Compositional Abstractions for Search Factories

Guido Tack and Didier Le Botlan

Programming Systems Lab, Saarland University, Germany
{tack,botlan}@ps.uni-sb.de

Abstract. Search is essential for constraint programming. Search engines typically combine several features like state restoration for backtracking, best solution search, parallelism, or visualization. In current implementations like Mozart, however, these search engines are monolithic and hard-wired to one exploration strategy, severely complicating the implementation of new exploration strategies and preventing their reuse.
This paper presents the design of a *search factory* for Mozart, a program that enables the user to freely combine several orthogonal aspects of search, resulting in a search engine tailored to the user's needs. The abstractions developed here support fully automatic recomputation with last alternative optimization. They present a clean interface, making the implementation of new exploration strategies simple. Conservative extensions of the abstractions are presented that support best solution search and parallel search as orthogonal modules. IOzSeF, the Interactive Oz Search Factory, implements these abstractions and is freely available for download.

1 Introduction

Constraint programming is at the heart of the Mozart programming system. Mozart provides a high-level language for describing the search problem in terms of propagators and distributors.

For programming exploration strategies, on the other hand, the situation is unsatisfactory: recomputation and exploration strategies are usually defined jointly, using low-level primitives. As a result, the implementor requires not only a deep understanding of the underlying abstractions but also carries the burden of implementing efficient recomputation strategies. This is a complex task: the book-keeping that is necessary for recomputation, especially when combined with other techniques like parallel search, last alternative optimization or branch & bound best solution search, is rather involved and a source of subtle bugs.

As an example, consider the `Search` module, the `Explorer` and parallel search (also called distributed search sometimes). All of these modules define independent search engines. Although most of the code for recomputation or search is similar, it is duplicated and therefore hard to maintain. Besides, if one wishes to implement a new exploration strategy, it is not possible to benefit from visualization for free, nor from parallelism.

Contribution This paper presents the design of a search factory for Mozart.

The specific contributions of this paper are abstractions for the search tree that provide fully automatic, encapsulated recomputation and a clean interface for implementing exploration strategies. Recomputation is only efficient when combined with last alternative optimization. The paper shows how to make this automatic and orthogonal. As two more complex modules of a search factory, branch & bound search and parallel search are considered. The paper shows that they can be made orthogonal to exploration strategy, recomputation and last alternative optimization by conservative extensions to the base abstractions. The correctness of the resulting parallel search engine is discussed.

All these abstractions are modeled in the Mozart object system, using inheritance to combine them. The resulting system is called IOzSeF, the integrated Oz search factory. It is freely available and can be used as a replacement for the Mozart search libraries including the Explorer.

This work is based on and extends previous research on search in constraint programming [13, 2, 3]. Differences and similarities will be discussed in the paper, in particular in the related work section (Section 7).

Organization of this paper The next section gives a short overview of some of the main concepts used in this paper. Section 3 introduces space nodes, the abstraction that encapsulates recomputation, and gives details about their implementation. Section 4 builds the tree node layer on top; it provides a tree interface to search. In the same section, we show that these abstractions lead to a simple and concise way of formulating exploration strategies. Section 5 explains how to extend space and tree nodes to refine the search engine: we address branch & bound optimization, parallel search, and last alternative optimization. Section 6 discusses IOzSeF, the Mozart implementation of the search factory. The related work is summarized in Section 7. We conclude in Section 8.

2 Concepts

In Mozart/Oz, *computation spaces* are used to implement constraint propagation, branching (also called distribution) and search. A computation space is created by applying the system procedure `Space.new` to a unary procedure, the *search script*, which contains the problem specification. Propagation immediately starts. The space becomes *stable* as soon as no more propagation can occur. It is up to the *search engine* to react on the space's state, which may be one of `alternatives`, `succeeded` or `failed`. Spaces with alternatives are *choice points*: the search engine *commits* to one of the alternatives (which triggers propagation again), choosing one of the possible branches.

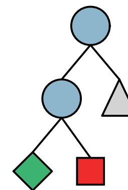


Fig. 1: A search tree

A search engine thus traverses a tree: inner nodes represent choices, leaf nodes can be either failed or contain solutions. Such a tree is called a *search tree* and

can be drawn as in Figure 1 (circles stand for choices, diamonds for solutions, squares for failures, and the triangle represents a yet unexplored subtree). Search engines must perform backtracking: once they reach a leaf node, they restore the state of the search engine to for example its parent node in order to explore the next branch. For state restoration, you can either memoize each node in the search tree, or have a method to reconstruct its state. In Mozart, memoization is called *cloning*, and the reconstruction of a state can be achieved by *recomputation*. The state of a node in the search tree can be reconstructed by redoing the choices on the path from the root node or any other ancestor node that has been cloned during search.

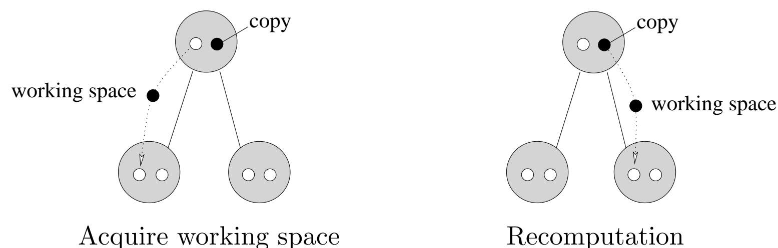
3 Abstracting Recomputation

In this section, we present the *space node* interface, an interface to computation spaces that abstracts from recomputation. From the outside, every space node looks as if it contained a computation space. Internally, space nodes perform recomputation automatically whenever it is needed.

We basically provide the same abstraction as the `Node` class introduced by Choi et al. [3]. We take a slightly different perspective though and split the interface into two parts: a space node deals with recomputation, but is not concerned with the interface for a search engine programmer. Search engines are built on top of the tree node interface, which is the topic of the next section.

At the implementation level, each space node contains two attributes for computation spaces:

- The first one possibly contains a *working space*, which represents its node's state (including propagators, distributors and constraint variables).
- The second one possibly contains a *copy*. Space nodes with a copy are used as a basis for recomputation.



Acquire working space   Recomputation

In addition, as in the `Node` class in [3] again, space nodes are organized in a tree with parent links and store the number of the alternative they represent. The straightforward way of implementing such a tree in Mozart is reflected in the following interface:

```
class SpaceNode
   attr workingSpace                    %% space or empty
   attr copy                            %% space or empty
   attr parent                          %% SpaceNode
   attr alternative                     %% int

   meth constructRoot(Root)             %% space
   meth newChild(Alternative ?Child)    %% int, returns SpaceNode
   meth ask(?Status)                    %% returns status
end
```

The constructor is straightforward. The `newChild` method takes an integer and creates a node representing that alternative, but with empty working space and copy. The `ask` method returns the status of a node's working space (whether it is failed, solved, or has alternatives for branching). What happens if the space node does not have a working space, for example because it has just been created? Space nodes obey the following protocol for creating and communicating computation spaces:

- If the parent has a working space, it will give it to its child.
- If the parent does not have a working space, `recompute`.

Recomputation Given that there is a copy at least in the root of the search tree (which we will assume from now on), the basic recomputation mechanism can be implemented in Mozart in a straight-forward way:

```
meth recompute(?C)
   if @copy\=empty then {Space.clone @copy C}
   else
      {@parent recompute(C)}
      {Space.commit C @alternative}
   end
end
```

There are two obvious recomputation strategies:

- Full recomputation: let `constructRoot` place a copy in the root node.
- No recomputation: let `recompute` place a copy in every node.

These two make it very clear that recomputation is a means of trading space for time. Schulte [12] discusses recomputation in detail (including a comparison to trailing). Really efficient recomputation requires more sophisticated strategies that can be implemented by refining the `recompute` method:

- Fixed recomputation: place a copy in every n-th node (n is called *maximal recomputation distance* or *MRD*).
- Adaptive recomputation: place a copy on the middle of the path between the node that is to be recomputed and the ancestor it is recomputed from.

4 Abstracting the Search Tree

On top of the space node interface, the second abstraction layer is built, namely *tree nodes*. Space nodes make recomputation fully transparent: tree nodes need no knowledge of the underlying recomputation strategy.

In this section, we first elaborate tree nodes as a high-level interface to search trees. Then, we show how to use the tree node interface to implement exploration.

4.1 The Tree Node Interface

The tree node interface provides a simple interface to the search tree that abstracts from its dynamic construction.

Indeed, the full structure of the search tree can only be known by computing the status of each single node that indicates if it is a leaf or branching. However, we want to avoid the full construction of the tree prior to search because it is exactly the role of the search engine to explore the tree. As a consequence, it is necessary to build tree nodes lazily, that is, only once they are required by the search engine. To sum up, the tree node interface can be seen as a regular tree interface, although nodes of the tree are only built on demand.

Trees can be implemented in many different ways. In the following, we present an object-oriented interface to tree nodes; however, the same technique can be easily adapted to any other tree representation. Tree nodes are implemented following this interface:

```
class TreeNode from SpaceNode
   feat Children                  %% TreeNode tuple
   meth constructRoot(RootSpace)  %% space
   meth getChildren(?Children)    %% return TreeNode tuple
end
```

The constructor for the root node requires an initial computation space. Then, the exploration of the tree is performed using the method `getChildren`.

Implementation As mentioned above, the main point is to create tree nodes lazily. We can use Mozart's by-need mechanism to achieve this: `getChildren` invokes `ask` to find out how many children to create, and then initializes `children` to a tuple of by-needs. The implementation is shown in Figure 4.1.

There are three levels of laziness in this design: The tuple of children is created only when `getChildren` is called, each child node is constructed by-need, and the underlying space node methods lazily copy and transfer their computation spaces.

4.2 Exploration Strategies

Implementing an exploration strategy is now as simple as traversing a tree. This makes the following code sample look like a text-book version of a depth-first tree traversal:

```
meth getChildren($)
   if {IsFree self.Children} then
      case {self ask($)} of alternatives(N) then
         self.Children = {MakeTuple c N}
      in
         {Record.forAllInd self.Children
          fun {$ I}
             {ByNeed fun {$} {self newChild(I $)} end}
          end}
      else self.Children=c
      end
   end
   self.Children
end
```

Fig. 2. Implementation of `getChildren`

```
proc {Explore Node}
   {Record.forAll {Node getChildren($)} Explore}
end
```

Incremental search It is possible to get more control over the search process, for example by defining a stateful search engine. Its interface consists of two methods `initSearch` and `nextSolution`. For parallel search (see Section 5.2), a more fine-grained control is necessary: we require only one exploration step, that is, explore only one node at each call.

Control You may have noticed that search algorithms do not handle the case that a solution was found. We leave this task to a separate `Control` module that takes care of collecting solutions, setting up the root node and starting and stopping search. Some of the extensions presented in the following sections will also require global control, always realized as extensions to the `Control` module.

5 Extensions

The architecture we have so far can be extended in orthogonal ways to support some more advanced search techniques. The features we develop here in detail are *branch & bound search* for solving optimization problems, *parallel search* for distributing a search problem over several computers, and *last alternative optimization*, a technique that reduces the number of copies in the search tree. All extensions happen below the tree node interface, making them completely orthogonal to the implementation of exploration strategies.

5.1 Branch & Bound Optimization

A well-known mechanism for solving optimization problems is the *branch & bound* metaphor: each time a solution is found, every node that remains to be

explored is constrained to yield a "better" solution (in terms of a given order). Branch & bound therefore maintains the invariant that every solution that is found is better than the previous one. As a direct consequence, the last solution is the globally best one. In practice, this optimization considerably reduces the size of the search tree by pruning whole subtrees that cannot give better solutions.

Implementation model Logically, every time a new solution is found, it is put in special nodes between all unexplored children and their parents. Each time a space is "pushed over" such a node (for example when a working space is given to a child, or during recomputation), the constraint that the space must be better is injected.

In Figure 3, the logical view of branch & bound is illustrated: assume the right child needs recomputation. It makes use of its mother's copy by cloning it, which gives a new computation space. Since a "best" constraint lies between these two nodes, the space must be constrained to yield a better solution (1) before being passed to the child (2). Still, this mechanism is transparent and does not appear in the space node or tree node interfaces: the special nodes are automatically inserted and traversed.

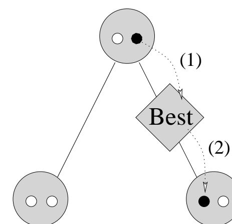


Fig. 3: Recomputation and B&B

Implementation This scheme can be implemented by inheriting from the `SpaceNode` class and refining the `ask` and `recompute` methods to post the additional "best" constraints. The `Control` module maintains the current globally best solution. The `newChild` method of `SpaceNode` is refined so that it inserts a special node reflecting the current best solution when a child is created.

As this does not influence the tree node interface, optimization is completely orthogonal to search: all search engines can be used without any modification for solving optimization problems using branch & bound.

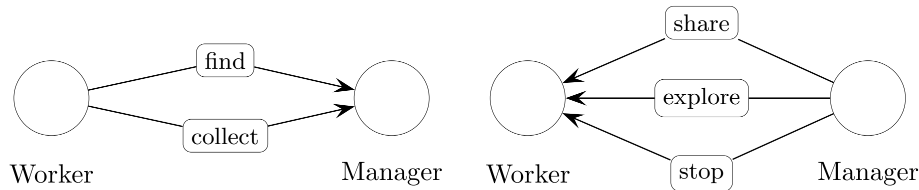
5.2 Parallel Search

In this section, we consider parallel search and show how it can easily fit within our layered abstractions. As a result, all exploration strategies designed over the tree node interface can be immediately used in a network-distributed setting.

Parallel search speeds up exploration of large constraint problems, by taking advantage of several networked computers that independently explore disjoint subtrees. This can be nicely implemented in Oz, as described by Schulte [13].

The main actors of the parallel search framework are a single *manager* and several *workers*. The former implements network distribution: it dispatches independent parts of the search problem to the workers and gathers solutions. In the case of branch & bound, the manager also propagates solutions in order to constrain each worker to yield a better solution.

The interface between the workers and the manager can be represented as follows (see Chapter 9 of [13] for a more detailed presentation of this interface):



Implementation Implementing the parallel search framework amounts on the one hand, to setting up the network distribution layer, which makes use of the Mozart distribution library; on the other hand, to providing the actual code corresponding to the messages of the interface. We only focus on the messages that are not straightforward to implement.

The `explore` message carries along a path that describes the location of a tree node in the search tree. In order to reconstruct a node given its path, a straightforward extension to `SpaceNode` is needed, namely a method `fromPath` that builds a space node from a given path. Then, a search engine independent of the parallel search implementation can be used, starting at the given tree node. Still, each worker must be able to reply concurrently to a `share` message, that is, to provide the path of a subtree that remains to be explored. One way to do so is to use an incremental search engine, as described in Section 4.2, and to extend its interface with a method `getUnexploredPath` that (may) return a path to some unexplored node. In order to maximize the work being shared, it is usually wise to return the highest unexplored node in the tree, because it is likely to correspond to the largest unexplored subtree. The tree node and space node interfaces need to be enriched with a straightforward `getPath` method.

In the case of branch & bound search, solutions sent by the manager must be taken into account by the workers. This requires an interaction with the `Control` module to update the current best solution. As the tree node interface remains unchanged, branch & bound optimization is independent of the exploration strategy.

Since the parallel search only relies on the standard interface of tree nodes (up to the additional method `fromPath`), it is possible to freely use different implementations of tree nodes and space nodes. Thus, for instance, it is possible for some workers to display a graphical representation of the subtree being explored, and to dynamically choose the most suitable recomputation technique.

Discussion In the following, we show that the network-distributed setting is correct in the sense that it yields the same solutions as a single search engine. The main difficulty arises from the fact that it is possible to use different exploration strategies on workers, as well as different recomputation techniques. We prove that this is not a problem, as long as some invariant is ensured.

Distributing search problems on networks mainly relies on paths, that is, an abstract representation of the location of a tree node in the search tree. Only two

extra methods are needed: `fromPath` that builds a tree node given a path, and `getPath` that returns the path associated to a tree node. It is mandatory that paths have the same "meaning" across different workers. In order to formalize this statement, we give the following definition (at first, we do not consider branch & bound):

Definition 1. *Two computation spaces are* equipotent *if and only if they admit the same set of solutions. By extension, two tree nodes are equipotent if and only if their associated spaces are equipotent.*

Parallel search amounts to dispatching subparts of the problem to workers. Soundness (found solutions solve the problem) and completeness (no solution is discarded) follow from the invariant on paths, to be found next. The notation $\texttt{fromPath}_i$ means the method `fromPath` executed on worker w_i.

Invariant 1. *For every pair of workers w_1 and w_2, $\texttt{fromPath}_2 \circ \texttt{getPath}_1$, considered as a binary relation, is a sub-relation of equipotence.*

In simpler words, the computation space at the origin (worker w_1) and the computation space reconstructed at the destination (worker w_2) are equipotent.

Notice that ensuring this invariant does not depend on the exploration strategy. As a consequence, all workers may use different exploration strategies.

In the case of branch & bound, it suffices to modify the definition of equipotence so that we only take into account the best *global* solution. Then, two computation spaces s_1 and s_2 are equipotent under a global solution g if and only if the best solution of $s_1 \cup \{g\}$ is as optimal as the best solution of $s_2 \cup \{g\}$. By lack of space, we omit the details.

5.3 Last Alternative Optimization

When all but one children of a node have been completely explored, and the node contains a copy, this can be handed down to the last child. This technique is known as *last alternative optimization* (LAO), and Schulte [13] presents a formal reasoning why it is important. To support it in an orthogonal, automatic way, we have to change the space node interface and the tree node implementation: space nodes need a special `askLast` method (analogous to `createLastLeftChild` and `createLastRightChild` in [3]) that acquires its parent's copy instead of working space – if available. Tree nodes internally maintain the number of *open children*, subtrees that have not been explored exhaustively yet. Thus `getChildren` can call either `ask` or `askLast`, depending on the parent's number of open children. This scheme makes LAO completely automatic, invisible to both search engines and recomputation strategies. It also fits seamlessly into our branch & bound setup, as pushing a copy over a special node during LAO of course constrains that copy.

We want to keep the invariant that the root node always stores a copy, so LAO must not be applied there.

Some search engines may require a different notion of last alternative, if they can say for sure that they will never visit a certain subtree again. This

can be accomplished by a method `closeSubtree` in the `TreeNode` interface that simply sets the number of open children to 0. Note that even a badly designed search engine cannot break system invariants: With a copy in the root node, recomputation is always guaranteed to terminate, even if some LAO was done prematurely. An interesting extension might be a search engine that can speculatively close a subtree that it will most probably not return to.

6 IOzSeF – A Search Factory for Mozart

A complete implementation of the system described in this paper, the Integrated Oz Search Factory (IOzSeF), is available from the Mogul archive under the URL `mogul:/tack/iozsef`.

IOzSeF is a replacement for both the Explorer and the standard `Search` module. It currently features the following exploration strategies:

– Depth first search
– Breadth first search
– Iterative deepening [8]
– Limited Discrepancy search [5]
– A^* search [10]

The user can chose between no, fixed, adaptive, and full recomputation, last alternative optimization is always done automatically. Branch & bound optimization can be combined with all the other features. The graphical user interface is closely modeled after that of the Explorer; it basically offers the same functionality. A prototype implementation of parallel search supplies evidence that the design carries over to a parallel setting.

Visualization Visualizing the search tree can be an important aid in modeling the problem. It helps to find sources of weak propagation and to match distribution heuristics and exploration strategy.

The space/tree node abstractions already provide a complete tree data structure. It can be refined further to contain all additional attributes necessary for computing a visual layout and displaying the tree. It is straightforward to reuse the Explorer's layout algorithm, which is an incremental version of Kennedy's tree drawing algorithm [7]. This yields a visualization module (similar to the one presented in [2]) that is truly independent of recomputation, branch & bound, and exploration strategy. We provide the tree visualization as an independent Mozart library, the `TkTreeWidget` (`mogul:/tack/TkTreeWidget`).

Implementation The implementation makes heavy use of Mozart's object system, especially of dynamic inheritance: the interfaces are modeled as classes that are combined dynamically at run time. For example, the class `TreeNode` inherits either directly from `SpaceNode` or from `BNBNode`, a class derived from `SpaceNode` that provides the extensions necessary for branch & bound optimization.

Evaluation IOzSeF is competitive with the explorer in terms of speed – for a number of standard examples, it performs sometimes better and never more than thirty percent worse. The overhead is due to the more complex data structures representing nodes, and the need for more method calls between the independent modules – the usual price that is payed for modularity. The implementation was not optimized towards efficiency but towards clean design and extensibility, so there is probably still quite some room for improvements.

The benefit of the principled design and orthogonality of the modules is that IOzSeF delivers correct results also when combining recomputation, branch & bound, and unorthodox exploration strategies (for example arbitrary manual exploration), whereas the Explorer is sometimes unpredictable.

7 Related Work

Schulte explains the need for a search factory in the *Future Work* section of his book "Programming Constraint Services" [13]. This book is the reference for computation spaces, recomputation, exploration strategies and parallel search. However, most of the algorithms presented there assume depth-first, left-to-right exploration, which we do not.

The Explorer [11], Mozart's graphical frontend to search, contains most of the features we present here, but in a monolithic, ad-hoc implementation. Besides, the combination of some features, especially branch & bound and recomputation, is not correct.

Chew et al. introduce STK [2], a SearchToolKit for Oz. Their design features several *dimensions*: memory policy, exploration, interaction, information, visualization, and optimization. They do not elaborate on how memory policy (recomputation) interacts with the other dimensions. Parallel search is not considered. However, their information dimension contains debugging functionality (like information on the choice a distributor made) that should be considered to be integrated into our setup.

Choi et al. [3] present an architecture for implementing state restoration policies in an orthogonal way. They also organize their fundamental data structures in a tree of nodes, making state restoration automatic and invisible to the user. Their version of last alternative optimization requires the search engine to collaborate, it is not automatic. The interactions of branch & bound and LAO with recomputation are only sketched. Choi et al. design their interface such that it supports novel state restoration policies (namely lazy copying, course-grained trailing and batch recomputation). We do not consider these extensions here, because Mozart does not provide for powerful enough primitives to implement them.

ILOG Solver [6] provides an object-oriented interface for programming exploration strategies. State restoration is realized through trailing. The exploration strategy is programmed only indirectly, by supplying a *node evaluator* and a *search selector* that specify which node and which branch to select, respectively (this is discussed in [9]). Although this also abstracts over state restoration, our

interface is more intuitive to use. ILOG Solver does not allow to extend the underlying abstractions.

8 Conclusion and Outlook

Analyzing the main features provided by usual search libraries in constraint programming, we identify orthogonal concepts that are however interleaved in available implementations. Separating these key elements, we design a search factory for Mozart that is based on two abstractions: space nodes, encapsulating recomputation, and tree nodes, providing a clean interface for programming exploration strategies. Then, we show how to implement parallel search, branch & bound, and last alternative optimization by slightly extending the core abstractions. We notice that these extensions are mostly orthogonal and can be easily combined.

As a first example, a new implementation of an exploration strategy immediately benefits from recomputation and visualization, for free. As a second example, the workers used in parallel search may transparently use different exploration strategies, different recomputation techniques, and different visualization modules. Soundness and completeness of the search is preserved.

The search factory not only serves as a proof of concept, but can be used as a replacement for the Explorer and the engines found in the `Search` module.

In a prototype implementation, the search factory has been ported to the Alice programming language [1], using the Gecode constraint library (available at [4], some implementation details can be found in [14]). As a result, the design we develop here maps pretty well to a functional, statically typed, non-object-oriented language (Alice is based on SML). Besides, the tree node layer provides a very clean interface that integrates perfectly within a functional language. The stack of layers starting from the computation space paradigm to the tree node interface allows for a higher-order view of search in constraint programming that reconciles the incompatible natures of logic variables and functional abstractions. New features available in Gecode (like batch recomputation [3]) easily fit into our framework.

Future Work Our future work goes in two directions: on the one hand, the Mozart-based implementation needs thorough testing and optimization for speed and memory requirements, and the parallel search engine should be fully incorporated. This will lead to a true alternative to Mozart's current search engines. On the other hand, the port to Alice/Gecode will be completed, to provide a full featured search environment on this platform. It may be interesting to build an extension similar to the "Information" dimension introduced by Chew et al. [2], which can provide important insight that is needed to debug constraint programs. Another opportunity for improvement consists in finding clear abstractions that describe precisely the chosen recomputation policy. In particular, it remains to design an interface that allows the programmer to easily specify hybrid recomputation strategies as suggested by Choi et al. [3].

Acknowledgements We would like to thank Christian Schulte, who proposed this topic as a student's project to Guido Tack and supervised it. He and Thorsten Brunklaus made helpful comments on a draft version of this paper. Marco Kuhlmann helped in testing and debugging the implementation; he also provided the A^* exploration strategy. We also want to thank the anonymous referees for their constructive suggestions that helped improve the paper.

References

1. The Alice Project. `http://www.ps.uni-sb.de/alice`, 2004. Homepage at the Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany.
2. Tee Yong Chew, Martin Henz, and Ka Boon Ng. A toolkit for constraint-based inference engines. In *Practical Aspects of Declarative Languages, Second International Workshop*, LNCS, Volume 1753, pages 185–199, Boston, MA, January 2000. Springer-Verlag.
3. Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, LNCS, vol. 2239, Paphos, Cyprus, November 2001. Springer Verlag.
4. Gecode, the generic constraint development environment. `http://www.gecode.org`, 2004.
5. William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, pages 607–615, Montréal, Québec, Canada, August 1995.
6. ILOG Inc., Mountain View, CA, USA. *ILOG Solver 5.0 reference Manual*, 2000.
7. A. J. Kennedy. Functional pearls: Drawing trees. *Journal of Functional Programming*, 6(3):527–534, May 1996.
8. Richard E. Korf. Iterative-deepening–an optimal admissible tree search. In Aravind Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 1034–1036, Los Angeles, CA, August 1985. Morgan Kaufmann.
9. Irvin J. Lustig and Jean-François Puget. Program does not equal program: Constraint programming and its relationship to mathematical programming. White paper, ILOG Inc., 1999. Available at `http://www.ilog.com`.
10. Stuart J. Russel and Peter Norvig. *Artificial Intelligence - A Modern Approach - Second Edition*. Prentice Hall, Englewood Cliffs, 2003.
11. 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.
12. Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289. The MIT Press, December 1999.
13. Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artficial Intelligence*. Springer-Verlag, Berlin, Germany, 2002.
14. Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In *Tenth International Conference on Principles and Practice of Constraint Programming*, LNCS, Toronto, Canada, September 2004. Springer-Verlag. To appear.