of partial and hybrid search algorithms for 50, 100, and 200 candidate photographs in 5 min

Problem size CPU time	50 5 min		100 5 min		200 5 min	
	Mean	(%)	Mean	(%)	Mean	(%)
Greedy	3600	0	4205	0	4732	0
DFS	3754	66	4290	36	4790	21
DDS-BBS16	3756	67	4313	45	4807	27
LDS-BBS1	3818	94	4364	67	4855	46
ISamp-LDS1-BBS1	3796	85	4383	75	4874	52
ISamp-LDS2-BBS1	3817	94	4381	74	4842	41
LNS1,2-DFS	3843	105	4434	97	4864	49
LNS1,2-LDS1-BBS1	3833	101	4428	94	4955	83
LNS1,2-LDS2-BBS1	3837	103	4438	99	4948	80
LNS1,2-LDS2-BBS1 (random)	3839	103	4438	99	4958	84
LNS1,2-LDS2-BBS1 (random)	3844	106	4442	100	4959	84
LNS1,3-DFS	3847	107	4384	76	4805	27
LNS1,3-LDS1-BBS1	3840	104	4433	97	4951	81
LNS1,3-LDS2-BBS1	3846	106	4437	98	4937	76
LNS1,3-LDS1-BBS1 (random)	3843	105	4438	99	4959	84
LNS1,3-LDS2-BBS1 (random)	3848	107	4442	100	4944	79
LNS1,2sw/LDS1-BBS1	3795	84	4396	81	4936	76
LNS1,2sw/LDS2-BBS1	3801	87	4403	84	4935	75
LNS1,2sw/LDS1-BBS1 (random)	3793	83	4406	85	4944	79
LNS1,2sw/LDS2-BBS1 (random)	3800	86	4408	86	4941	77
LNS1,3sw/LDS1-BBS1	3805	89	4401	83	4930	73
LNS1,3sw/LDS2-BBS1	3813	92	4411	87	4929	73
LNS1,3sw/LDS1-BBS1 (random)	3810	91	4413	88	4947	80
LNS1,3sw/LDS2-BBS1 (random)	3816	93	4419	91	4938	76
LNS1,5sw/LDS1-BBS1	3816	93	4408	86	4935	75
LNS1,5sw/LDS2-BBS1	3828	99	4420	91	4908	65
LNS1,5sw/LDS1-BBS1 (random)	3826	98	4422	92	4950	81
LNS1,5sw/LDS2-BBS1 (random)	3829	99	4424	93	4908	65
VNDS1,n/LDS1-BBS1	3822	96	4411	87	4941	77
VNDS1,n/LDS2-BBS1	3834	101	4420	91	4940	77
VNDS1,n/LDS1-BBS1 (random)	3835	102	4427	94	4951	81
VNDS1,n/LDS2-BBS1 (random)	3842	105	4424	93	4946	79
VNDS2,n/Random+VNDS1,2/LDS1-BBS1	3844	106	4428	94	4936	76
VNDS2,n/Random+VNDS1,2/LDS1-BBS1 (random)	3847	107	4430	95	4939	77
DPA	3830	100	4440	100	5000	100

Mean is the mean value of the best solution found after 5 min for 100 randomly-generated instances. The percentage is equal to 100 (mean(Algorithm)−mean(Greedy))/(mean(DPA) − mean(Greedy)).

This priority queue mechanism has a potential risk of memory explosion that becomes problematic in case of large scale combinatorial optimization problems. The search procedure framework [48] also defines static cutoffs in terms of backtracks, nodes or CPU time, where the priority queue is not necessary and can be removed. But, if we want to implement for instance LDS with an increasing number of discrepancies, the priority queue mechanism is needed. On the contrary, the ToOLS primitive `increasedScope` changes the cutoffs dynamically from one search to another. ToOLS keeps the depth-first search principle (chronological backtracking) in all cases. Only the current search path is stored, avoiding any memory problems. For iterative weakening methods, ToOLS will revisit search nodes while Ref. [48] will perform state recomputation. DLDS [20] is an attempt to improve the results in [48] by changing the way of doing state recomputation, but it still has the same worst-case memory complexity. OPL Script [50] allows several searches to be combined, eventually on different models, but it was not designed for global/local hybrid search. Compared with OPL [49], the major novelty of ToOLS is the addition of the third dimension for the sequencing (and interleaving) of distinct algorithms. Such features do correspond to a novel form of support for trying multiple heterogeneous search procedures without the need for scripting.

Mozart/OZ [51] is a concurrent constraint programming language. Search strategies are programmed using primitives which rely on first-class computation spaces. This feature allows explicit manipulation of search tree states. But this language provides no high level primitive dedicated to partial and hybrid search. Eclipse [52] provides several incomplete tree search methods by means of parameterized routines. This CLP solver also includes repair techniques and predefined local searches. CHIP [2] implements the concept of parameterized search. Parameterized search allows the design of partial search algorithms thanks to predefined parameterized objects. The CHIP system offers search limits, like fail limitation for a variable, and combination of subtrees.

6. Conclusions

Recent progress has been made in improving the efficiency of search algorithms used in constraint programming solvers, including Ilog Solver [1]; CHIP [2] or Eclipse [52] by replacing depth-first search by partial search methods and with possible extension to large neighborhood search. In our opinion, these new search methods should not be provided as black-box functions, but rather as a combination of search primitives in order to be able to exploit the specificities of the problem we want to solve. To express partial search methods, we found that it was more convenient and comprehensible to represent them by a set of search limits (number of nodes, number of discrepancies, relative to tree depth intervals) plus an iterative scheme that specifies how the limits will change from one iteration to another, rather than by a specification of the order in which nodes are visited. We recommend using a clear decomposition for the design of search algorithms: the definition of a complete search tree, a set of conditions for which nodes of that tree are visited and a (temporal) strategy for combining several partial explorations. Each part of the decomposition scheme can be reused separately and several combinations can be tried.

Another experiment on a military application showed that partial search methods significantly improve the solution quality compared to an existing customized greedy algorithm [11]. Moreover, it demonstrated a reduction in development time of customized search algorithms, compared to the traditional approach. The code is clearer and more concise when using the ToOLS primitives. Efficiency issues were taken into account throughout all the design process and the implementation of ToOLS (e.g. search limits are computed incrementally, the code interpreting choice points is optimized). The whole framework, Claire, Eclair, and ToOLS, has been successfully integrated in an operational on-board hard real-time system implemented at Thales [8]:

Following Shaw's recommendation [4], our experimental results confirmed the efficiency of using partial search methods inside large neighborhood search, rather than partial search alone. LNS not only introduces no model overhead, but also enables the exploration of neighborhoods that are large enough to offer more benefits than classical neighborhood enumeration. If more time is available, two-level variable neighborhood decomposition search [36] might prove to be a powerful strategy. This remains to be confirmed on other benchmarks. To deal with the complexity of designing problem-specific partial and hybrid search algorithms, future work should address the way of learning the values of search limits, the combination of search limits, and the parameter values of LNS strategies, as it has been done in genetic programming [53]. In particular, more work should be done on how to distribute a given time contract to every search included in a hybrid search method.

Acknowledgements

We thank Michel Lemaître for providing us the benchmark on agile satellites. Many thanks to the reviewers of this paper and also to the reviewers of a preliminary version submitted to the CP-AI-OR'03 workshop for their useful comments. This work was partially funded by the RNRT EOLE project [54].

Appendix A. 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 } )
         -> FAILURE
         -> <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 )
```

References

[1] ILOG SA. ILOG: ILOG Solver 6.0 user's manual, 2003.

[2] Cosytec SA. CHIP V5, 1997.

[3] Beldiceanu N, Bourreau E, Chan P, Rivreau D. Partial search strategy in CHIP. In: Proceedings of the second international conference on meta-heuristics, Sophia-Antipolis, France, 1997.

[4] Shaw P. Using constraint programming and local search methods to solve vehicle routing problems. In: Proceedings of CP-98, Pisa, Italy, 1998. p. 417–31.

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

[6] Van Hentenryck P. OPL: The optimization programming language. Cambridge, MA: The MIT Press; 1999.

[7] Laburthe F, Caseau Y. SALSA: a language for search algorithms. Constraints 2002;7(3):255–88.

[8] de Givry S, Jeannin L, Josset F, Mattioli J, Museux N, Savéant P. The THALES constraint programming framework for hard and soft real-time applications. The PLANET Newsletter, Issue 5 ISSN 1610-0212, December 2002. p. 5–7. http://planet.dfki.de/service/Resources/Rome/degivry.pdf (slides).

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

[10] Caseau Y, Josset FX, Laburthe F. Claire: combining sets, search and rules to better express algorithms. In: Proceedings of ICLP'99, Las Cruces, New Mexico, 1999. p. 245–59.

[11] de Givry S, Savéant P, Jourdan J. Optimisation combinatoire en temps limité: Depth first branch and bound adaptatif. In: Proceedings of JFPLC-99, Lyon, France, 1999. p. 161–78.

[12] Jourdan J, de Givry S, Savéant P. Designing limited search algorithms for time constrained combinatorial optimization problems. Technical report, Thales Research & Development, 1999. http://www.inra.fr/bia/T/degivry/Givry99c.ps.gz.

[13] de Givry S, Hamadi Y, Mattioli J, Gérard P, Lemaître M, Verfaillie G, Aggoun A, Gouachi I, Benoist T, Bourreau E, Laburthe F, David P, Loudni S, Bourgault S. Towards an on-line optimisation framework. In: CP-2001 workshop on on-line combinatorial problem solving and constraint programming (OLCP'01), Paphos, Cyprus, December 1, 2001. p. 45–61. http://www.inra.fr/bia/T/degivry/Givry01d.ps.gz.

[14] Hogg T, Huberman BA, Williams CP. Phase transitions and the search problem. Artificial Intelligence 1996;81(1–2):1–15.

[15] Gomes C, Selman B, Kautz H. Boosting combinatorial search through randomization. In: Proceedings of AAAI-98, Madison, WI, 1998.

[16] Ginsberg ML, Harvey WD. Iterative broadening. Artificial Intelligence 1992;55:367–83.

[17] Harvey WD, Ginsberg ML. Limited discrepancy search. In: Proceedings of IJCAI-95, Montréal, Canada, 1995. p. 607–13.

[18] Walsh T. Depth-bounded discrepancy search. In: Proceedings of IJCAI-97, Nagoya, Japan, 1997.

[19] Beck JC, Perron L. Discrepancy-bounded depth first search. In: Proceedings of CP-AI-OR'2000, Paderborn, Germany, March 8–10, 2000.

[20] Michel L, Van Hentenryck P. A decomposition-based implementation of search strategies. ACM Transactions on Computational Logic 2004;5(2).

[21] Chu L-C, Wah BW. Optimization in real time. In: Proceedings of the twelfth real time systems symposium, Washington, DC, 1991. p. 150–9.

[22] Korf RE. Real-time heuristic search. Artificial Intelligence 1990;42:189–211.

[23] Bresina JL. Heuristic-biased stochastic sampling. In: Proceedings of AAAI-96, Portland, OR, 1996. p. 271–8.

[24] Meseguer P. Interleaved depth-first search. In: Proceedings of IJCAI-97, Nagoya, Japan, 1997. p. 1382–7.

[25] Gomes C, Selman B. Algorithm portfolios. Artificial Intelligence 2001;126(1,2):43–62.

[26] Prcovic N, Neveu B. Progressive focusing search. In: Proceedings of ECAI-02, Lyon, France, 2002. p. 126–30.

[27] Perron L. Fast restart policies and large neighborhood search. In: Proceedings of CP-AI-OR'2003, Montréal, Canada, 2003. p. 246–60.

[28] Pesant G, Gendreau M. A constraint programming framework for local search methods. Journal of Heuristics 1999;5: 255–79.

[29] Applegate D, Cook B. A computational study of the job shop scheduling problem. Operations Research Society of America 1991;3(2).

[30] Caseau Y, Laburthe F. Disjunctive scheduling with task intervals. Technical report, LIENS Technical report 95-25, Paris, 1995.

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

[32] Loudni S, Boizumault P. Solving constraint optimization problems in anytime contexts. In: Proceedings of IJCAI-03, Acapulco, Mexico, 2003. p. 251–6.

[33] de Givry S, Jeannin L. ToOLS: a library for partial and hybrid search methods. In: Proceedings of CP-AI-OR'2003, Montréal, Canada, 2003. p. 124–38.

[34] Mladenovic N, Hansen P. Variable neighbourhood search. Computer and Operations Research 1997;24:1097–100.

[35] Gendreau M, Rousseau L-M, Pesant G. Using constraint-based operators to solve the vehicle routing problem with time windows. Journal of Heuristics 2002;8:43–58.

[36] Hansen P, Mladenovic N, Perez-Brito D. Variable neighborhood decomposition search. Journal of Heuristics 2001;7: 335–50.

[37] Lin S, Kernighan BW. An effective heuristic algorithm for the traveling salesman problem. Operations Research 1973;21:498–516.

[38] Bockmayr A, Kasper T. Branch-and-infer: a unifying framework for integer and finite domain constraint programming. INFORMS Journal of Computing 1998;10(3):287–300.

[39] Van Hentenryck P. Constraint satisfaction in logic programming. Logic programming series. Cambridge, MA: The MIT Press; 1989.

[40] Van Hentenryck P. Intelligent scheduling. Chapter: scheduling and packing in the constraint language cc(FD). Los Altos, CA: Morgan Kaufmann; 1994.

[41] Beldiceanu N, Bourreau E, Simonis H. A note on perfect square placement. CSPLib, 1999.

[42] Harvey WD. Nonsystematic backtracking search. Ph.D. thesis, Stanford University, March 1995.

[43] Bedrax-Weiss T. Optimal search protocols. Ph.D. thesis, University of Oregon, 1999.

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

[45] Lemaître M, Verfaillie G, Jouhaud F, Lachiver J-M, Bataille N. Selecting and scheduling observations of agile satellites. Aerospace Sciences and Technology 2002;6:367–81.

[46] Laburthe F 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/

[47] Baptiste P, Le Pape C. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. Constraints 2000;5(1,2):119–39.

[48] Penron L. Search procedures and parallelism in constraint programming. In: Proceedings of CP-99, Alexandria, Virginia, October 11–14, 1999. p. 346–60.

[49] Van Hentenryck P, Perron L, Puget J-F. Search and strategies in OPL. ACM Transactions on Computational Logic (TOCL) 2000;1(2):285–320.

[50] Van Hentenryck P, Michel L. New trends in constraints, chapter OPL script: composing and controlling models. Berlin: Springer; 2000.

[51] Schulte C. Programming constraint services. Doctoral dissertation, Universitat des Saarlandes, Saarbrucken, Germany, 2000.

[52] IC-Parc. Eclipse 5.7, 2004. http://www-icparc.doc.ic.ac.uk/eclipse/.

[53] Banzhaf W, Nordin P, Keller RE, Francone FD. Genetic programming: an introduction. Los Altos, CA: Morgan Kaufmann Publishers, Inc; 1997.

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