Programming Constraint Services

Konstantin Halachev

Universität des Saarlandes, Saarbrücken 66123, Germany
halachev@mpi-sb.mpg.de

Abstract. The goal of this paper is to introduce the oz approach of modeling search and exploration strategies in constraint based problems.Oz space based model is introduced as a solution to restrictions of older constraint programming systems. The oz model is based on two main ideas. Spaces as a notion of capsulation and abstraction of the constraint computations and recomputation reconstruction model as an alternative to the widely used trailing.

1 Introduction

Constraint programming has become the method of choice for modeling and solving many types of problems in a wide range of areas: artificial intelligence,databases, combinatorial optimization, and user interfaces, just to name a few.In this paper we shortly introduce the basic methodology of constraint programming system (CPS). Chapter 1 shows a summary of the main problems of the older CPS and the Oz approach of tacking them. Chapter 2 makes a short introduction of how CPS work. Chapter 3 introduces the notion of spaces with some examples how they can be used. Chapters 4 and 5 focus on the reconstruction methods, starting with the naïve and trailing and then presenting the recomputation approach and some of its modifications.

1.1 Problem Overview

A cornerstone for the initial success of constraint programming has been the availability of logic programming systems. All today's systems evolved from them and inherited their problems and restrictions. The main ones are formulated in the following paragraphs.

Search: All these systems have small fixed number of search strategies. They do not allow search to be programmed and search is usually hard-wired to depth-first exploration. Lack of high-level programming support for search is an impediment to the development of new strategies and the generalization of the existing ones.

Concurrency: Integration of CPS into today's concurrent computing is made difficult or even impossible. This is because the backtracking model inherited from Prolog is incompatible with concurrency.

Programming backtracking: Each node in a search tree represents an abstract node, but also it is a process. This process consists of set of propagators and a constraint store that change each other until the node reaches a ‘stable’ state. However using backtracking reconstruction model we have the need to be able to restore the initial state of the node. This requirement becomes harder to program the more complex the problems are.

1.2 Outline of Oz approach

The approach taken by Oz is to devise simple abstractions for the programming of constraint services that are concurrency-enabled to start with and overcome the problems discussed in the previous section.
Oz introduces first-class computation spaces which are responsible to handle and encapsulate constraint-based computations thus providing object-oriented abstraction.
Spaces are applied to state-of-the-art search engines, such as plain, best-solution and best-first search. Programming techniques for space-based search are developed and applied to new and highly relevant search engines. Given that search engines and search strategies are now programmable, developers may choose or develop a specific search engine best matching their problem.
In order to be expressive and compatible with concurrency, reconstruction is based on copying rather than on trailing.

2 Introductions to Constraint Programming

In this chapter we overview the CPS process from scratch.

2.1 How constraint programming systems works?

It all starts with some problem to be solved. In order to use constraint programming system we fit this problem in a model consisting of set of variables and a set of constraints for these variables. Constraints are mainly divided in two types: basic and non-basic.

Basic constraints evaluate some variable domain .They are of the kind $x \in D$ .The set of all basic constraints forms the constraint store.

Non-basic constraints express relation between two or more variables and are computationally involved. Each non-basic constraint is imposed by a propagator. Propagator expressing specific constraint has the role to narrow the variables domains to only the domains subset fulfitting the constraint. Whenever a constraint can be propagated in order the solution set to be narrowed, the propagator relevant to this constraint takes the appropriate action by telling the constraint store some relevant basic constraint. A propagator becomes ‘entailed’ if it detects that the constraint it expresses is always

relevant in the constraint store values domains. It becomes ‘failed’ if the constraint is inconsistent with the store.

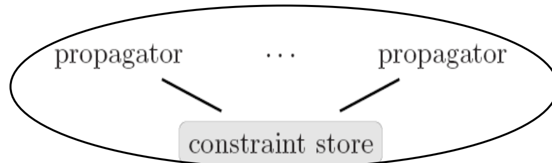


Fig. 1. Constraint store and propagators

A space S is called ‘stable’ if no propagation can further modify the constraint store. A stable space S is called ‘failed’, if S contains failed propagator. A stable space S is ‘solved’ if S does not contain any propagators. However a space may become stable but be neither solved nor failed. This space is called distributable.

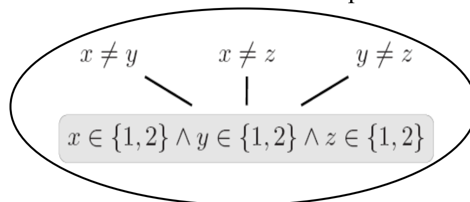


Fig. 2. Distributable space

2.2 Distribution

If a space reaches distributable state ‘distribution’ has to be made. ‘Distribution’ is the action of processing to easier to solve spaces but retaining the same solution set. Common distribution strategy is dividing the domain space into two new spaces by introducing a new basic constraint β. Distribution of spaces with respect to β results in two new spaces $(S \wedge \beta)$ and $(S \wedge \neg\beta)$.It is crucial to choose β so that further propagation is triggered. β and $\neg\beta$ are called alternatives. ‘Distribution’ is also known as labeling or branching.

Popular distribution strategy is to select a variable with non-singleton domain and split its domain into two distinct parts. However even in this simple distribution strategy there are still refinement decisions that affect its usefulness like which variable to choose ,how to split the domain and so on.

In the general case distribution uses set of constraints to create alternatives. Iterating constraint propagation and distribution lead to a tree of spaces called search tree. The distribution strategy chosen in a problem defines fully its search tree. This tree is then independent on the exploration strategy.

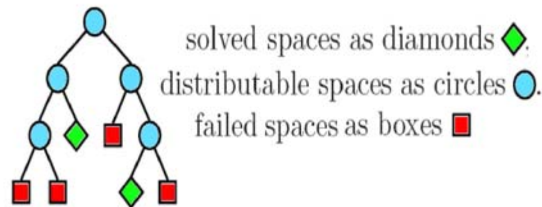


Fig. 3. A possible search tree as a result of specific distribution strategy

2.3 Exploration

Search tree is defined entirely by the distribution strategy. An orthogonal issue in finding problem solution is how to explore the search tree. A program that implements exploration is called search engine. A strategy for exploration is referred to as search strategy.
Some of the most common search strategies are depth-first and breadth-first exploration. However there are many more advanced search strategies.
Besides of different strategies engines can offer a great variety of functionality, like:
--search for single, several or all solutions.
--interactive or visual search.
--search in parallel

2.4 Best Solution Search

For a large class of application only the best solution with respect to some criterion is needed. The naïve way for solving such problem is finding all solutions and then choosing the best one. However constraint programming systems provide a better and faster way of solving this class of problems. The idea is to use the currently computed solutions to reduce the search space. This is done by injecting a new constraint into the problem which states that the next solution found should be better than the current best one. In this manner the next solution found will be better than the current one and due to the increased number of constraints the search space is reduced. In this solution the order of exploration influences the search tree and it is no longer independent of the exploration strategy.

2.4.1 Branch and bound Best Solution Search Example

Here we present an example of how branch and bound BSS works.

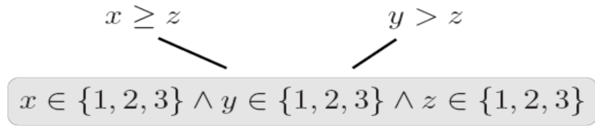
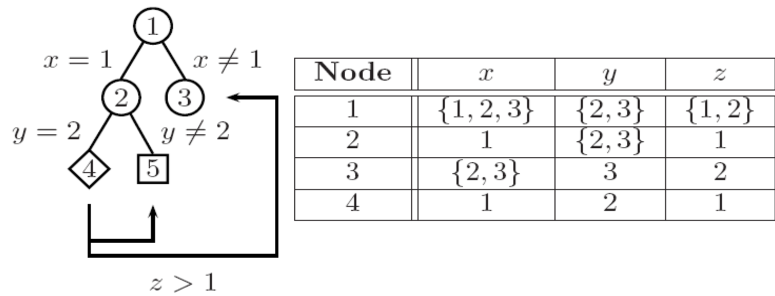
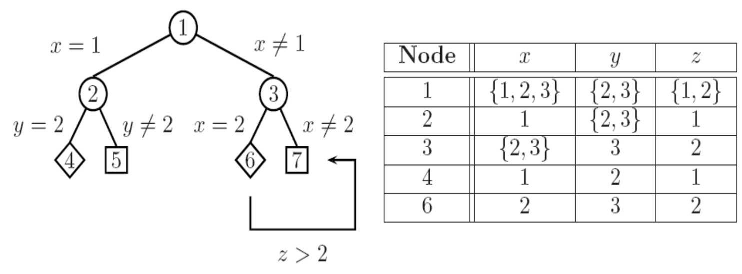


Fig. 4. The problem for which we search for a solution where z is largest



Node	x	y	z
1	$\{1, 2, 3\}$	$\{2, 3\}$	$\{1, 2\}$
2	1	$\{2, 3\}$	1
3	$\{2, 3\}$	3	2
4	1	2	1

Fig. 5. After exploring the left branch a new constraint ‘z > 1’ is introduced to the right branch



Node	x	y	z
1	$\{1, 2, 3\}$	$\{2, 3\}$	$\{1, 2\}$
2	1	$\{2, 3\}$	1
3	$\{2, 3\}$	3	2
4	1	2	1
6	2	3	2

Fig. 6. After second (best) solution found

3 Introduction to spaces in Oz

In this chapter we introduce the spaces notion. Why do we need spaces, how are they implemented and other questions are answered.

3.1 Need for spaces

When we represent the search tree corresponding to a problem, each node is viewed as an abstract point for which we ignore the computations and the data structures that it contains. But as we have already seen each node is a process of propagators triggering and changing variable domains until this node reaches stable state. The computations that are executed and take place in this node are important part of the process of computing solution of the problem. The top level process need to be protected from possible speculative calculations in these nodes and also must not influence them. The approach to that comes with the notion of spaces, where space is used to encapsulate all these speculative node computations thus making the system safe, easier to program and providing possibilities for concurrency.

3.2 Oz light

Computation in Oz takes place in computation space. A computation space features multiple threads computing over a shared store. The constraint store contains information about possible values of the variables represented by a conjunction of basic constraints. For the purpose of this paper it is sufficient to regard a propagator as a thread that implements constraint propagation.

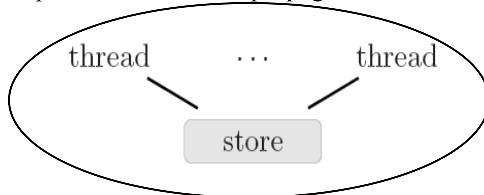


Fig. 7. Oz computation space

3.3 Spaces in Oz

Computation spaces have been introduced as a central mechanism for search. Here the integration of spaces into Oz in order to program search engines is shown.
A local computation space (LCS) encapsulates speculative constraint computations. The execution in LCS resembles the execution in the top-level space. However the LCS does not have any access to the variables of the top level thus ensuring that after the initial setup the computation is independent of the top-level space. Next we show the basic operations on spaces which Oz implements so that spaces are made powerful enough abstraction.

3.3.1 Operation on spaces

Space Creation - {NewSpace x y}
This operator creates a space y which executes the procedure x.

Merging Spaces – {Merge x y}
This operator merges space x with the top level one and returns the result from space x in variable y.This functionality is required for two reasons. First, accessing the result of speculative computation and second accessing the entire computation by removing the encapsulation barrier.

Injection into Spaces - {Inject x y}
This adds some new computations y to the space x. Spawning new computation in a space is useful to achieve more powerful programming techniques. An example is the best-solution search or the case when we need to kill a space without waiting its computation to end.

Status Access – {Ask x y}
This operation blocks until space x becomes stable and then returns the status of the space. The status of space x is returned in variable y.It can be:
--'failed', if space x failed
--the solution reached if the space is solved
--'alternatives(n)' where n is the number of possible branching alternatives if the space is distributable.

Choose possible alternatives – {Choose x y}
Blocks until x is determined to a natural number n, and then expects 'Commit' to choose alternative.

Committing to alternative – {Commit x y}
Commits the space x to the alternative with number y from the possibilities defined by the Choose operation.

Clone Space – {Clone x y}
Creates a full copy of space x in variable y.

3.4 Depth first search – explanation and code

```
fun {DFS P}
   case {DFE {NewSpace P}} of nil then nil
   [] [S] then [{Merge S}]
   end
end

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

Fig. 8. Programming depth-first search in the terms of spaces.DFS is a procedure which takes a script as input creates a new space to execute the script, and then applies DFE to the newly created space, finally returns the result from DFE. DFE is a procedure which takes a space as input and tries to solve it using depth-first strategy. If the space becomes distributable, then the procedure creates a clone of the space and runs the procedure for the first alternative. If this branch does not result in solution then the saved clone of the space is committed to the second alternative. This procedure can be easily changed to calculate all solutions. A better implementation idea is to raise exception whenever a solution is found so that the canceling process up the tree is omitted.

4 Naïve and Backtracking approach

In this chapter we present the naïve and backtracking approach for reconstruction during search and the problems they encounter.

4.1 Naive Approach

The naïve approach of searching through a search tree is to keep in memory the states of all the traversed nodes and the ones, which are about to be traversed. This approach is simple to program. However, the problem is that even in not so large problems the amount of space required for these node states to be kept in the memory is much. Much unused and redundant information is saved. A possible improvement to not keep in the memory

the nodes which are fully traversed. But again the memory needed is at least linear in the depth of the tree.

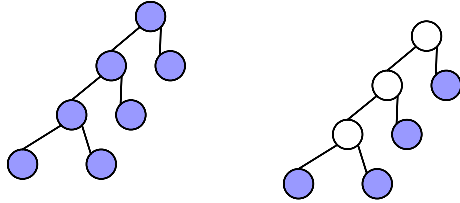


Fig. 9. The blue circles represent spaces that are kept in the memory. On the left we see the full naïve approach and on the right we see if we use the improvement to not save the spaces which are fully traversed.

4.2 Backtracking approach – trailing

The trailing approach was implemented in Prolog to handle the memory problem of the naïve approach. It allows a tradeoff between computation time and memory used. The idea is to keep in memory only the state of the current node and the information needed to backtrack it to the previous state. However in more complex problems where node structures may have internal states, retrieving the state of the preceding node can be hard problem. For each branch we keep information for the differences between the two nodes so that later by using this information the previous node state is reconstructed.
The problems with this approach are as follows: Trailing is hardwired with DFS and it is hard to be used with another search strategy. It cannot be integrated for concurrent search. The undo operation becomes time consuming in more complex problems.

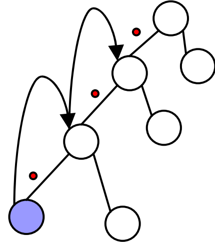


Fig. 10. The blue circle is the current state, the only one which is kept explicitly in the memory. The red points represent the difference information between the states and are used for the undo operation when we backtrack.

5 Recomputation

5.1 Recomputation basic introduction

There are two basic techniques for reconstruction of a node state: trailing and recomputation.Trailing is based on undo operation and backtracking to the previous state.Recomputation is based on copy operation. Main idea of recomputation is that any node can be computed without search from the root node and the description of its path. For recomputation only two nodes are needed to be saved in the memory. These are the current node and the root node. Whenever we need to explore a new node for which we know the path, we start with the root node we clone it and commit it consecutively to an alternative which form the path to the desired node. This approach uses only small amount of memory. On one hand it can be time consuming ,but on the other hand copying is a simpler than operation undo and thus recomputation can be much faster than trailing for complex problems.

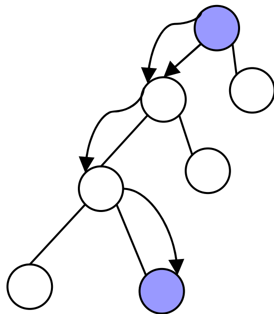


Fig. 11. The recomputation path of a node

However it might be the case that full recomputation requires many exploration steps and involves much computation so there are some modifications and optimizations of this technique.

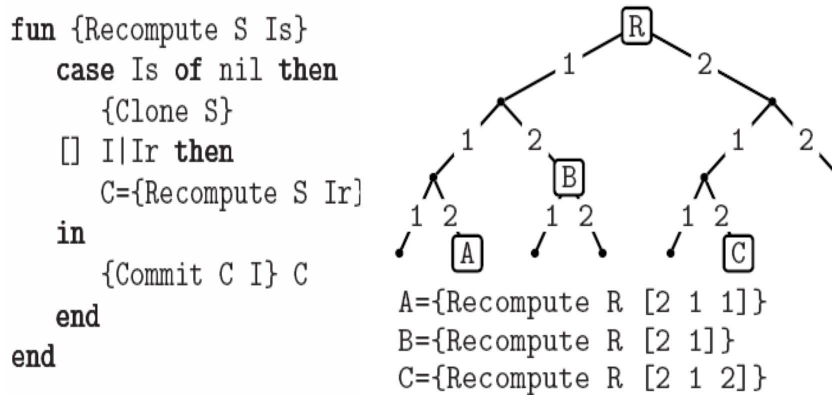


Fig. 12. On the left is the code for recomputing a state using the root and the path and on the right a search tree is shown with example runnings of the recomputation procedure for three nodes

5.2 Last Alternative Optimization

Last alternative optimization uses the observation that when all but one alternative A of the root node N are explored, all other recomputation will start with recomputing the branch between N and A. In order to save resources computing this move for each node, we can switch the root node to be A and mark N as fully traversed.

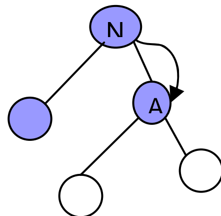


Fig. 13. When the left branch of N is traversed the only possible not traversed alternative A of N becomes root.

5.3 Fixed and adaptive recomputation

Another modification of the full recomputation is if we explicitly ensure that each recomputation will not have more than a fixed number of steps called maximum recomputation distance (MDR). We achieve this by saving additional nodes for each path at distance equal to MDR from the last saved node. This consumes more memory resources but achieves constant recomputation time independent of the search tree. This modification allows a tradeoff between memory and computational resources.

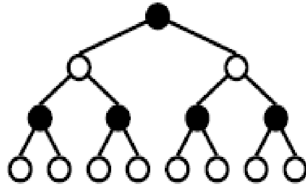


Fig. 14. This picture is an example of a search tree when using fixed recomputation.The black nodes are the ones which are kept explicitly in the memory.

Another modification of full recomputation is if during recomputation from N to A, additional copy is saved of the space which is in the middle of the path between N and A. These techniques provide flexible possibilities for flexible adjustable tradeoff between additional memory used and computational time.

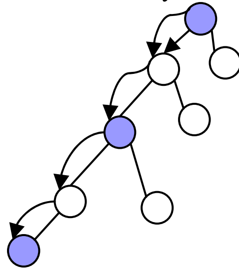


Fig. 15. Additional saved space in adaptive recomputation

5.3 Batch Recomputation [Henz 2001]

Batch recomputation is recently invented technique which shows better results then the recomputation modifications of which were mentioned above. In recomputation process at each step we introduce some new constraints to the node and we run the propagation step until we compute a stable space and. But given that the final state is known, we can run the propagation process only once after having included all the constraints which will be introduced for the target node. In this manner a lot of computations might be saved because of the larger number of constraints which are introduced at the same time. Condition for correctness of this method is the monotonicity of constraints.

C_0,C_1,C_2 - Constraints

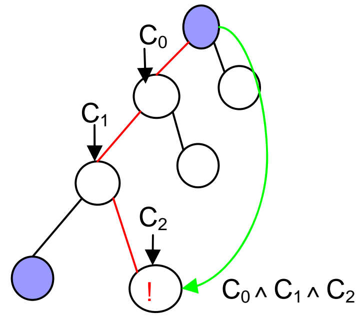


Fig. 16. The red line represents how the normal recomputation is executed and the green line shows how batch recomputation is executed

6 Advantages of Oz space based model

To recapitulate the main advantages of Oz space based model are:

1. Oz encapsulates constraint computation in spaces thus providing a safe way to handle speculative calculations
2. Oz model is easier to program no matter how complex the internal structures of a space are because no support for the undo operation is needed
3. Oz allows programmers the freedom to program every aspect of the solution finder so that it solves the problem better and faster. Oz provides possibility to develop problem dependant search engines.
4. Oz model allows computing in distributed and concurrent environment, which is the basic way of tackling larger problems.

The new ideas and possibilities which came with Oz space based model gave an incitement of programming systems, which were considered old fashioned and no longer effective. Constraint programming systems are again a research field which attracts many people's interest and efforts in exploring the newly discovered horizons.

References

[Schulte 2002] Christian Schulte. Programming Constraint Services : High-Level Programming of Standard and New Constraint Services, LNAI 2302 Springer-Verlag, 2002.

[Henz 2001] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. LNCS 2239, Springer-Verlag, 2001