Oz Explorer: A Visual Constraint Programming Tool

Christian Schulte
Programming Systems Lab, German Research Center for AI (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
schulte@dfki.uni-sb.de

Abstract

This paper describes the Oz Explorer and its implementation. The Explorer is a visual constraint programming tool intended to support the development of constraint programs. It uses the search tree of a constraint problem as its central metaphor. Exploration and visualization of the search tree are user-driven and interactive. The constraints of any node in the tree are available first-class: predefined or user-defined procedures can be used to display or analyze them. The Explorer is a fast and memory efficient tool intended for the development of real-world constraint programs.

The Explorer is implemented in Oz using first-class computation spaces. There is no fixed search strategy in Oz. Instead, first-class computation spaces allow to program search engines. The Explorer is one particular example of a user-guided search engine. The use of recomputation to trade space for time makes it possible to solve large real-world problems, which would use too much memory otherwise.

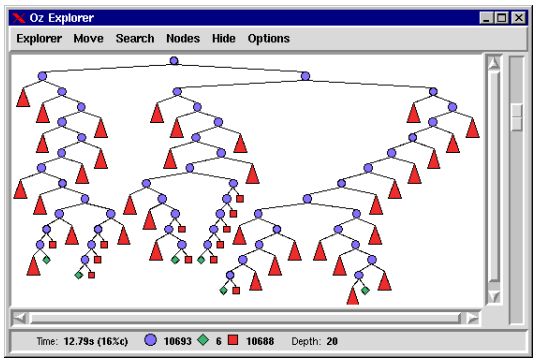
1 Introduction

In the last decade constraint programming, especially with finite domain constraints, has become popular in many application areas. To a large extent, the growing interest is caused by both the expressive power and the availability of efficient implementations like CHIP [5], ECLiPSe [1], ILOG Solver [8], clp(FD) [3], just to name a few.

Development of applications based on constraint programming proceeds in two steps. The first step is to design a principally working solution. This is followed by the much harder task to make this solution work for problems of real-world size. The latter task usually involves a high amount of experimentation to gain additional insights into the structure of the problem. Meier reports in [12] that a large part of the development process is spent on performance debugging. Therefore it is surprising that existing systems offer little support for the development of constraint programming applications (with the recent exception of [12]).

Appears in: Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, Leuven, Belgium, pages 286–300. The MIT Press, July 1997.

This paper describes the visual constraint programming tool Oz Explorer for the language Oz. Oz is a concurrent constraint language providing for functional, object-oriented, and constraint programming [18, 7]. It has a simple yet powerful computation model [17] which extends the concurrent constraint model [15] by first-class procedures, concurrent state, and encapsulated search.



The Explorer uses the search tree as its central metaphor. The user can interactively explore the search tree which is visualized as it is explored. Visible nodes carry information on the corresponding constraints that can be accessed interactively by predefined or user-defined procedures. The Explorer can be used with any search problem, no annotations or modifications are required. In particular, the Explorer does not rely on a fixed constraint system.

First insights into the structure of the problem can be gained from the visualization of the search tree. How are solutions distributed? How many solutions do exist? How large are the parts of the tree explored before finding a solution? The insights can be deepened by displaying the constraints of nodes in the search tree. Is constraint propagation sufficient? Does the heuristic suggest the right choices? Interactive exploration allows following promising paths in the search tree without exploring irrelevant parts of it. This supports the design of heuristics and search engines.

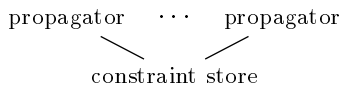
Complex real world problems require a tool to be practical with respect to both efficiency and display economy. The amount of information displayed by the Explorer is variable: the search tree can be scaled and subtrees can be hidden. In particular, all subtrees that do not contain solutions can be hidden automatically.

One reason that there are only so few tools for the development of constraint programs is that controlling search is hard in existing systems. Systems like those mentioned earlier provide for a small set of search strategies. In contrast to that, search engines like single, all, and best solution search are not built-in into Oz, but are programmed using first-class computation spaces. To deal with problems which would use too much memory otherwise, recomputation trading space for time can be programmed with first-class computation spaces.

The Explorer is one particular example of a user-guided interactive search engine that would not have been possible without first-class computation spaces. The paper shows how the Explorer is implemented on a high level in Oz using first-class computation spaces and how recomputation is used in its implementation. Figures are presented that support the practicality of the implementation as it comes to efficiency and memory usage.

Plan of the paper. Section 2 introduces some basic ideas of constraint programming in Oz, followed by an example of a constraint program where the Explorer is used in its development. First-class computation spaces are introduced in Section 4. The most important features of the Explorer are given in Section 5. Section 6 gives an overview of the Explorer's implementation, followed by a discussion of the implementation's core and recomputation in Sections 7 and 8. The practicality of the Explorer is discussed in Section 9 and the last section gives some related work.

2 Constraints, Propagators, and Search



Central to constraint programming in Oz is the notion of a computation space. A *computation space* consists of propagators connected to a constraint store.
The *constraint store* stores information about values of variables expressed by a conjunction of basic constraints. *Basic constraints* are logic formulae interpreted in a fixed first-order structure. In the following we restrict ourselves to finite domain constraints. A basic finite domain constraint has the form $x \in D$ where D is a finite subset of the positive integers. Other basic constraints which make sense here are $x = y$ and $x = n$, where y is a variable and n a positive integer.

To keep operations on basic constraints efficient, more expressive constraints, called *nonbasic*, e.g., $x + y = z$, are not written to the constraint store. A nonbasic constraint is imposed by a propagator. A *propagator* is a concurrent computational agent that tries to amplify the store by *constraint propagation*: Suppose a constraint store hosting the constraint C and a propagator imposing the constraint P. The propagator can *tell* a basic constraint B to the store, if $C \wedge P$ entails B and B adds new and consistent information to C. Telling a basic constraint B updates the store to host $C \wedge B$.

A propagator imposing P disappears as soon as it detects that P is entailed by the store's constraint. A propagator imposing P becomes *failed* if it detects that P is inconsistent with the constraint hosted by the store. A space S is *stable*, if no further constraint propagation in S is possible. A stable space that contains a failed propagator is *failed*. A stable space not containing a propagator is *solved*.

Usually, constraint propagation alone does not suffice to solve a constraint problem: A space may become stable but neither solved nor failed. Hence, we need distribution. *Distributing* a space S with respect to a constraint D yields two spaces: One is obtained by adding D to S and the other is obtained by adding $\neg D$ to S. The constraint D will be chosen such that adding of D ($\neg D$) enables further constraint propagation.

To solve a constraint problem, a space containing basic constraints and propagators of the problem is created. Then constraint propagation takes

place until the space becomes stable. If the space is failed or solved, we are done. Otherwise, we select a constraint D with which we distribute the space. A possible distribution strategy for finite domain constraint problems is: Select a variable x which has more than one possible value left and an integer n from these values and then distribute the space with $x = n$.

Iterating constraint propagation and distribution as sketched above leads to a tree of computation spaces ("search tree") where leaves are failed or solved spaces. To these nodes of the search tree we refer to as solved or failed nodes, to the remaining as *choice* nodes. A problem solver in Oz is factored into two orthogonal components: the problem and the search engine exploring the problem's search tree. One particular example of a hand-guided search engine is the Explorer.

3 Example: Aligning for a Photo

This section introduces the Oz Explorer by means of an example. Five people want to make a group photo. Each person can give preferences next to whom he or she wants to be placed on the photo. The problem to be solved is to find a placement that satisfies as many preferences as possible.

Figure 1 shows an Oz program that solves this problem. The problem is stated as unary procedure `Photo`, where its argument `Sol` is constrained to the problem's solution. The record `Pos` maps the person's name to a position, that is, an integer between 1 and 5. All fields of `Pos` are enforced to be distinct by the propagator `FD.distinct`. The list of preferences is mapped to a list `Ful` of finite domain variables between `0` and `1`, such that each of its elements is either `1` in case the preference can be fulfilled or `0` otherwise. The overall satisfaction `Sat` is given by the sum of all elements of `Ful`. The positions `Pos` are distributed (by `{FD.distribute naive Pos}`) following the strategy sketched in the previous section, where variables are selected in alphabetical order of their respective fields, and where the smallest value is selected first.

So-called reified propagators are used to map preferences to finite domain variables. A reified propagator employs a boolean control variable b. If the propagator is entailed (disentailed), then b is constrained to `1` (`0`). If b is `1` (`0`), the constraint of the reified propagator is enforced (its negation is enforced). The reified propagator `Pos.A+1=:Pos.B` (`Pos.A-1=:Pos.B`) expresses that `A` is placed to the left (right) of `B`. Thus, the control variable of the reified propagator stating that the sum of both is `1`, yields `1` if `A` and `B` are placed next to each other, and `0` otherwise.

We use the Explorer to search for an optimal solution to the `Photo` problem. The optimality criterium is described by a binary procedure stating that the satisfaction must increase with the solutions found:

```
{Explorer solver(Photo proc {$ Old New} Old.sat <: New.sat end)}
```

```
Names = [alice bert chris deb evan]
Prefs = [alice#chris bert#evan chris#deb chris#evan
         deb#alice deb#evan evan#alice evan#bert]
proc {Photo Sol}
   Pos = {FD.record pos Names 1#{Length Names}}
   Ful = {Map Prefs
          fun {$ A#B} (Pos.A+1=:Pos.B)+(Pos.A-1=:Pos.B)=:1 end}
   Sat = {FD.int 0#{Length Prefs}}
in
   {FD.distinct Pos} {FD.sum Ful ~=:~ Sat}
   Sol = sol(pos:Pos ful:Ful sat:Sat)
   {FD.distribute naive Pos}
end
```

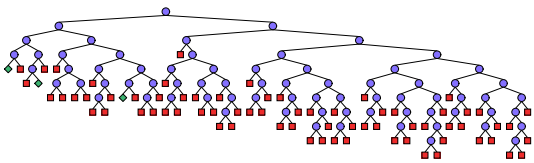
Figure 1: Oz program to solve the photo alignment problem.

The Explorer shows a single choice node (drawn as a circle). Prompting for the next solution explores and draws the search tree up to the first solution as shown to the right. The solution node is diamond-shaped. Exploring and drawing the search tree can be stopped at any time and resumed later at any node in the tree. This is important for problems which have large or even infinite subtrees in its search tree.

Double-clicking the solved node displays the constraints of the succeeded computation space using the Oz Browser (a concurrent tool to visualize basic constraints). The first solution is as follows:

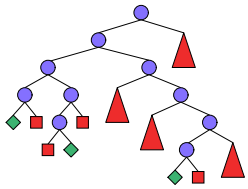
```
sol(pos: pos(alice:1 bert:2 chris:3 deb:4 evan:5)
    ful: [0 0 1 0 0 1 0 0]
    sat: 2)
```

Understanding textual output can be difficult. Therefore, the Explorer can employ user-defined display procedures. Suppose a procedure `DrawPhoto` that displays constraints graphically. The Explorer is configured such that double-clicking a node applies `DrawPhoto` to the node's constraints by `{Explorer add(information DrawPhoto)}`. The figure above shows a particular instance of graphical output for the previously found solution. An arrow between names shows a fulfilled preference, whereas the circled number above a name yields the number of non-fulfilled preferences of that person.



Invoking search for all solutions yields the search tree shown to the right. The optimal solution is the rightmost solved node. Squares represent failed nodes.

Although we are solving a simple problem, it is hard to find solutions and paths leading to them. The Explorer provides support to hide all subtrees which contain only failed leaves by drawing these subtrees as triangles. After applying this functionality, the search tree looks as shown on the right.



By double-clicking the rightmost solution (the Explorer assists in finding certain nodes by providing functionality to move a cursor to it) we get the optimal solution as shown above.

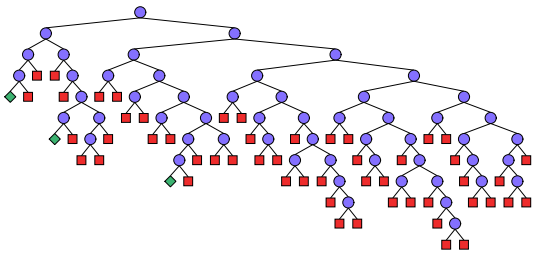
The Explorer reports in its status bar that the entire search tree has 72 choice, 3 solution, and 70 failed nodes. The tree indicates by the length of paths leading to failed leaves that the choices do not result in much constraint propagation. A better distribution heuristic should lead to more constraint propagation. The amount of constraint propagation depends on how many propagators are triggered to amplify the constraint store. So it would be better to assign a value to a variable on which many propagators depend.

This is done by replacing the distribution strategy in the program shown in Figure 1 by a strategy implementing our idea from above:

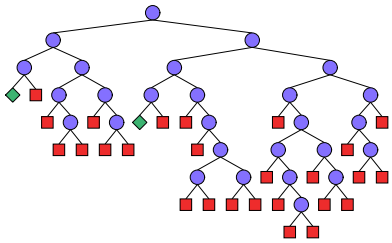
```
{FD.distribute generic(order:nbSusps) Pos}
```

Oz provides predefined distribution strategies to express some common heuristics, but any distribution strategy can be programmed (see Section 4).

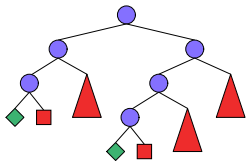
The Explorer is applied to the modified problem to study the impact of our new heuristic on the search tree. The resulting tree is shown on the right. The Explorer's status bar displays that the tree now has 54 choice nodes, 3 solution nodes, and 52 failed nodes, that is, the number of nodes has decreased by about 25%. From the displayed search tree we can conclude that it is much harder to prove optimality of the last solution than to actually find it.



If we use the Explorer to access constraints of nodes in the right part of the tree, we find that search is aiming at solutions symmetrical (i.e., with people placed in reverse order) to those in the tree's left part. The search tree can be reduced in size by removing these symmetries. Some of them can be removed by placing two persons, say the first and the second in the list of persons, in a fixed order. Hence, we add the following constraint to our program: `Pos.{Nth Names 1} >: Pos.{Nth Names 2}`



Applying the Explorer to the new problem and searching for all solutions draws the search tree as shown above. The tree now has only 27 choice nodes, 2 solution nodes, and 26 failure nodes. Thus, removing just these symmetries reduces the number of nodes by 50%. On the right the tree after hiding all failed subtrees is displayed.



4 First-Class Computation Spaces

Section 2 shows that computation spaces are central to constraint programming in Oz. This is reflected by the fact that computation spaces are first-class entities, and that search engines are built from operations on them. They can be passed as arguments of procedures, can be tested for equality and the like. They are abstract values, similar to procedures and cells in Oz (more information on this can be found in [18]). They can be created, their status can be asked for, they can be copied, merged with other computation spaces and additional constraints can be injected into them.

Besides of the constraint store and propagators, a computation space also hosts threads. Like propagators, threads are concurrent computational entities. A *thread* is a stack of statements. It runs by reducing its topmost statement, possibly replacing the reduced statement with new statements. Threads synchronize on their topmost statement: if the topmost statement cannot be reduced, the entire thread cannot be reduced; we say it *suspends*. Statements include procedure application, procedure definition, variable declaration, sequential composition of statements, tell statements, conditional statements, thread creation statements, propagator creation, and so-called choices. A *choice* is either unary (**`choice`** S **`end`**) or binary (**`choice`** S_1 `[]` S_2 **`end`**) where S, S_1, and S_2 are statements called *alternatives*. A thread that contains a choice as its topmost statement suspends.

Computation in Oz usually is carried out in the so-called toplevel computation space. Computations involving constraint propagation and distribution are speculative. They are encapsulated in a first-class computation space. First-class computation spaces lead to a tree of computation spaces. Telling a basic constraint amounts to telling the constraint also in all spaces below. By this, the tree of computation spaces maintains the invariant that all constraints in a parent space are visible in the spaces below.

Reduction of the statement `S={NewSpace P}` creates a new computation space, where `P` is a unary procedure and `S` yields a reference to the newly created space. The newly created space inherits its constraint store from its parent space. It features a single fresh variable, the so-called *root variable*. In `S` a thread is created that contains as single statement the application of `P` to the root variable. Typically, the procedure `P` is the problem to be solved, and its single argument gives access to the solution of the problem (see the procedure `Photo` in Section 3). Running the newly created thread

```
proc {DS Xs}
   choice
      case {SelectVar Xs} of nil then skip
      [] [X] then N={SelectVal X} in
         choice X=N {DS Xs} [] X\=:N {DS Xs} end
      end
   end
end
```

Figure 2: Programming a distribution strategy with choices.

might create variables, constraints, propagators, and further threads.

A computation space `S` is *stable* if no thread and no propagator in `S` can reduce, and cannot become reducible by basic constraints told in a space above. A computation space `S` is *failed* if it contains a failed propagator. When a computation space becomes stable and contains a thread with a unary choice as its topmost statement, the unary choice is replaced by its alternative. This means that the alternative of a unary choice synchronizes on stability. A stable computation space not containing threads with unary choices but with binary choices as their first statements is called *distributable*. When a space becomes distributable one thread containing a binary choice as its topmost statement is selected. A stable space is *succeeded*, if it does not contain threads which suspend on choices.

A computation space `S` can be asked by `A={Ask S}` for its status. As soon as `S` becomes stable, the variable `A` gets bound. If `S` is failed then `A` is bound to the atom `failed`. If `S` is distributable, `A` is bound to `alternatives`. Otherwise, `A` is bound to `succeeded`.

How to program a distribution strategy like that mentioned in Section 2 from choices is shown in Figure 2. The procedure `DS` takes a list of finite domain variables. After synchronizing with the unary choice on stability, `SelectVar` selects a variable if possible and then creates a binary choice.

A distributable space `S` allows to commit to one of the alternatives of the selected choice. By `{Commit S I}` the space `S` commits to the `I`-th alternative of the selected choice. That means that the choice is replaced by its `I`-th alternative. To explore both alternatives of a selected choice, stable computation spaces can be cloned. Reduction of `C={Clone S}` creates a new computation space `C` which is a copy of the space `S`.

The constraints of a local computation space `S` can be accessed by the procedure `Merge`. Reduction of `X={Merge S}` merges a stable computation space `S` with the current computation space and binds the variable `X` to the root variable of `S`.

A procedure `DF` which takes a space as argument and tries to solve it following a depth-first strategy is shown in Figure 3. If no solution is found, but search terminates, the empty list `nil` is returned. Otherwise the procedure returns the singleton list `[T]` where `T` is a succeeded computation space. For example, solving the `Photo` problem from Section 3 and displaying its

```
fun {DF S}
   case {Ask S}
   of failed then nil
   [] succeeded then [S]
   [] alternatives then C={Clone S} in
      {Commit S 1}
      case {DF S} of nil then {Commit C 2} {DF C}
      [] [T] then [T]
      end
   end
end
```

Figure 3: Depth-first exploration.

solution can be done by:

```
case {DF {NewSpace Photo}}
of nil then {Browse ´No solution´}
[] [S] then Root={Merge S} in {Browse Root}
end
```

For implementing branch-and-bound search to find a best solution a further primitive is needed. After having found a solution, the constraints in all remaining computation spaces need to be strengthened such that they can only yield a better solution. For this the operation `{Inject S P}` is provided, it applies the unary procedure `P` to the root variable of the space `S`.

The presentation here has been simplified in that we only consider binary choices. Oz in fact provides also for non-binary choices, this requires the operations `Ask` and `Commit` to be enhanced in a straightforward manner.

Previously we developed the so-called search combinator [16]. It spawns a local computation space and resolves remaining choices by returning them as procedures. First-class computation spaces are more expressive, the search combinator can be expressed by combining the operations `Ask`, `Clone`, and `Commit`. Having only a combination of these operations available turned out to be too limited for some inference engines. The Explorer is one particular example where the expressiveness of the search combinator turned out to be inadequate: Section 8 shows how the Explorer employs recomputation which would not have been possible with the search combinator.

5 The Explorer

Direct use and manipulation. The Explorer is provided as an object in Oz. It can be invoked by applying the object to a message containing the problem to be solved. Its usage does not require any modification of the problem. To search for a best solution, an order implemented as a binary procedure must be provided as an additional argument. After having applied the Explorer to the problem, all actions can be invoked by mouse-clicking, menu-selection, or keyboard accelerators. Since the Explorer is provided as

an object, creating new instances of the Explorer is possible by creating new object instances.

Interactive and incremental exploration. Search can be used in an interactive fashion: the user can explore any part of the search tree in a step-by-step manner. Thus, promising paths in the search tree can be followed without being forced to follow a predefined strategy. Furthermore, depth-first exploration of the search tree for one solution or for all solutions is supported. The Explorer is fully incremental: exploration of the search tree can be stopped at any time and can be resumed at any node.

Ergonomic visualization. After creation of the search tree, the Explorer computes a layout for the newly created part of the search tree and updates the drawing of the tree. The drawn tree can be scaled by direct manipulation of a scale bar. Any subtree of the search tree can be hidden by replacing it with a small triangle. Special support is provided to hide subtrees which contain only failed leaves. By visualizing the search tree one can gain insights into the search process. How are the solutions distributed? Is a first solution found without too many failed nodes? Is it hard to prove optimality of the last solution found? The possibility of hiding failed parts of the search tree assists finding relevant paths leading to solutions.

User-defined access to constraints. All but the failed nodes carry as information their computation spaces. Each node's space can be displayed with user-defined or predefined display procedures. It is also possible to compare the spaces attached to any two nodes, which assists in understanding what is the difference in the constraints between two nodes.

Statistics support. Besides brief statistical information the Explorer provides in a status bar, it is possible to display statistical information for each subtree. User-defined procedures can be used to process and display the statistical information. For instance, a bar chart showing how many failures occur between solutions can help to understand how hard it is to prove optimality in best solution search.

6 Implementing the Explorer

The implementation of the Explorer is divided into three parts:

1. The search engine that constructs and maintains the search tree.

2. Laying out and drawing the tree. That includes support for scaling the tree, as well as support for hiding and unhiding of subtrees.

3. The user interface which provides for menus, cursor control, graphical dialogs for options and the status bar.

The user interface controls the invocation of operations on the search tree. The search engine manipulates a search tree that is implemented as a tree of objects. Each node is an object which stores a first-class computation

```
fun {CreateNode S}
   case {Ask S}
   of failed       then {New FailedNode init}
   [] succeeded    then {New SolvedNode init(S)}
   [] alternatives then {New ChoiceNode init(S)}
   end
end
```

```
class FailedNode
   meth init
      skip
   end
end

class SolvedNode
   attr space
   meth init(S)
      space <- S
   end
end
```

```
class ChoiceNode
   attr space kids
   meth init(S)
      space <- S
   end
   meth step
      LC={Clone @space} RC={Clone @space}
   in
      {Commit LC 1} {Commit RC 2}
      kids <- [{CreateNode LC} {CreateNode RC}]
   end
end
```

Figure 4: Search tree creation.

space in its state. The class the object is created from depends on the space to be stored, i.e., whether the space is failed, succeeded, or distributable. Each of these classes is created by multiple inheritance from classes which provide for the different kinds of methods needed. Invoking an operation at the user interface sends a message to the object and leads to execution of the corresponding method.

The methods for computing the layout use an incremental version of the algorithm presented in [10]. The graphical part of the user interface and the drawing of the tree uses the object-oriented graphics interface to Tcl/Tk [13] available in Oz [11]. We first considered using existing tools for computing and drawing layouts for graphs (e.g., VCG [14], daVinci [6]). Unfortunately, it is hard to design a powerful user interface since the tools have a user interface on their own which can be customized in a limited fashion only. More severe, they fail to support efficient incremental updates of the drawn tree.

7 Creation of the Search Tree

The search tree is represented as a tree of objects where objects representing solved or choice nodes carry their computation space as part of their state. Construction of the search tree is started with creating the root node. If the Explorer is given the unary procedure `P` as the problem to be solved, the root node is created by:

```
meth recompute(S)
   case {IsInt @space} then
      {@parent recompute(S)} {Commit S @space}
   else S={Clone @space}
   end
end
```

Figure 5: Recomputing spaces in the search tree.

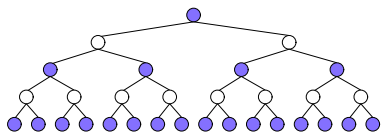
```
RootNode={CreateNode {NewSpace P}}
```

The procedure `CreateNode` (as shown in Figure 4) takes a space `S` as input and returns an object that is an instance of a class corresponding to the status of `S`. The procedure `New` creates a new object of the given class and initializes the object by applying it to the initialization message supplied.

Figure 4 shows the classes from which `CreateNode` creates the search tree's objects. The method `step` of the class `ChoiceNode` explores the children of a given node which are stored under the attribute `kids`. They are created as given by the computation spaces obtained by committing to the respective alternatives. Having the nodes of the search tree represented by objects with attributes referring to parents and children makes it straightforward to implement operations such as depth-first search for a next solution.

8 Using Recomputation

The implementation shown so far is not suited for large problems. The problem is that every node carries a computation space. Computation spaces may contain a large number of variables, constraints and propagators, thus the Explorer might use too much memory.



We trade space for time by using recomputation: Only few spaces are stored, the non-stored spaces are recomputed when needed. The Explorer uses as configurable parameter the so-called recomputation distance. A recomputation distance of n means that only at every n-th layer in the tree computation spaces are stored. The figure above shows nodes with spaces stored as filled for $n = 2$.

To implement recomputation, the attribute `space` of an object either holds the space itself or an integer. The integer value is i if the node corresponds to the i-th alternative of the parent's computation space. Then recomputation of the space can be implemented by the method shown in Figure 5. Sending a node the message `recompute(S)` binds `S` to the corresponding computation space. If the object stores a space, a copy of that space is returned in `S`. Otherwise, the computation space of the node's parent is recomputed recursively and this space is committed to the alternative corresponding to the node.

9 Some Performance Figures

This section gives some rough performance and memory usage figures intended to show the Explorer's practicality. To give an idea, finite domain programming in Oz is competitive in speed with other CLP systems (performance figures for some scheduling problems can be found in [19]). All figures have been taken with DFKI Oz 2.0.3 on a PC with a Pentium 133 MHz Processor running Linux 2.0.27 where the heap memory was fixed to 8 Megabytes. Drawing times are given as wall time, which includes the time spent in the window manager.

The overhead incurred by the Explorer compared to a non-visual search engine depends on how much runtime is spent on constraint propagation. The harder the problem, the less relative overhead the Explorer incurs. Here, we choose the bridge example [4], a medium-sized scheduling problem. Its search tree consists of 743 nodes (371 choice, 3 solved, and 369 failed). The Explorer takes 2.09 seconds to search for the best solution compared to 1.75 seconds for a non-visual Oz search engine (a relative overhead of 19%). Drawing takes another 1.73 seconds for the entire tree. If failed subtrees are hidden away (see Section 5) drawing takes 0.77 seconds.

Even large search trees can be handled. Searching for all solutions of the 11-Queens problem (there are 2680 solutions) creates a tree of 48897 nodes. Creation time is 35.6 seconds (compared to 15.0 seconds for a non-visual search engine), drawing time is 40.1 seconds, if failed subtrees are hidden, and 103.5 seconds otherwise. The search tree needs 4366 Kilobytes, if all computation spaces are recomputed when needed (i.e., the recomputation distance is infinite).

10 Related Work

In the following we relate the Explorer to the Grace tool [12], which is built on top of the ECLiPSe Prolog system [1]. The Grace tool is intended to support the development and debugging of finite domain constraint programs. Rather than using the metaphor of a search tree, it maintains and displays a backtracking history of the finite domain variables involved.

Exploration of the search space is not user-guided but fixed to a depth-first strategy. In contrast to the Explorer it allows tracing of constraint propagation. The display of information supports different levels of detail, but cannot be replaced by user defined display procedures. To use the Grace tool the user's program requires modification.

In the area of parallel logic programming, tools are used to visualize the parallel execution of programs, e.g., the Must Tool [9] and the VisAndOr Tool [2]. These tools also visualize the (OR-parallel) search process, however they are designed to be used off-line. During execution of a program a trace file is created. After execution has finished, the tool is used to visualize and analyze the created trace. This is very different from the Explorer, where

exploration is interactive and user-controlled and where the user has access to the constraints of the search tree.

Acknowledgements

I would like to thank all users of the Oz Explorer who helped to improve it. I am grateful to Gert Smolka for a program on which the example in Section 3 is based, and to Martin Henz, Leif Kornstaedt, Tobias Müller, Joachim Niehren, Gert Smolka, Peter Van Roy, Joachim Walser, Jörg Würtz, Detlev Zimmermann, and the anonymous referees for providing very helpful comments on this paper.

The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105 and FTZ-ITW-9601), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL-II (EP 22457).

References

[1] A. Aggoun, D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Meier, D. Miller, S. Mudambi, B. Perez, E. Van Rossum, J. Schimpf, P. A. Tsahageas, and D. H. de Villeneuve. ECL^{i}PS^e 3.5. User manual, European Computer Industry Research Centre (ECRC), Munich, 1995.

[2] M. Carro, L. Gómez, and M. Hermenegildo. Some paradigms for visualizing parallel execution of logic programs. In *International Conference on Logic Programming*, pages 184–200, Budapest, 1993. MIT Press.

[3] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In *International Conference on Logic Programming*, pages 774–790, Budapest, 1993. MIT Press.

[4] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):74–94, 1990.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, 1988.

[6] M. Fröhlich and M. Werner. Demonstration of the interactive graph visualization system daVinci. In *Graph Drawing, DIMACS International Workshop GD'94*, LNCS 894, Princeton, USA, 1995. Springer Verlag.

[7] M. Henz, G. Smolka, and J. Würtz. Object-oriented concurrent constraint programming in Oz. In *Principles and Practice of Constraint Programming*, pages 29–48. The MIT Press, Cambridge, MA, 1995.

[8] ILOG. ILOG Solver. URL: `http://www.ilog.com/`.

[9] R. Karlsson. *A High Performance OR-parallel Prolog System*. PhD thesis, Swedish Institute of Computer Science, Kista, Sweden, 1992.

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

[11] M. Mehl and C. Schulte. Window programming in DFKI Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1997.

[12] M. Meier. Debugging constraint programs. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, LNCS 976, pages 204–221, Cassis, France, 1995. Springer Verlag.

[13] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, Cambridge, MA, 1994.

[14] G. Sander. Graph layout through the VCG tool. In *Graph Drawing, DIMACS International Workshop GD'94*, LNCS 894, pages 194–205, Princeton, USA, 1995. Springer Verlag.

[15] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7th Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, 1990. ACM Press.

[16] C. Schulte and G. Smolka. Encapsulated search in higher-order concurrent constraint programming. In *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, NY, 1994. The MIT Press.

[17] G. Smolka. The definition of Kernel Oz. In *Constraints: Basics and Trends*, LNCS 910, pages 251–292. Springer-Verlag, 1995.

[18] G. Smolka. The Oz programming model. In *Computer Science Today*, LNCS 1000, pages 324–343. Springer-Verlag, 1995.

[19] J. Würtz. Constraint-based scheduling in Oz. In *Symposium on Operations Research*, LNCS, Braunschweig, Germany, 1997. Springer-Verlag. To appear.