IOzSeF

Integrated Oz Search Factory

Guido Tack*

Abstract

In this report, **IOzSeF**, a *search factory* for Mozart/Oz, is presented. Search plays an important role in constraint programming. In Mozart/Oz, search is done by library procedures or the Explorer, a graphical search tool. All these have some shortcomings, mainly that features like recomputation, best solution search and visualisation cannot be combined with freely. **IOzSeF** aims at combining the advantages of the currently available tools into one library, enabling the user to build custom-made search engines.

1 Introduction

This project report presents **IOzSeF**, the **I**ntegrated **Oz Se**arch **F**actory. **IOzSeF** is a generic search framework, implemented in Mozart Oz [2], which combines the most important aspects of search in constraint programming systems into a library that is simple to use.

The main goals of the project were the identification and implementation of the orthogonal aspects of search. Some powerful abstractions were found that allow for a "layered" view on the *search tree* (the central metaphor used throughout **IOzSeF**). In particular, the implementation of new search strategies becomes very easy.

1.1 What is a *search factory*?

A search factory is a library which provides a generic interface to search in constraint programming systems. It provides a sort of "construction set", as the user can choose from several options for the orthogonal aspects of search, and let the *factory* build a search engine, tailored to the user's needs.

In addition, the search factory should provide powerful abstractions that make it easy to extend.

1.2 Why?

Currently, Mozart offers several independent search engines: Those in the system module "`Search`" (including parallel versions) and the Explorer (providing interactive search and a visualisation of the search tree).

IOzSeF aims at providing the same functionality as a single library, with the benefit of being able to freely combine the different modules, obtaining a search engine that fits the problem.

*Programming Systems Lab, Saarland University, Saarbrücken, Germany. Email: `tack@ps.uni-sb.de`

1.3 Organisation of this report

The next section gives a short overview on some of the main concepts used in **IOzSeF**. Section 3 deals with the orthogonal aspects of search that were identified, in Section 4 some implementation details are given, and in Section 5 the implementation is evaluated. The last Section (6) gives a conclusion and an outlook on future work.

2 Concepts

This section does not attempt to explain search in constraint programming systems in detail. It only explains the concepts and abstractions that **IOzSeF** makes use of.

2.1 Computation Spaces

In Mozart/Oz, *computation spaces* are used to implement constraint propagation, distribution and search. (See [8] for a detailed discussion.) A computation space is created by applying the system procedure `Space.new` to a unary procedure, the so called *search script*, which contains the problem specification.

Then propagation starts, and when it reaches a fix point, the space becomes *stable*. Now it is the *search engine's* task to react on the space's state, which may be one of `alternatives`, `succeeded` or `failed`. For spaces with alternatives (i.e., for which distribution has to be done), the search engine may *commit* to one of the alternatives (which triggers propagation again) or *clone* the space (this is used for backtracking).

2.2 The Search Tree

The *search tree* is the major concept that underlies **IOzSeF**. It is a labelled, ordered tree with choice points as its inner nodes and solutions or failures as its leaves. (See Figure 1; blue circles represent choice points, green diamonds succeeded leaves, red squares failed leaves. White circles have not yet been explored, their state is undefined.) The nodes are ordered according to the alternative they represent.

During search, the search tree is built while it is being explored by the *search engine*. **IOzSeF** does all the book-keeping involved in building and manipulating the search tree; for the programmer it offers different views on the tree through different levels of abstraction.

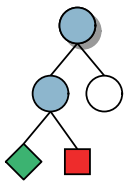


Figure 1: A search tree

3 Orthogonal Aspects Of Search

Several orthogonal aspects can be identified:

3.1 Recomputation

A computation space can consume a lot of memory, so it may be not feasible to store a computation space in every node of the search tree. There are two different solutions to this problem:

Trailing stores some information that can be used to *undo* the changes that were made to a computation space, so that at a later point the original state can be reconstructed.

In Mozart, *recomputation* is used instead: The path from some ancestor (that contains a computation space) to the current node is stored, so that the current node's space can be recomputed without search from that ancestor node.

There are several strategies where to store computation spaces:

- only in the root node (called *full recomputation*)
- in every n-th node (called *fixed recomputation*, n is called *maximal recomputation distance* or *MRD*)
- adaptively, creating a copy on the middle of the path between the node that has to be recomputed and the ancestor it is recomputed from

Thus, recomputation is a means of trading space for time. For a detailed discussion of recomputation (including a comparison to trailing), see [8].

IOzSeF uses recomputation as the glue between the two concepts of computation spaces and the search tree: For the programmer it looks as if each tree node contained a computation space. Internally, recomputation is triggered automatically whenever the space of some tree node is needed.

3.2 Search strategies

Search strategies specify the way the tree is explored. There are a lot of "classical" search strategies, e.g. *depth first search* (DFS), *breadth first search* (BFS), *iterative deepening* (ID, see [5]) or *limited discrepancy search* (LDS, see [3] and [6] for details). **IOzSeF** provides an interface for programming search strategies. An example for a depth first search engine can be found in Section 4.3.

3.3 Best solution search

Best solution search (optimisation) is a very important application of constraint programming. The most prominent variant is *Branch & Bound*, where, after a solution has been found, every node that is explored afterwards is constrained to yield a "better" solution (in terms of some given order). Branch & Bound therefore has the nice invariant that every solution that is found is better than the previous one.

3.4 Visualisation

The *visualisation* of search can be an important aid in designing the problem "script". It helps in finding sources of weak propagation and in matching distribution heuristics and search strategy. The visualisation part of **IOzSeF** is tightly modelled after that of the *Explorer* (see [7]), including incremental layout and interactive exploration. In contrast to the Explorer, visualisation is implemented as a separate module; that way, arbitrary search strategies can be visualised[1].

3.5 Parallel search

Parallel search explores a problem's search tree on several networked computers in parallel. It can speed up search almost linearly, and its implementation is made simple by the powerful distribution concepts of Mozart/Oz (see [8]). Nevertheless, parallel search is beyond the scope of this project, as it would have been a lot of work to define how the other aspects can be kept orthogonal in a parallel setting.

[1] The module is available at [9]; examples are included to illustrate how it can be used to draw arbitrary trees.

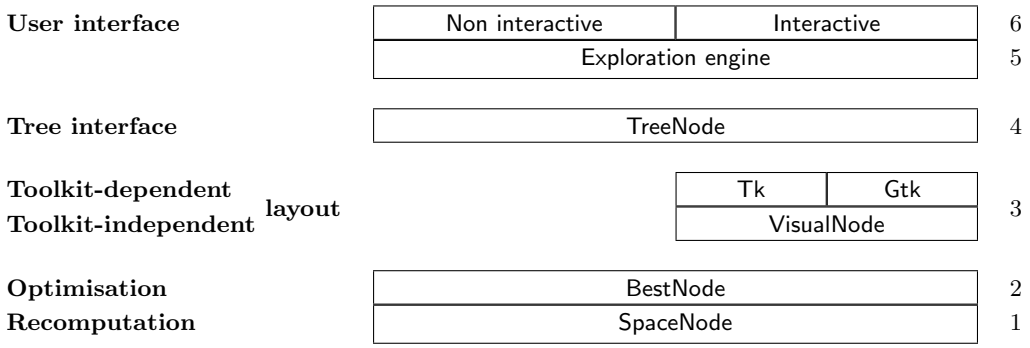


Figure 2: Layers

4 Implementation Details

Figure 2 gives an overview on the layers **IOzSeF** is built of.

In the following sections, layers 1, 4, and 5 are presented in more detail, followed by a short discussion of the visualisation module and some other implementation issues.

4.1 Layer 1 – SpaceNode

4.1.1 Interface

At this layer, the magic happens that hides computation spaces behind tree nodes (with implicit, automatic recomputation). This is achieved as follows:

A *SpaceNode* is an Oz object with "public" methods

- `new(Script)`
  creates a new *SpaceNode* (the root node) from a problem script
- `init(Mom I)`
  creates a new *SpaceNode* as child of node `Mom` with alternative number `I`
- `getSpace($)`
  returns the computation space associated with this node
- `ask($)` corresponding to `{Space.ask S}`
- `merge($)` corresponding to `{Space.merge S}`

So to the outside, *SpaceNodes* look very much like a wrapper class for Mozart's computation spaces.

In addition, a *SpaceNode* has some "private" attributes/methods:

- `attr space`
  Holds the current *working space* (see below), if the SpaceNode has one
- `attr copy`
  May hold a copy of this node's space
- `attr commits`
  Contains the number of the alternative that this node represents

- `meth aquireSpace($)`
  Tries to get a working space from the node's ancestor
- `meth donateSpace($)`
  "donates", if possible, this node's working space to one of its children
- `meth recompute($)`
  recomputes a working space for this node (recursively)

The working space is the computation space that is used for the operations `getSpace($)`, and via that for `ask($)` and `merge($)`. It is passed on to the first child node who requests it via `donateSpace`.

The way `getSpace` works is illustrated in Figure 3.

It is essential that the acquisition of a computation space is delayed until `getSpace` is invoked. That way, all the children of a node can be created (e.g. for breadth first search), but only when one of them is being explored, their corresponding computation spaces have to be created. The expensive recomputation is therefore only done when it is really necessary.

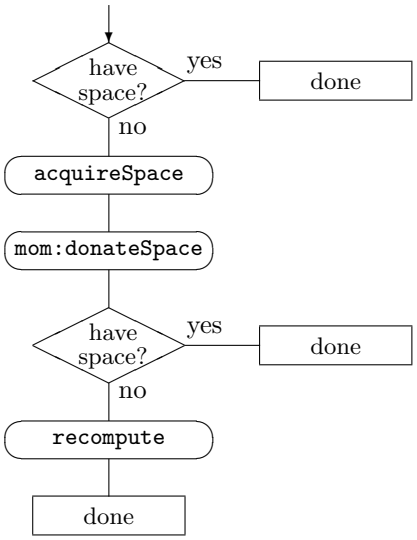


Figure 3: Acquisition of computation spaces

4.1.2 Implementation of recomputation

Figure 3 explains *when* recomputation has to be done. So *how* is it done?

On creation of a new node, the number of the alternative it represents is stored. When `recompute($)` is invoked, the following happens:

- If the current node contains a copy, just clone it and return the clone.
- Otherwise, invoke `recompute($)` recursively for the node's parent (yielding the parent node's computation space), and commit to the current node's alternative.

For a simple recomputation method, see Figure 4.

Of course, the copies have to be created at some time (in particular, the root node always has to contain a copy). This is done according to the recomputation strategy that was chosen:

```
meth recompute(?C)
   case @copy of
      nil then
      {self.mom recompute(C)}
      {self commit(C @alternative)}
   else
      C = {Space.clone @copy}
   end
end
```

Figure 4: A simple method for recomputation

- For full recomputation, simply store a copy in the root node.
- Fixed recomputation requires to store a copy in every n-th node. This can be done easily by incrementing a counter on node creation, and when it gets larger than the maximum recomputation distance, create a copy as soon as `acquireSpace` is called (remember that the node's space is not created on node creation but on the first invocation of `acquireSpace`).
- Adaptive recomputation (as introduced in [8]) requires the recomputation procedure to be modified: It has to create a copy in the middle of the path from the current node to the node it is recomputed from.

4.2 Layer 4 – The Tree Interface

This layer abstracts away from the "low level" *asking* and explicitly creating children. It offers the following interface:

- `meth initScript(Script)`
  Creates the root node from a problem script
- `meth getChild(I $)`
  Returns child number `I`
- `meth noOfChildren($)`
  Returns the number of children

The method `noOfChildren($)` is the most interesting one: When it is invoked for a node V, the state of V has to be determined. Therefore the invocation of `noOfChildren($)` is the point where recomputation may be triggered, where a solution may be found and where optimisation may have to take place.

Furthermore, each node has to keep track of its children such that `getChild(I $)` creates a child only on its first invocation; afterwards it should just return the child.

The collection of solutions is also done by the *TreeNodes* with the help of a *Collector* module, resulting in a generic way to access the solutions after the search has been stopped or completed.

4.3 Layer 5 – Implementing Search Strategies

Now it becomes easy to implement new search strategies: With the abstractions from layer 4, one just has to deal with nodes and their numbers of children. Figure 5 shows an example of a realistic implementation of depth first search. A class that models a search strategy has to provide the method `search(Node)`, which is called to explore the subtree of `Node`. In this example, the

```
class DFSClass
   from SearchClass

   meth DFSKids(M N Node)
      if M>N then skip
      else
         Kid = {Node getChild(M $)}
      in
         DFSClass, search(Kid)
         DFSClass, DFSKids(M+1 N Node)
      end
   end

   meth search(Node)
      case {Node noOfChildren($)} of
         0 then skip
      [] N then
         DFSClass, DFSKids(1 N Node)
      end
   end
end
```

Figure 5: A depth first search engine

number of `Node`'s children is determined, and then `search` is invoked recursively for each of the children.

As an extension to the interface mentioned above, *binarisation* (mapping an n-ary tree to a binary one) can be done implicitly. That makes the implementation of e.g. *limited discrepancy search* a bit more "natural". (Binarisation is also described in detail in [8], chapter 5.3.)

Search strategies can be implemented in a way that reflects that they are based on *phases*; this again simplifies the formulation of some strategies (e.g., iterative deepening extends the tree one level per phase). At the moment, only "one level" of phase exploration can be done: A phase cannot itself be computed in several phases. It might be interesting to extend this to at least two levels, because then a well-known optimisation technique called *iterative best solution search* could be defined in a very natural way.

Examples for all the search strategies mentioned above can be found in the file `Search/searchengines.oz`, which is part of the **IOzSeF** package.

4.4 Visualisation

The layout algorithms are very similar to those of the *Explorer*, which are an incremental version of the algorithm presented by Kennedy in [4].

The layout and drawing routines have been implemented as a separate Oz module which provides a generic tree drawing widget (see [9] for documentation).

4.5 Miscellaneous

In this section, some more implementation issues (and small problems that arose) are discussed.

4.5.1 Branch and bound

For Branch & Bound, it is very important that the book-keeping is implemented properly. This is not a trivial thing, as it has to be made independent of the search strategy (especially because **IOzSeF** allows for interactive exploration).

Every time a solution is found, it is remembered as the currently best solution (this is the invariant guaranteed by Branch & Bound). On creation of a new node, a reference to the currently best solution is stored inside the new node, so that its space can be constrained accordingly (this may have to be done several times because of recomputation). As an optimisation, the currently best solution has to be stored in a node only if it is different from the best solution stored in its parent node (one can easily see that this preserves all invariants).

As **IOzSeF** allows for arbitrary search strategies, the currently best solution is not necessarily the right-most solution. To make the best solution easier to find, **IOzSeF** displays it as a violet diamond (instead of the green diamond for non-optimal solutions).

As mentioned above, **IOzSeF** handles Branch & Bound correctly no matter what search strategy is applied. This is an important advantage over the Explorer: If you use the latter to manually explore the search tree, no pruning is done, so that the important invariant mentioned above does not hold any longer.

4.5.2 Last alternative optimisation (*LAO*)

LAO is an important and well-known technique used to reduce running time. It means that if a node contains a copy of a space, and search in all of its subtrees except one has been completed, then the copy can be given down to the last child to be explored.

This involves book-keeping of the "open" children of a node (those whose subtrees have not yet been completed).

4.5.3 Suspension

Suspension means that a space cannot reach a stable state because one of its propagator threads suspends on a logic variable. This is usually an error in the problem script, so it is important to be able to locate where exactly suspension happens.

In contrast to the Explorer, **IOzSeF** is able to deal with arbitrarily many suspended nodes and can display all of them at once. The Explorer displays only one suspended node at a time; it then freezes, so that the user cannot get information on any node any longer. In **IOzSeF** instead, exploration can continue if there are other nodes that have not yet been explored (the suspended node is treated as a leaf node). That way, all suspended nodes can be found. Of course exploration can go on if a node eventually gets unsuspended. Thus, **IOzSeF** must keep track of suspended nodes and provide a thread-safe way of concurrently changing the tree structure when a node gets unsuspended. This is especially important because the lazy linking feature of Mozart sometimes generates suspended root nodes. These become unsuspended just a moment later, but that can be enough if **IOzSeF** already tried to explore that node.

Branch & Bound as well as *LAO* require a special treatment of suspended nodes, as both need to manipulate a node's space (or its copy) when the node is created. This may only be done if the node is not in a suspended state, and it has to be done in case the node gets eventually unsuspended.

5 Evaluation

5.1 Correctness and adaptability

IOzSeF has been tested with some problem scripts, including optimisation problems. It works as one would expect, delivering the same solutions and drawing the same trees as the standard search tools.

Several search strategies have been implemented (depth first, breadth first, iterative deepening and limited discrepancy), which proves that the abstractions are powerful enough.

Combinations of Branch & Bound and different search strategies work well, no matter what recomputation method is used, and even in the presence of suspension. That means that all features of **IOzSeF** are completely independent of the search strategy. This should be noted explicitly, because most implementations of search only allow for a depth-first left-most strategy.

5.2 Performance benchmarks

To evaluate the design in terms of performance, several well-known example problem scripts have been run:

10-Queens, 10-S-Queens: The famous n-Queens puzzle, in a naive and a smart version

n-(S)-Magic: A magic-sequence puzzle of length n, also in a naive and a smart version

Alpha: A cryptoarithmetic puzzle

Bridge: A well-known scheduling example

Photo: Aligning for a photo

18-Knights: Finding a certain sequence of knight-moves on a 18×18 chess board

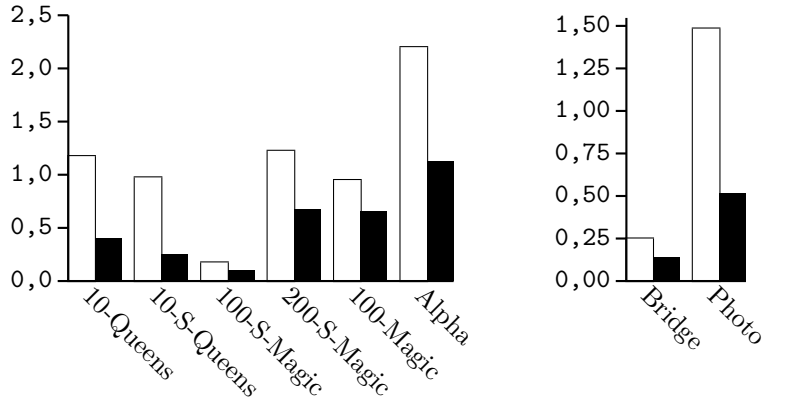
These example programs are the same as in [8], Appendix A.

Figure 6 shows a comparison of **IOzSeF** with the standard library search engines (the numbers represent the run times in seconds, **IOzSeF** was used without visualisation). One can see that **IOzSeF** doesn't perform as well as the library search engines, but that is no big surprise: **IOzSeF** has to build the whole search tree, consisting of Mozart objects, which is far more expensive than dealing directly with computation spaces (as the library engines do). That directly corresponds to the fact that **IOzSeF** performs better on problems which involve a lot of propagation and less search (such as 200-S-Magic, 100-Magic and Alpha).

A more interesting comparison is shown in Table 1. Here the runtime is shown as a factor of the time that the library search engine needs for the problem. The run times of both the Explorer and **IOzSeF** are the *search* times; they do not include layout and drawing of the tree. The table shows that **IOzSeF** performs quite good, under some circumstances even better than the Explorer.

6 Conclusion And Outlook

This report presented a *search factory* for Oz, its essential parts and some issues about their implementation. **IOzSeF** provides a user interface through library routines and a graphical front-end similar to that of the Explorer, enabling the user to freely combine the orthogonal parts to the search engine he needs. In addition, it supplies the programmer with an interface that makes it easy to implement new search strategies.



(a) All solution search (b) Best solution search

Figure 6: Comparison to standard library search engines
(White bars represent library search engine times, black bars **IOzSeF** times)

Problem	*No. of solutions*	*Lib5*	*Exp5*	**IOzSeF***5*	**IOzSeF***a*
18 Knights	one	1.0	0.83	0.70	0.63
Photo	best	1.0	1.14	1.73	1.78
Bridge	best	1.0	1.40	1.75	1.86
MT10	best	1.0	0.98	1.11	1.23
Alpha	all	1.0	1.08	1.29	1.38
Magic-S-200	all	1.0	1.20	1.25	1.03
Queens-S-10	all	1.0	1.41	1.95	2.04

Lib5 ≙ Library search engine with MRD=5
Exp5 ≙ Explorer, fixed recomputation with MRD=5
IOzSeF*5* ≙ **IOzSeF**, fixed recomputation with MRD=5
IOzSeF*a* ≙ **IOzSeF**, adaptive recomputation

Table 1: Comparison of Library Search Engine, Explorer and **IOzSeF**

All this is achieved by abstraction from the difficult concepts: **IOzSeF** features implicit and automatic recomputation, a correct implementation of *Branch & Bound* and an independent visualisation module.

The implementation shows that the parts that had been identified to be orthogonal really *can* be implemented orthogonally. As a consequence, the concept of *computation spaces* and *copying-based search* proves to be quite a powerful one, and a strong basis for data structures like **IOzSeF**'s search tree.

The performance of the implementation is already competitive for real life problems. There should be room for optimisations, though, as speed was not the primary design goal.

As already mentioned above, parallel search could not be investigated in this project. This could be an interesting and promising point to start from for future work; at the moment, it is not clear whether the other parts can still be kept orthogonal in a parallel environment.

Another part that needs further development is the graphical user interface. The current implementation uses Tk as the underlying graphics library and is tuned for generality rather than speed. Thus, a port to the Gtk graphics library seems promising (Gtk is supposed to deliver a much better performance than Tk), and first steps are done. Apart from that, some interaction is still missing, like inspecting a node's state or user-defined action and comparison procedures.

On the low level, support for *iterative best solution search*, another method of optimisation, could be interesting, as well as some more implementations of search strategies.

The current implementation can be obtained from the *Mogul* (Mozart Global User Library) [1], the package is made available under `mogul:/tack/iozsef` and can be installed with the `ozmake` tool.

I hope that the remaining problems can be solved so that **IOzSeF** may replace the Explorer in the near future.

Acknowledgements

Christian Schulte proposed **IOzSeF** to me as a "Fortgeschrittenenpraktikum" here at the Programming Systems Lab of Saarland University. It was carried out in winter/spring 2001/2002.

I would like to thank him for proposing this interesting project, for many conversations that gave me the necessary insight as well as overview. I also would like to thank Denys Duchier and Joachim Niehren for their lecture on Oz (without which I never would have understood that language), and Thorsten Brunklaus and Leif Kornstaedt for help with Oz, Tk and Gtk; finally the whole staff of the Programming Systems Lab for the friendly atmosphere.

References

[1] MOGUL, The Mozart Global User Library, `http://www.mozart-oz.org/mogul`.

[2] The Mozart programming system, `http://www.mozart-oz.org`.

[3] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, pages 607–615, Montréal, Québec, Canada, August 20-25 1995. Morgan Kaufmann, 1995.

[4] A. J. Kennedy. Functional pearls: Drawing trees. *Journal of Functional Programming*, 6(3):527–534, May 1996.

[5] 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.

[6] Richard E. Korf. Improved limited discrepancy search. In *AAAI/IAAI, Vol. 1*, pages 286–291, 1996.

[7] 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.

[8] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artficial Intelligence*. Springer-Verlag, Berlin, Germany, 2002.

[9] Guido Tack, TkTreeWidget, `mogul:/tack/TkTreeWidget`.