University of
St Andrews

CS4402 Constraint Programming

Constraint Solver Implementation

April 17, 2018

Lecturer:
Ian Miguel

Submitted By:
140011146

1 Introduction

In this practical, a forward checking constraint solver with 2-way branching is implemented. The implementation provides an interface to allow different heuristics for variable and value selection. To test the solver works as intended, problem instances of the N-queens, langfords and sudoku problems were generated and solutions created. Further, tests were written to ensure the right number of solutions were found for problem instances, for example exactly 724 solutions to the 10-queens problem. Finally, the performance of the solver and different heuristics are tested empirically using various metrics to explore and understand their differences.

2 Design and Implementation

Many of the algorithms presented in this section were taken and adapted from the lecture slides to provide a complete method of what was done and any changes made during the implementation when compared to the original algorithm.

2.1 2 way forward checking

The 2 way forward checking algorithm was implemented from the following algorithm:

Algorithm 1 2-way forward checking algorithm, adapted from lecture slides by Ian Miguel.

```
1: procedure ForwardChecking($varList$)
2:     if completeAssignment() then
3:         addSolution()
4:     else if empty($varList$) then return
5:     else
6:         $var \leftarrow$selectVar($varList$)
7:         $val \leftarrow$selectVal(domain($var$))
8:         branchFCLeft($varList$, $var$, $val$)
9:         branchFCRight($varList$, $var$, $val$)
```

The small change made to the algorithm was to not exit when the first solution was found, but to add that solution to a list and continue to find more solutions. This was done to allow all solutions to be found as a means of testing the solver is working properly. To check for `completeAssignment()`, all variables in the problem instance contained a boolean to indicate that they have been assigned. `completeAssignment()` is true if all the variables are assigned. The variables can further be unassigned later by changing the assignment boolean in the corresponding variable.

The two methods `selectVar()` and `selectVal()` are implemented using heuristics which are explained below in section 2.2. Each heuristic implements its own variable and value selection functions which the solver uses, allowing the solver to be generic and extensible over any heuristic. Further, this means the heuristics do not have to worry about any other part of the solver, as the list of variables and domain of values are updated and kept track of in the solver to be passed to the heuristics.

Next, we look into detail on the algorithms for left and right branching.

Algorithm 2 Branching left during forward checking.

```
procedure branchFCLeft($varList$, $var$, $val$)
    assign($var$, $val$)
    if reviseFutureArcs($varList$, $var$) then
        ForwardChecking($varList$ - $var$)
    undoPruning()
    unassign($var$, $val$)
```

When we branch left, the selected value is assigned to the selected variable and future arcs revised. The idea is to ensure that for any particular value assignment, the arcs are revised to ensure local arc consistency. If the arc is still consistent, then we explore further down the tree by recursively forward checking with the reduced list of variables.

When the functions eventually return or if the arc is inconsistent, the current value assignment does not lead to a solution, so any pruning done during arc revision must be undone for backtracking as the value assignment has not worked out. This is done with the `UndoTracker` class which keeps a map of each variable and the list of pruned values that need to be undone. Next, the solver contains a map of each variable to an `UndoTracker`. Whenever arcs are revised, the tracker for the variable chosen for revision is updated with the list of other future variables and their pruned values. This works to prune only values from each step of the recursion, because every time the algorithm goes down the left branch, a new variable is selected to be assigned and its arc revised. So when coming back up the tree, each depth corresponds directly to the selected variable, which acts as the key to an `UndoTracker`.

Algorithm 3 Branching right during forward checking.

```
procedure branchFCRight($varList$, $var$, $val$)
    deleteVal($var$, $val$)
    if $\neg$ empty(domain($var$)) then
        if reviseFutureArcs($varList$, $var$) then
            ForwardChecking($varList$)
        undoPruning()
    restoreVal($var$, $val$)
```

As this is a 2-way branching scheme, the right branch is the branch where the given value assigned on the left branch is now removed. This removed value in `deleteVal()` does not have to be stored or kept track of as the same value is passed to the `restoreVal()` method later in the same function scope. Future arcs are revised again before any further forward checking and of course the pruning is undone afterwards as well.

Finally are the algorithms for revising arcs. The algorithm for `reviseFutureArcs()` is not shown, as it remains unchanged. For all variables passed into it, the arc between the currently assigned variable and the list of future variables is revised and if the local arc consistency is broken at any stage, the function returns false. The method for which each arc is revised is detailed as follows:

Algorithm 4 Algorithm for revising an arc.

```
1: procedure REVISE(Arc($var_1$, $var_2$))
2:     $val$ ←$var_1$.assignedValue()
3:     $constraints$ ←getArcConstraints(Arc($var_1$, $var_2$))
4:     if empty($constraints$) then return False
5:     else
6:         for each constraint $c$ in $constraints$ do
7:             for each value $v$ in domain($var_2$) do
8:                 if ¬ constraintHolds($c$, $v$) then dropVal($var_2$, $v$)
```

There were a few difficulties encountered during the implementation of the revision step. First, because of the way the assigned values are implemented in any given variable, if a variable has been assigned, `revise()` only uses the assigned value to revise the arcs to all other variables. This means to deal with revision during right branching, the revision function has to be altered to revise for each value in the variable's domain. A more streamlined way to handle this would have been to represent assigning a value by only leaving the assigned value in the domain of the chosen variable. That way the same code can be reused for both left and right branching, as both options loop through all values in the variable's domain. The second issue was an issue with the way the `.csp` constraint problems were written and parsed. In the `.csp` format, an arc is specified both ways under one `BinaryConstraint` which contains both variables.

Listing 1: Example of how constraints are specified in `.csp`

```
c(0, 1)
1, 3
2, 4
3, 5
4, 6
```

In the above listing, the tuples in constraint `c(0, 1)` applies both ways and have to be matched accordingly, rather than simply being the arc in the direction arc(v_0, v_1). For example if the selected variable was v_1, the second value of the tuples have to be used rather than the first. The change needed to implement this was to keep a copy of the tuples reversed so that when matching to the constraint, the reversed tuples can be returned if that order is needed (i.e, checking the arc(v_1, v_0) rather than the arc (v_0, v_1).

Furthermore, there is a distinction between no arc existing between two variables and no constraints in an arc between two variables. An example of the first case is common in Sudoku problems, where two cells that don't share the same row, column or grid do not have an arc between them as they do not affect each other. An example of the latter case is in the 2-queens problem, where there are no constraints specified between the first and second row, because there are no valid values for either. This was a case that had to be handled carefully, as the arcs are still consistent if no arc exists between two variables, but the arc not consistent if there are no valid values for the arc, as in the 2-queens problem.

2.2 Heuristics

The heuristics used for the solver are implemented as an abstract class to allow any additional heuristic to be easily implemented. The class exposes three functions for any subclass implementations:

- `getNextVariable(`*varList*`)`
- `getNextValue(`*varDomain*`)`
- `toString()`

The first two allow any implementation of a heuristic to choose the next variable and value that should be returned from the list of variables and domain of values. The `Heursitic` abstract superclass implements a default behaviour which returns the first element from the list of variables and domain values. This further allows a heuristic implementation to choose to only implement the choice of variables, value or both and default to choosing the first element otherwise. For example, the heuristic `RandomValueHeuristic` chooses the next value randomly from the domain, but uses the default behaviour of the superclass to choose variables.

The `toString()` method simply gives a name for the heuristics to easily identify them for data gathering reasons.

2.2.1 Static variable heuristics

A few simple static variable ordering heuristics were used:

- **Ascending order** - The static ascending order heuristic is the default heuristic which always chooses the first element in the list of variables.
- **Descending order** - Order of the variables are chosen in descending order. For example if there were 3 variables v_1, v_2, v_3 the order for choosing variables becomes v_3, v_2, v_1.
- **Odd/even order** - All odd numbered variables are chosen first, then even numbered variables. The variables are further chosen in ascending order. For example, given 5 variables v_1, v_2, v_3, v_4, v_5, the chosen order becomes v_1, v_3, v_5, v_2, v_4.
- **Maximum degree** - Choose variables which are the most constrained first, as they are likely to be the hardest choices. This was done by calculating the total number of possible value combinations between each pair of variables minus the number of tuples between the pair from the CSP description. This gives the total number of constraints for one variable to all other variables. The variables are sorted and chosen by highest number of constraints first.
- **Minimum degree** - Simply the opposite of maximum degree, where the variables with the *least* number of edges in the primal graph are chosen first. This heuristic was implemented to show that the opposite behaviour of maximum degree performs more poorly.

For all these variable heuristics, the values from the domains are chosen in ascending order as the domain is stored as a sorted set, so the first value chosen is the lowest one.

2.2.2 Dynamic heuristics

The only dynamic heuristics implemented was smallest domain first and its inverse (largest domain first). These were done by checking the domain of each variable in the list of variables and choosing the one with the smallest or largest number of domain values.

2.2.3 Random heuristics

Random heuristics were implemented for empirical testing to see how they would perform if a random variable or value assignment was chosen on every step. The three different random heuristics (`Random`, `RandomVariable` and `RandomValue`) also allow us to see the difference between variable and value heuristics. `Random` randomly chooses both variable and value, while `RandomVariable` and `RandomValue` only choose the variable and value respectively as their name implies. The random nature of these heuristics combined with many iterations should be able to show any differences between choosing the variable and value, showing if one may impact performance more than another. There is also a fourth `RandomStatic` heuristic, which statically chooses the random order at the beginning rather than choosing randomly every step. This is subtly different from a dynamic random order because each left subtree that is explored will stay in the same random order, whereas choosing randomly dynamically will change every subtree to be different and random.

2.2.4 Value heuristics

The idea of using value heuristics seems very problem dependent. For certain problems, the value of the variable may matter, but it may not matter for others. As such TODO

3 Experimental methodology

3.1 Testing

To make sure that the constraint solver worked as intended, JUnit tests were written. Because the constraints problems and solutions can be very large, it was not feasible to write tests that checked the correctness of all solutions. However, smaller problems such as the 4Queens problem could be tested. Furthermore, the solver can be specified to output all solutions in which case the number of solutions (including all symmetrical solutions) can be tested to ensure the correct number of solutions were found.

Further, tests were written for each heuristic to make sure each heuristic was not doing anything wrong that changed how the solver worked.

3.2 Metrics used

Multiple metrics were used to empirically measure the performance of different heuristics. Time, number of solver nodes and number of arc revisions were all used as metrics. The number of solver nodes was calculated as the number of times `forwardChecking()` was called and did not return immediately as it represents a step down into the recursion and search tree. The number of arc revisions was calculated each time a constraint between two variables was revised. For example, if there are two variables and two and three values in their domains respectively, the number of arc revisions between one variable to another is 6. Finally the time taken only takes into account the time the solver took to solve the problem and find all solutions rather than the total time of the whole process.

3.3 Experiments

Multiple runs of the experiments were done to ensure reliable results for using the random heuristics. The number of search nodes and revisions do not normally require multiple iterations because they are deterministic, so every run would lead to the same resulting number of arc revisions and solver nodes. However a "random" heuristic that randomly chooses the next variable to assign would require the extra runs to determine an average number of nodes and arc revisions.

The time taken was also measured during the experiments for completeness. Due to time benchmarking issues in Java due to the JVM [1], the time taken is not unreliable without a robust benchmarking framework.

To run the experiments, an `Experiment` class was developed that could take a list of heuristics, number of runs and constraint problems and run the solver, returning a `Result` object which contained the time taken, number of solver nodes and number of arc revisions averaged over all the runs.

4 Results and analysis

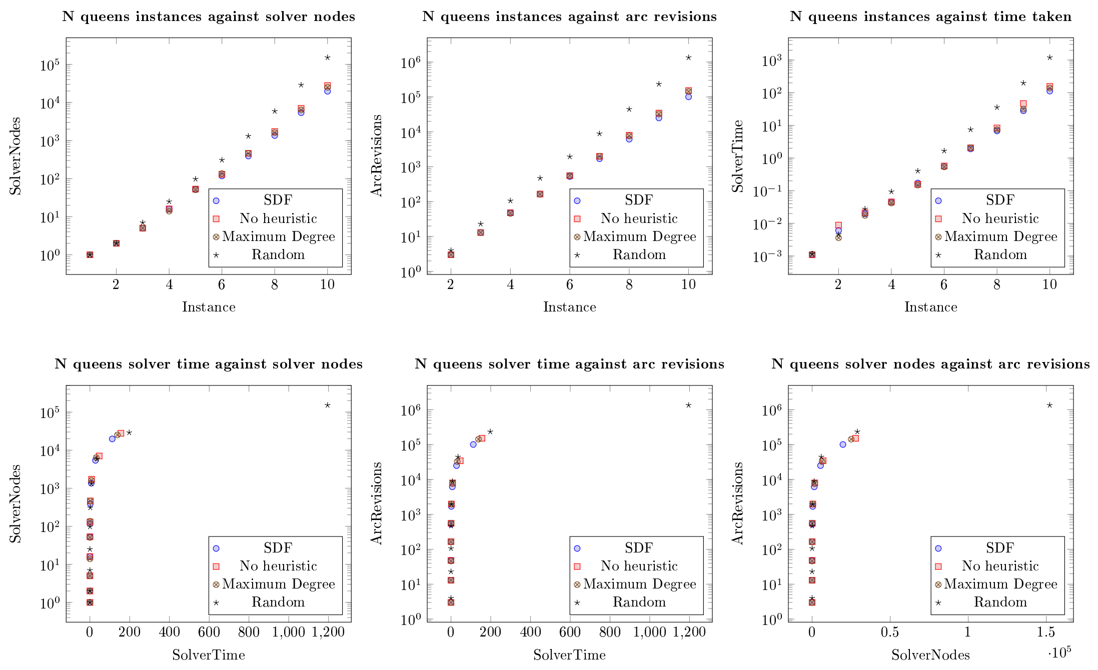

Figure 1: N queens problem performance with different metrics

Figure 2: N queens problem performance with different metrics and heuristics.

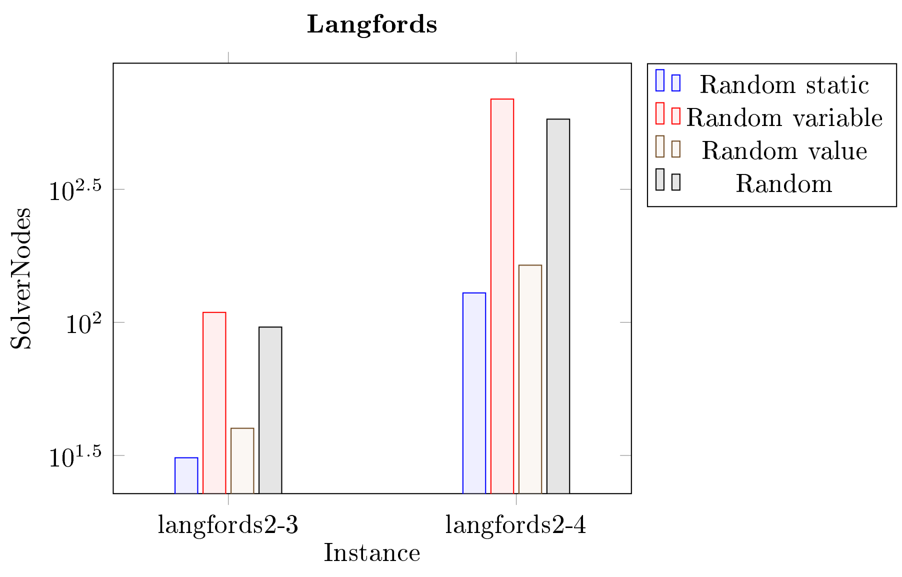

Figure 3: Langfords problem with different metrics and heuristics.

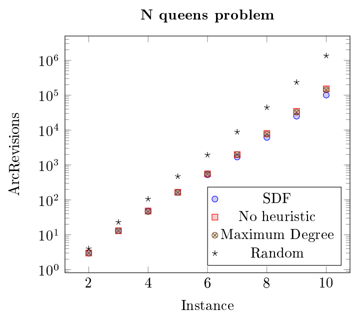

Figure 4:

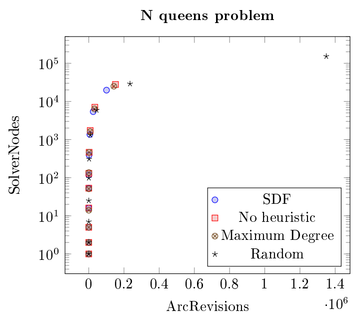

Figure 5:

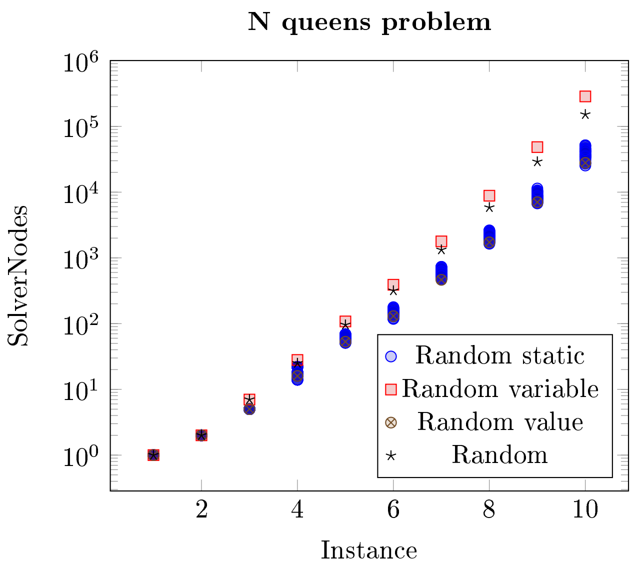

Figure 6:

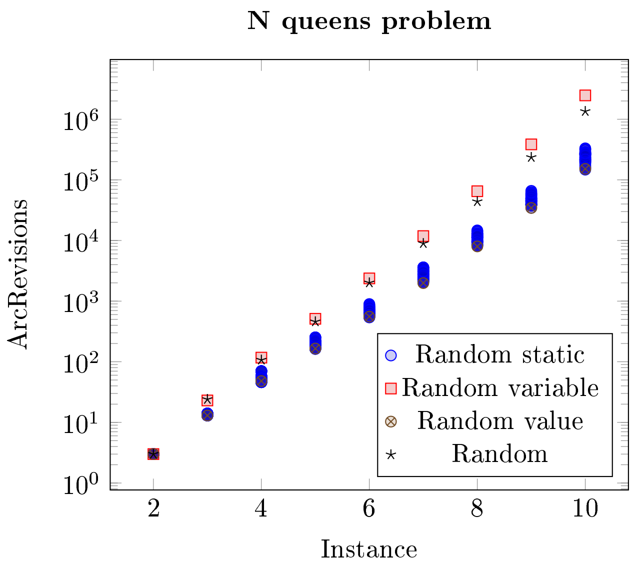

Figure 7:

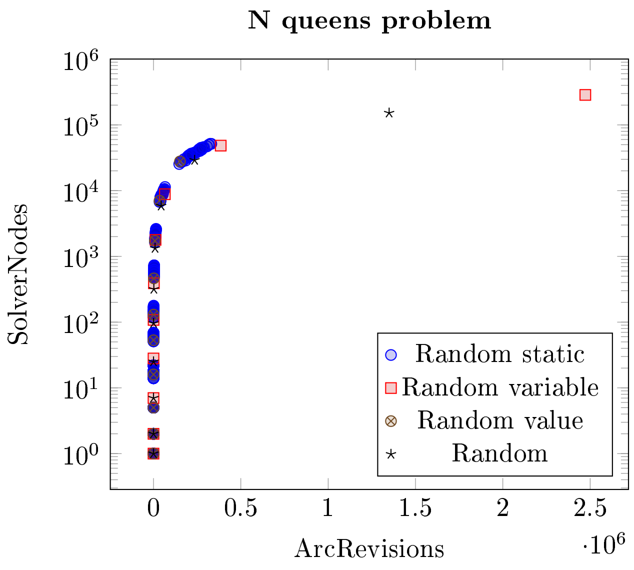

Figure 8:

5 Conclusion and evaluation

References

[1] Julien Ponge. *Avoiding Benchmarking Pitfalls on the JVM*. 2014. URL: `http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html`.