CS4402 - Constraint Programming
Practical 2: Solving

110021627

April 2016

Introduction

The aim of this practical was to implement a constraint solver using a forward checking algorithm with d-way branching. 2-way branching could have also been developed as an addition to the main task. Furthermore, it was required to support variable assignments based on user specified static ordering or on the smallest domain first principle. Finally, experiments had to be carried out to compare the two rules of variable assignments using instances from a chosen problem class.

Solver Implementation

In order to design a solver, decisions had to be first made with regard to a representation of variables together with domains and constraints. Thus, separate classes were created for decision variables, matrices, expressions and constraints. The *DecisionVariable* class, thus, comprises of variable name and domain. A variable name element is not a compulsory one and can be omitted for example in the case where variables exist within a matrix. A domain is represented with a *TreeSet* data structure since there cannot be any duplicate values in a domain and a set ensures that. The *Matrix* class containes a double *ArrayList* of *DecisionVariable* for an easier construction of and iteration through matrices that would be useful for such problems as, for instance Sudoku. The *Expression* class was made to deal with arithmetic expressions within constraints and is constructed with decision variables, operators and numeric values as attributes. It supports basic mathematical operations such as addition, subtraction, multiplication and division and can be easily extended to support more of these. The *Constraint* class includes expressions on the left and right sides and a comparator between them. A comparator in its turn can be one of the following: $<, >, \leq, \geq, =$ or $\neq$.
Methods to solve an expression or to check the validity of a constraint with provided values for decision variables are implemented within the corresponding

classes. Moreover, the *DecisionVariable* implements the *Comparable* interface to be able to later sort variables according to the sizes of their domains.
During the constraint problem specification all the variables and constraints are placed into *ArrayLists* and then passed to he solver.
For a static ordering of variable assignments the forward checking algorithm was implemented in the same way as described in lectures. The variable located in the *ArrayList* with decision variables at index which equals to the current depth is set in a loop with values from its domain. Once a value is assigned the consistency of those arc that contain the variable that was just set and future variables is checked and the domain is reduced for the variables that have not been set with a value yet. Before the first reduction of the domain of a particular variable is made the state of the domain is saved so that if the current branch does not lead to the solution domains could be restored. Thus, if the domain of any variable becomes empty during reductions, the changed domains are recovered and the next assignment is made. However, if after arc revisions all variables have something in their domains the variable that is located one level deeper is given a value and so on. If all variables are set with values, the solution is found.

Heuristics

No changes in the implementation needed to be made to support user defined static ordering. The user can simply changed the order of variables in the *ArrayList*.
Nevertheless, there was a slight difference in the implementation of forward checking algorithm when the principle of smallest domain first was applied. Instead of choosing the variable with the current depth index for the next assignment, the unassigned variable with the smallest domain was found and chosen.

Additional features

In addition the ac3 algorithm from the lectures was implemented for an initial check of arc consistency. It first goes through unary constraints and then goes through binary constraints and reduces domain of variables if necessary. If while revising the arc (x_i, x_j) the domain of a variable x_i is changed, all the arcs (x_k, x_i), $k \neq j$ are added to the queue of arcs to be revised if they are not there. If during the ac3 algorithm the domain of any variable is emptied, there is no solution to the problem and the forward checking method is never called.
Furthermore, some of the parsing was implemented to simplify the process of problem specification.
Finally, forward checking algorithm with a support of 2-way branching was attempted by a minor modification of the forward checking algorithm with d-way branching with dynamic ordering. In the implementation of the latter after reaching a dead end the same variable is chosen to be assigned a next value to

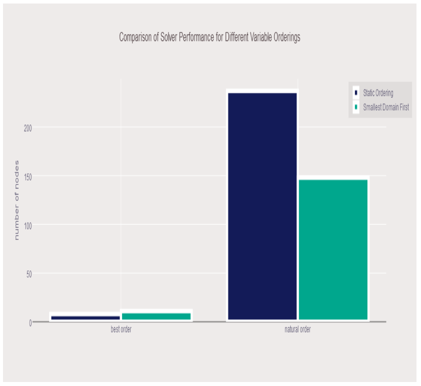
since this variable is still the one with the smallest domain. For 2-way branching instead of choosing the variable with the smallest domain it is selected randomly from a list of unassigned variables. Thus, on one branch assignment $x = m$ is made, if the dead end is reached pruning is undone and the assigned value is removed, so that on the next branch $x \neq m$ and another variable can be chosen.

Experiments

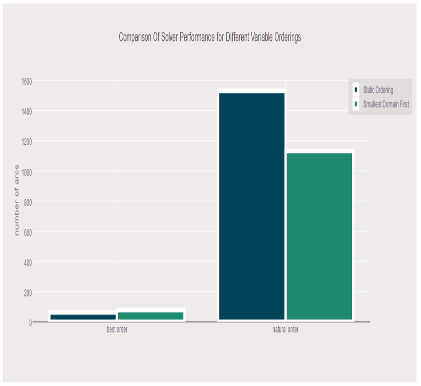
The parametrised crystal maze puzzle from the lecture slides was chosen. The aim of this puzzle is to assign a value for each node in the given network of order n such that every node x_i has a different value in the range from 0 to $n-1$ and the values of nodes that have an edge between them are not consecutive. Thus, this problem for the solver is specified by creating n decision variables with the domain from 0 to $n-1$ and a list of inequality constraints.
To compare the performance of the solver with static and dynamic ordering of variables three types of ordering was considered.

- natural order (variables ordered from x_0 to x_{n-1})
- random order
- best order (the order in which variables should be assigned to immediately lead to solution)

Initially it was decided to compare them according to the number of nodes traversed, number of arcs revised and the time taken. However, since the problem was always the same after the first run the time taken would be significantly less for the rest of them. This might be explained by the JVM optimisation and therefore, the time was not considered.
The graphs below show the differences in the number of nodes and arcs for various static and dynamic orderings.

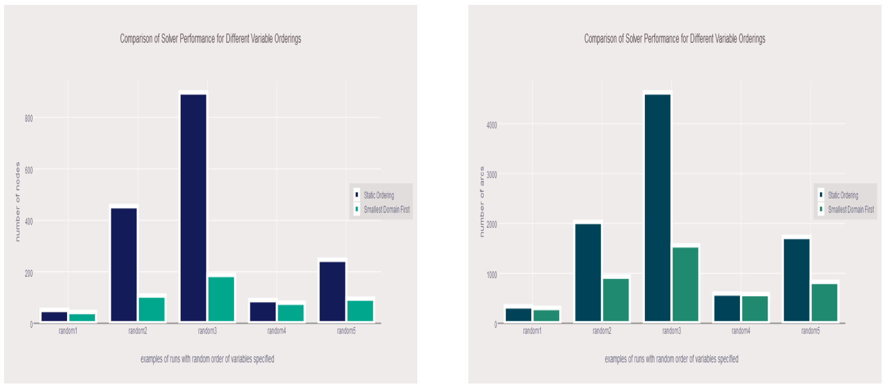


(a) number of nodes



(b) number of arcs

Figure 1: Best and natural orders.



(a) number of nodes (b) number of arcs

Figure 2: Random order. Examples of Several Runs.

Since the results for the random order varied, the average across 1000 runs was calculated and is visualised below.

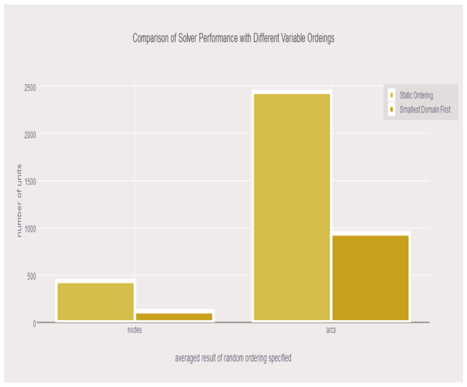


Figure 3: Number of nodes and arcs for the averaged result of random order.

All the results except when the best order was specified showed that the solver with dynamic ordering that uses smallest domain first rule behaved better than with the static ordering. It makes sense that with the best order the static ordering works well since no search needs to be done and even then the result with smallest domain first technique was not much worse. The reason for this is that for this problem initially every domain size is the same, so the first variable taken to assign a value to is the right one even for the dynamic ordering implementation and the choice of the first variable is the most important one.