Search Procedures and Parallelism in Constraint Programming

Laurent Perron*

ILOG SA
9, rue de Verdun, BP85, 94253 Gentilly Cedex, France
perron@ilog.fr

Abstract. In this paper, we present a major improvement in the search procedures in constraint programming. First, we integrate various search procedures from AI and OR. Second, we parallelize the search on shared-memory computers. Third, we add an object-oriented extensible control language to implement complex complete and incomplete search procedures. The result is a powerful set of tools which offers both brute force search using simple search procedures and parallelism, and finely tuned search procedures using that expressive control language. With this, we were able both to solve difficult and open problems using complete search procedures, and to quickly produce good results using incomplete search procedures.

1 Introduction

Combinatorial Optimization and Combinatorial Problem Solving is an interesting application for Constraint Programming (CP). It has grown out of the research world into the industrial world as demonstrated by the success of different products (ILOG optimization suite, CHIP).

However, when looking at real applications, it appears that Depth-First Search, which is a standard way of searching for solutions is often an obstacle to optimization. DFS, which is directly linked to the Constraint Logic Programming and the SLD resolution of Prolog, needs exponential time to break out of subtrees if initial choices have been incorrect. Searching for solutions using Depth-First Search usually goes against reactivity and robustness. Therefore, real applications, to be successful, have to override this limitation by using tailored search techniques. Over the years, many different approaches have been proposed: A simple approach using some *greedy heuristics*, coupled with some look-ahead can be interesting in scheduling [5]. Another traditional approach which has often proved successful is to change the *order of evaluation of nodes* [8, 20, 12] in the search tree. These approaches usually give good results, but they are difficult to implement in a generic way on any application of Constraint Programming. Usually, only a specific search procedure is implemented, or only on a limited subset of problems. Another promising family of techniques is based on *Local-Moves* and *Local-Search* methods. The first technique consists in applying local moves to a particular

* This research is partially funded by the Commission of the European Union contract ESPRIT 24960. The author is solely responsible for the content of the document.

solution of the problem until a improvement is found [3, 14, 13, 18]. The second one consists in freezing part of a problem and applying a tree-search based optimization process on the non-frozen part [2, 15]. Statistical methods have also been proposed: *Genetic Algorithm* [1, 6], *Simulated Annealing* [1]. All these techniques are appealing as they provide good results. They also have drawbacks: (1) Their implementations are linked to the problem they attack. Genericity is rare as it implies a uniform API to represent different problems and different optimization those implementations rely on. For instance, Simulated Annealing is based on a notion of temperature that may prove difficult to exhibit on different classes of problems. (2) They usually not extensible. Few efforts have been made to propose paradigm which offers genericity and versatility. (3) Recently, some very interesting ideas have been proposed [10, 19], but they lack usability as they provide (in our opinion) an API at too low a level, which ruins the expressiveness of their proposal.

Our work is a bottom-up approach which aims at improving the search performance of our constraint system. The starting point is tree search. We then distinguish between two orthogonal notions: *Search Heuristics* which are linked to the definition of the search tree and *Search Procedures* which are linked to the exploration of this search tree. This article deals only with search procedures. We try to improve the exploration of a search tree using (1) an expressive extensible object-oriented language to describe search procedures, and (2) parallelism to give more computing power.

We provide an efficient implementation of those search procedures – which leads to significant reductions in running time – and parallelism based on a multi-threaded architecture covering the full functionality of the sequential constraint solver. The combination of the search procedures and parallelism is a necessary element for solving large combinatorial problems such as crew rostering, large jobshop scheduling problems, and others.

Experimental results included in the paper show impressive speedups on jobshop problems, and demonstrate the benefits of both parallelism and the new search procedures.

The rest of the paper is organized as follows: Section 2 defines a few concepts on search trees. These concepts are the basis of the search procedures we introduce in section 3. Then, section 4 presents some examples of the use of the search procedures, while section 5 gives an overview of the experiments we have conducted.

2 Open Nodes and Search Procedures

In order to implement user-defined search procedures, we need to be able to get access to individual parts of the search tree. These parts are called *open nodes*. Once open nodes are defined, we can present how one search process can explore a given search tree arbitrarily. These are the fundamental building blocks needed to implement generic search procedures.

2.1 Open Nodes and Node Expansion

To model search, the search tree is partitioned into three sets: the set of *open nodes* or *the search frontier*, the set of *closed nodes* and the set of *unexplored nodes*. These sets

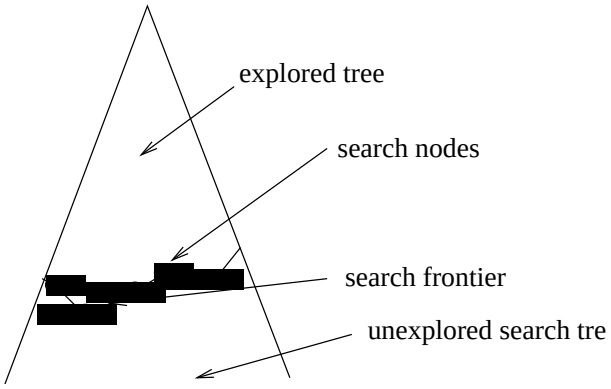
evolves during the search through an operation called *node expansion*. Throughout this paper, we will make the assumption the search tree is a binary search tree. This is true in our implementation.

Open Nodes and Search Frontier At any point during the search, the set of nodes of the search tree is divided into three independent subsets: the set of *open nodes*; the set of *closed nodes*; the set of *unexplored nodes*.

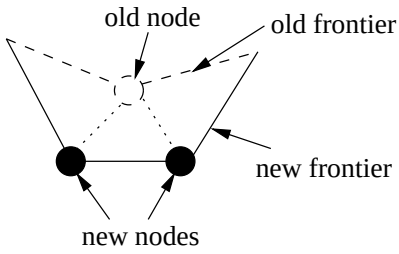
These subsets have the following properties:

- All the ancestors of an open node are closed nodes.
- Each unexplored node has exactly one open node as its ancestor.
- No closed node has an open node as its ancestor.

The set of open nodes is called the *search frontier*. The following figure illustrates this idea.



Node Expansion The search frontier evolves simply through a process known as *node expansion*. (Node expansion corresponds to the *branch* operation in a branch & bound algorithm.) It removes an open node from the frontier, transforms the removed node into a closed node, and adds the two unexplored children of that node to the frontier. This move is the only operation that happens during the search.

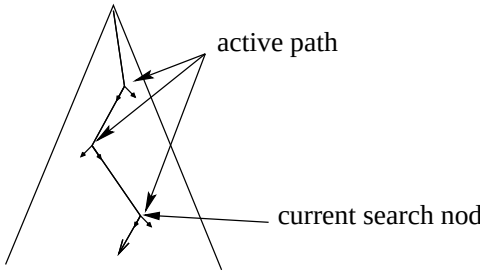


We shall later use the method to show how basic search procedures such as LDS[8] can be described as different selections of the next open node to expand.

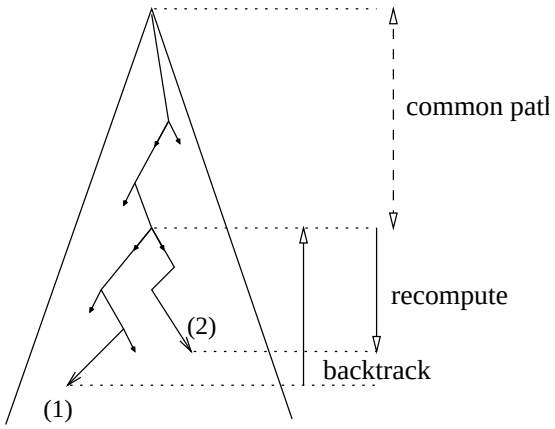
2.2 Active Path and Jumps in the Search Tree

Expanding one node after another may require changing the state (the domains of the variables of the problem) of the search process from the first node to the second. However, from a memory consumption point of view, it is unrealistic to maintain in memory the state associated with every open nodes of the search tree. Therefore, the search process exploring the search tree must reconstruct each state it visits. This is done using an *active path* and a *jumping* operation.

Active Path in a Search Tree When going down in the search tree, our search process builds an *active path*. An active path is the list of ancestors of the current open node, as illustrated in the following figure.



Jumping in the Search Tree When our search engine moves from one search node to another, it must jump in the search tree. To make the jump, it computes the maximum common path between the current position (1) and the destination. Then it backtracks to the corresponding choice point (the lowest node in the maximum common path). Next, it recomputes every move until it gets to (2). Recomputation does not change the search frontier as there is no node expansion during that operation.



2.3 Discussion of the implementation of the Branch & Bound Paradigm

Other methods exist to implement the Branch & Bound paradigm. As the problem is closely related to the implementation of or-parallelism in Prolog, a comparison with available implementations can be enlightening. In [17], a complete presentation of the different techniques developed for parallel Prolog was made: many proposals have been made. These proposal were primarily motivated by hardware considerations. There are basically two kinds of parallel computers: *shared* memory and *non-shared* memory computers. From these two categories of computer come two categories of parallel implementation.

As we may deal with large problems, we cannot save the problem at each node. Thus our implementation must use recomputation. Furthermore, managing multiple binding to the same variable leads to quadratic behavior. Therefore, we decided that each search process would have its own copy of the problem. Thus, faced with the constraint of robustness, extensibility and memory consumption, it appeared early in our search design that the only practical choice was an implementation of the Delphi Model [4] on shared-memory computers. In our implementation, each search process share the same search tree with nodes being represented as binary paths in this tree.

3 A Object-Oriented Language for Search Procedures

Search heuristics and search trees are defined using goals. There are different goals: predefined goals and user-defined goals. Two important predefined goals are the `And` goal and the `Or` goal. They are the most fundamental blocks used when writing search heuristics.

Our approach relies on these goals. A search procedure will be defined as a constructor that will be build with search goals and special objects. Basically, each atomic search procedure, itself a goal, will be written as: `Function (goal, object)`. We will present three different `functions`: `Apply` to change the order of evaluation of nodes, `SelectSearch` to select some leaves out of a search tree, and `LimitSearch` to limit the exploration of a search tree. These functions will be applied onto goals using `objects`. We will present predefined `objects` for each `function` and also a quick overview of how user-defined `objects` can be created.

3.1 Changing the Order of Evaluation of Nodes

Many references demonstrate the usefulness of changing the order of evaluation of nodes in a search tree [16, 8]. We perform these changes using the `Apply` function and instances of the `NodeEvaluator` class.

Node Evaluators for the `Apply` function A node evaluator is an object which has two purposes: evaluate a node and decide whether the search engine should jump from one node to another. This leads to an implementation of the B&B scheme where (1) the default behavior is chronological backtracking, and (2) nodes are stored in a priority queue, ordered by their evaluation.

Thus, the term `Apply(Goal g, NodeEvaluator e)` returns a goal which applies the evaluator `e` to the search tree defined by the goal `g`. This changes the order of evaluation of the open nodes of the search tree defined by `g` according to `e`.

Some Common Node Evaluators We implemented several important search procedures. **Depth First Search:** This is the standard search procedure. **Best First Search:** We have implemented a variation of Best First Search Strategy with a parameter ϵ. When selecting an open node, we determine the set of open nodes the cost of which is at most ϵ (this is a difference and not a factor) worse than the best open node. If a child of the current node is in the set, we go to this child. If not, we choose the best open node. **Limited Discrepancy Search:** Limited discrepancy search was first defined in [8]. We did not implement the original scheme but a variation name *Discrepancy Bounded Depth First Search*. This variation will be presented more fully in a future paper. The *discrepancy* of a search node is defined as its right depth, that is, the number of times the search engine has chosen the right branch of a choice point to go from the root of the search tree to the current node. Given a parameter k, the discrepancy search procedure is one that will first explore nodes with a discrepancy less than k. After this exploration is complete, it will explore nodes with a discrepancy between k and $2k$, and so on. This search procedure cuts the search tree into *strips*. **Depth Bounded Discrepancy Search:** This search procedure was introduced in [20]. It is a variation of LDS which makes the assumption that mistakes are more likely near the top of the search tree than further down. For this reason, not the number of discrepancies but the depth of the last one is used to evaluate the node. We do not implement Walsh's schema exactly, but a version we found more robust. Rather than the depth of the deepest discrepancy, we consider the depth of the w^{th} deepest discrepancy, where w is a parameter of the search. This parameter is in fact an allowed width after the depth limit of the last discrepancy as described in the original DDS design. **Interleaved Depth First Search:** This search procedure was introduced by [12]. It tries to mimic the behavior of an infinite number of threads exploring the search tree. We use a variation which limits the depth limit of this interleaving behavior.

Defining a New Node Evaluator A node evaluator is linked to the life cycle of an open node. When a node is created, the method `evaluate` is called to give the node its evaluation. This method returns a real value which will be used when the node is stored. When the manager has to decide whether it should jump to another node, the method `subsume` is called. This function is called with the evaluations of the current open node and of the best open node. A return value of **true** indicates that the search engine should jump from the current open node to the best one.

For instance, we could implement the Best-First Search evaluator this way in a C++ syntax:

```
IntBFSEvaluator::IntBFSEvaluator(IntVar var, IlcInt epsilon) {
  _epsilon = epsilon;
  _var = var;
}
```

```
Float IntBFSEvaluator::evaluate(const SearchNode) const {
  return _var->getMin();
}

Bool IntBFSEvaluator::subsume(Float val1, Float val2) const {
  return  (val1 + _epsilon <= val2);
}
```

This node evaluator is constructed with a variable and an epsilon. The evaluation of a node is the minimum of the variable. The function subsume will return **true** if the minimum of the variable for the best node is lower than the current minimum value of the variable minus epsilon.

3.2 Selecting Leaves of a Search Tree

In a minimizing process, or in a incomplete search, it may be interesting to focus on some promising leaves of a search tree and to forget the other leaves. This is done using the `SelectSearch` function and instances of the `SearchSelector` class

A Goal to Select Leaves A search selector is used to select (or filters) leaves of a search tree. It has three purposes: to store leaves of the search tree and to re-activate them once the search tree is completely explored, to implement a minimization process, and to perform a simple feasibility check on nodes.

Thus, the term `SelectSearch(Goal g, SearchSelector s)` is a goal which applies the selector `s` to the search tree defined by `g`. A selector then executes the complete search tree and selects leaves of this search tree.

Some Search Selectors Some simple search selectors are already implemented. **Minimization:** The selector `Minimize(IntVar v)` implements a minimization process on the integer variable `v` and selects a leaf which minimizes this variable. **FirstSolution:** The selector `FirstSolution(int n)` simply selects the first n solutions of a search tree.

Defining a New Search Selector Defining a new search selector is more complex than defining a new node evaluator. There are three types of methods to implement:

Minimization Management: To implement a minimization process, the search selector must check whether a known upper bound on the objective is better than the current upper bound of the objective. In this case, the constraint stating that the objective is strictly less than this known bound is imposed. This information is stored so that it can be used during recomputation to re-post the same constraint. **Feasibility Test:** A feasibility test is implemented using an evaluation function which returns a real value and a test which decides whether the evaluation attached to a node corresponds to an infeasible node. When a node is declared infeasible, it is simply postponed and not

evaluated. **Leaf Management:** When the search engine arrives at a leaf of the search tree, it can decide to store it, to delete an old leaf, or to forget it. When the search tree is completely explored, another method is called to re-activate stored leaves.

3.3 Search Limits

In real application, one cannot afford to see its search for solution lost in a uninteresting sub-search tree. Therefore is needed some way to limit the time spent by a search process in any part of the search tree. This is done using the `LimitSearch` function and instances of the `SearchLimit` class.

A Goal to Limit Search A search limit is a function which implements a periodic test to decide whether a particular limit has been reached. When this limit arrives, the set of open nodes covered by this limit are discarded. Thus, a call to `LimitSearch(Goal g, SearchLimit l)` returns a goal which limits the exploration of the search tree defined by `g` with the limit `l`. Once a limit is crossed, all nodes explored afterwards are discarded.

Pre-defined Limits Two simple limits are offered. **Time limit:** The limit `TimeLimit(double time)` creates a search limit. With this limit, the search engine explores the search tree for only `time` seconds. Afterwards, all remaining open nodes are discarded. **Failure limit:** The limit `FailLimit(int numOfFails)` creates a search limit. With this limit, the search engine explores the search tree until `numOfFails` failures have been encountered. Afterwards, all unexplored open nodes in the search tree are discarded.

Defining a New Search Limit A limit is used to prune part of the search tree. Its main method is a check method which indicates whether the limit has been reached.

3.4 Parallelism

In parallel search, different instances of the search engine (called workers) run on different processes and explore the same search tree. The workers communicate and coordinate their work via a virtual communication layer.

A virtual communication layer must fulfill three tasks: **Starvation Balancing:** The layer must propagate the work and insure that no worker is starving while there is work available; this means moving nodes from one worker to another. **Load Balancing:** The layer can periodically balance good open nodes such that every worker can work on a promising part of the search tree. **Termination Detection:** The layer must also have a mechanism to detect termination (every worker is starving). In this case, it must terminate the search cleanly.

In our design(figure 1), the storage of open nodes is distributed over the different workers, the same open node cannot be expanded by two different workers. This choice reduces the synchronization cost between workers and minimizes the differences between the sequential code and the parallel code.

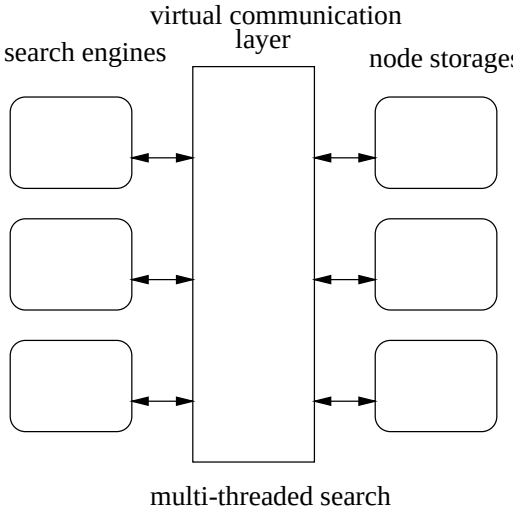


Fig. 1. Parallel Solver Architecture

3.5 Comparison with related work

All our work is greatly inspired from Salsa [10]. There are three main improvements over F. Laburthe's original design:

1. Salsa is based on choice points whereas our work is based on search goals and their corresponding search tree. Thus, basic search procedures like LDS are natural in our design. They are not so natural in Salsa because these basic search procedures are difficult to implement on a choice point level.
2. Furthermore, as we are dealing with goals, we do not need the two channels we find in Salsa (the leaf channel and the solution channel).
3. Our design is completely extensible. We can define easily new limits, new evaluators and new selectors. This is not the case in Salsa.

4 Using the Object-Oriented Language for Search Procedures

4.1 Some Small-Scale Examples to Illustrate Search Procedures

Given a goal `g`, we can write different search goals using our Search Language. We suppose we have a function `solve` which takes a goal as parameter and returns the first solution of this goal. Therefore,

- `solve(g)` will search for the first solution of the search tree defined by `g`.
- `solve(Apply(g, LDSEvaluator(3)))` will search in LDS with strips of width 3 for the first solution of the search tree defined by `g`.

– `solve(SelectSearch(g, Minimize(var)))` will minimize the variable `var` in the search tree defined by `g`. When `solve( )` returns, the solver is in the state of the best solution.
– `solve(SelectSearch(LimitSearch(g, TimeLimit(5.0)), Minimize(var)))` will return the best solution (according to `var`) found in a 5-second time frame in the search tree generated by `g`.

4.2 Using Search Procedures to Implement a Complex Incomplete Search

Using Limited-Discrepancy Search to solve a problem is already an improvement over standard Depth-First Search. But our language for search procedures can be used in a more ambitious way. We will present an example where we search for solutions by composing two goals, `inner1` and `inner2`. We will change the search procedures such that the search (a) uses limits to keep the process from being stuck in a subtree; and (b) implements a two-phase decomposition of the problem to have a better overview of the complete search tree than with the limited one.

The original goal is equivalent to :
`LimitSearch(And(inner1, inner2), TimeLimit(t))` where `t` is a parameter of the search. Later, we will adjust `t` such that exactly the same time will be spent searching with the original goal and our complex goal. This will allows us to compare their behavior in the same time frame.

The First Phase This phase will try to solve the first part of the problem using a Depth-Bounded Discrepancy Search, along with a fail limit. We will restrain ourselves to the first five leaves of the search tree.The corresponding goal is:

```
Goal apply1 = Apply(inner1, DDSEvaluator(2, 2));
Goal limit1 = LimitSearch(apply1, FailLimit(15));
Goal restrict1 = SelectSearch(limit1, FirstSolution(5));
```

The goal `limit1` will explore the search tree associated with the goal `inner1` only until 15 failures happen. This search tree will be explored by increments of 2, with 2 remaining discrepancies after the depth limit. Afterward, this goal will simply fail. Thus the effect of the `LimitSearch` function is to prune part of the search tree. The goal `restrict1` will keep the first 5 solutions of the goal `limit1`.

The Second Phase This second phase will solve the second part of the problem using a Limited-Discrepancy Search. We will limit the maximum number of discrepancies during the search to 1 in order to have a simple limit on the time spent. The code is the following: `Goal apply2 = Apply(inner2, LDSEvaluator(1, 1));`

The Complete Goal The complete goal is the composition of the two previous goals (`limit1` and `apply2`) embedded in a minimization process. The complete goal along with the search for solution results in the following code:

```
Goal comp1 = And(restrict1, apply2);
SearchSelector minim = Minimize(totalCost);
Goal minimizeGoal = SelectSearch(comp1, minim);
```

The Explored Search Tree Simple graphs can illustrate the explored part of the search tree in our case (figure 2) and in the original case (figure 3). The complex search is quite different than the simple limited one we could have obtained with a time limit. With this complex search procedure, we have a better overview of the complete search tree using our complex goal instead of the original one, we claim that our complex goal is more robust. This will be verified experimentally in section 5.2.

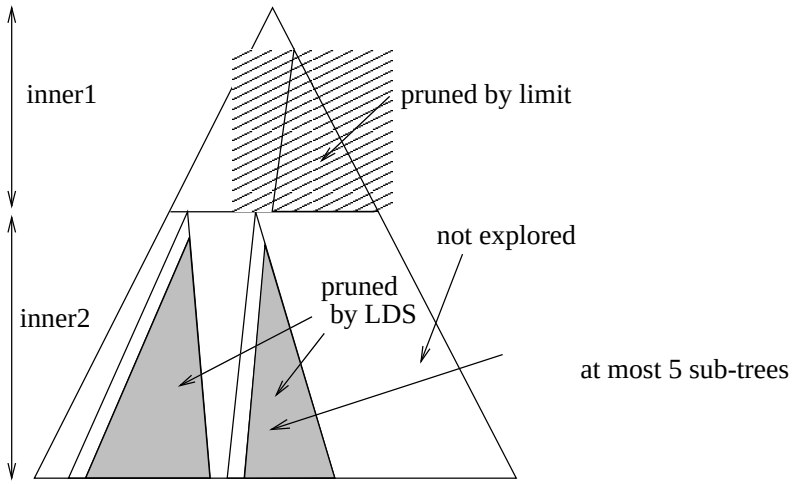


Fig. 2. Complex Search

5 Results

In this section, we will present some results to illustrate the benefits of the search procedures and parallelism. In these examples, the standard behavior is defined by a depth first search procedure and a single processor search.

All the times are given in seconds. The computer used was a 4 processors Pentium Pro 200Mz computer running Linux. The compiler used was egcs 1.1.1.

This was implemented on top of ILOG SOLVER and ILOG SCHEDULER.

5.1 Complete Search on Jobshops Problems

We will try complete search procedures on different jobshops in order to see how they find good solutions and how they prove them

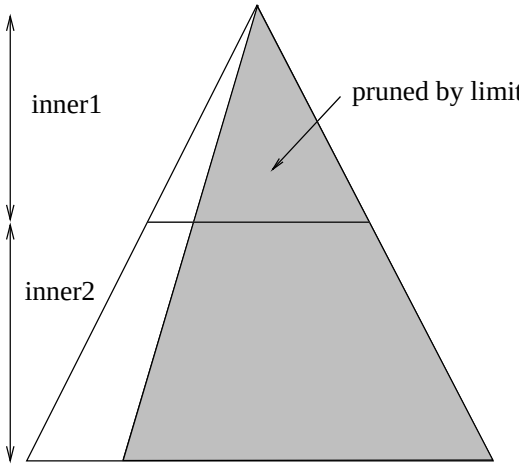


Fig. 3. Limited Simple Search

Two Jobshop Examples These two jobshops come from [7]. The search heuristic is a simple one based on ranking of activities. The edge finder algorithm is used at each stage. The time to find the best solution and the total time (best solution + proof) is given.

Problem	Strategy	1 worker	4 workers
MT10	DFS	434.79 s / 654.80 s	
MT10	LDS	83.12 s / 536.97 s	34.57 s / 177.87 s
MT20	DFS	not found	not found
MT20	LDS	111.50 s / 144.97 s	10.85 s / 11.95s

These examples demonstrate clearly the benefit of both parallelism and changing the order of evaluation of search nodes. The most interesting part is that LDS changes a problem too difficult to solve in a reasonable time into an easy one.

A Small Example : Abz5 Abz5 is a small jobshop (10×10) from [9]. It has been solved using various strategies. In the table below, the first figure represents the time (in seconds) to get the optimal solution; the second figure represents the total running time (best solution + proof of optimality). We can deduce that on a small problem, parallelism, as it gives more breadth during the exploration of the search tree, leverages the differences between different search procedures. However, when it comes to proving optimal solution, the best method is DFS as it implies no recomputation as opposed to other methods. This is visible with one processor when comparing LDS and DFS on a problem where the proof is difficult like this one. LDS finds the optimal solution much faster than DFS, but the total time is greater.

Strategy	1 worker	2 workers	4 workers
DFS	89.15 s / 98.98 s	52.90 s / 60.27 s	14.87 s / 22.11 s
BFS	92.91 s / 125.55 s	62.18 s / 78.41 s	10.67 s / 20.02 s
BFS with tuning	131.95 s / 133.85 s	65.39 s / 66.01 s	9.14 s / 28.8 s
LDS with k=4	35.04 s / 104.68 s	19.04 s / 52.06 s	3.86 s / 18.49 s
LDS with k=2	51.12 s / 116.71 s	40.24 s / 72.12 s	11.29 s / 32.47 s
DDS	63.98 s / 112.88 s	16.34 s / 36.72 s	2.21 s / 21.4 s
IDFS	51.77 s / 58.46 s	29.45 s / 32.07 s	12.37 s / 13.95 s

A Bigger Example : la36 la36 is a medium jobshop (15×15) from [11]. For solving these problems, a global time limit of 500 seconds was set on the resolution of these problems. With this time limit, IDFS and un-tuned BFS are not able to find a solution, even with 4 processors. The table below illustrates the fact that when the problem gets more difficult, the difference in robustness between search procedures becomes more evident. Which a medium problem and a good search heuristic, LDS and DDS appear more robust.

Strategy	1 worker	2 workers	4 workers
DFS	>500 s / >500 s	>500 s / >500 s	231.07 s / 246.71 s
BFS with tuning	>500 s / >500 s	>500 s / >500 s	100.37 s / 210.79 s
LDS with k=4	297.35 s / 361.12 s	241.69 s / 282.36 s	93.2 s / 141.4 s
LDS with k=2	402.35 s / 464.06 s	211.82 s / 252.57 s	131.23 s / 163.37 s
DDS	411.79 s / 479.90 s	269.87 s / 362.79 s	47.75 s / 83.60 s

An Open Problem: swv01 Using LDS, 4 processors and a good ranking goal and shaving, we were able to solve completely an open scheduling problem: SWV01 (20×10)

Strategy	No of Workers	Time for Best solution	Total time
LDS	4	451385 s	461929 s

Thus, five and a half days were necessary on a Quadri Pentium Pro computer to break this difficult problem and prove the optimal solution to be **1407**.

5.2 Incomplete Search on Some Jobshops

We have adapted the two goals of the section 4.2 on jobshop problems. The goal `inner1` correspond to the complete ordering of two resources. The goal `inner2` corresponds to the ranking of all remaining activities.

We fixed some parameters of the complex goal. We keep the 10 first solutions of the goal `limit1`. The goal `limit1` has a failure limit of 50. The width of the LDS part of goal `inner2` is respectively 1 in case A and 2 in case B. To compare with the original goal, we ran the original goal with a time limit equal to the total running time of the complex goal. When doing this, we can compare the objective value found. Here are the results on 4 different jobshops : ABZ5, FT10, LA19 and LA21. One cell display the running time for the modified goal, the bound found and the bouind found in the same time by the original goal.

Problem	**Abz5**(1165)	**FT10** (930)	**LA19** (842)	**LA21** (1046)
Case A	2.87s : 1272/1256	2.18s : 1029/1044	2.27s : 884/900	15.08s : 1126/1135
Case B	5.62s : 1272/1256	2.86s : 1013/1044	5.19s : 867/867	35.09s : 1098/1127

We can see that except on easy problems where good solution are found fast using the original goal (Abz5), the complex goal give consistently better results than the original goal.

6 Conclusion

In this paper, we have stressed the importance of overriding the Depth-First Search limitation in constraint programming. We believe we have proposed an elegant language for both brute force search and finely tuned complex search procedures. We have shown that (a) this language is expressive and useful, (b) search procedures and parallelism greatly improves greatly the performance of our search engine, namely ILOG SOLVER and (c) this whole design can be implemented efficiently: the overhead between parallel DFS with 1 processor and the original ILOG SOLVER DFS is as low as 2-3% for the jobshop examples we have solved in this paper.

We think there are many perspectives for this work. The most important one being (a) using this control language to implement special search procedures like adaptative search for real time optimization problem; (b) implementing parallelism and distributed-memory architecture using PVM or MPI; and (c) extending the language itself with other `functions` implementing other useful behaviors.

Finally, in the long run, we would like to integrate other search techniques not based on tree search into this framework: for example local moves, statistical methods, genetic algorithms. This would allow us to prove our driving assumption which is that by improving the expressivity of search procedures, we increase radically the set of problems we can solve effectively.

Acknowledgement

First, I would like to thank the referee for their insightful comments on my article.

Then, I would like to thank everybody who helped me in my work, especially Paul Shaw for doing much more than a complete proofread of this article and Jean-François Puget for helping me getting out of design and implementation problems, and François Laburthe, for his work on Salsa, which was a wonderful start for my own work.

References

1. E.H.L. Aarts, P.J.M. van Laarhoven, J.K. Lenstra, and N.J.L. Ulder. A computational study of local search algorithms for job shop scheduling. 6:113–125, 1994.
2. D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.
3. J.W. Barnes and J.B. Chambers. Solving the job shop scheduling problem using tabu search. *IEE Transactions*, 27/2:257–263, 1994.

4. W. Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.
5. M. Dell'Amico and M. Trubian. Appplying tabu-search to the job-shop scheduling problem. *Annals of Operational Research*, 41:231–252, 1993.
6. U. Dorndorf and E. Pesch. Evolution based learning in a job shop scheduling environment. *Computers and Operations Research*, 22:25–40, 1995.
7. G.L. Thompson H. Fisher. Probabilistic learning combinations of local job-shop scheduling rules. In G.L. Thompson J.F. Muth, editor, *Industrial Scheduling*, pages 225–251. Prentice Hall, Englewood Cliffs, New Jersey, 1963.
8. William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceeding of IJCAI*, volume 1, pages 607–613, August 1995.
9. E. Balas J. Adams and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401, 1988.
10. François Laburthe and Yves Caseau. Salsa: A language for search algorithms. In *Principles and Practice of Constraint Programming*, number 1520 in LNCS, pages 310–325, 1998.
11. S Lawrence. *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques*. Carnegie-Mellon University, Pennsylvania, 1984. supplement.
12. Pedro Meseguer. Interleaved depth-first search. In *International Joint Conference on Artificial Intelligence*, volume 2, pages 1382–1387, August 1997.
13. L. Michel and P. Van Hentenryck. Localizer: A modelling language for local search. In *CP'97*, number 1330 in LNCS. Springer, 1997.
14. E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.
15. W. Nuijten and C. Le Pape. Constraint-based job shop scheduling with ILOG SCHEDULER. *Journal of Heuristics*, 3:271–286, 1998.
16. Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.
17. Laurent Perron. An implementation of or-parallelism based on direct access to the MMU. In *Proc. of Compulog-Net worshop on parallelism and implementation technology, JICSLP'96*, 1996.
18. G. Pesant and M. Gendreau. A view of local search in constraint programming. In *CP'96*, number 1118 in LNCS, pages 353–366. Springer, 1996.
19. Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.
20. Toby Walsh. Depth-bounded discrepancy search. In *International Joint Conference on Artificial Intelligence*, volume 2, pages 1388–1393, August 1997.