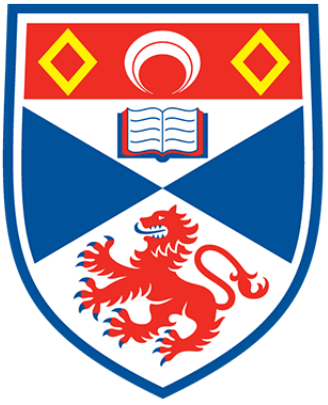

University of
St Andrews

CS4402 Constraint Programming

Constraint Solver Implementation

April 15, 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, the performance of the solver and different heuristics are tested empirically using various metrics.

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 ←selectVar(varList)
7:         val ←selectVal(domain(var))
8:         branchFCLeft(varList, var, val)
9:         branchFCRight(varList, var, val)
```

The small change in the algorithm is not to exit when the first solution is found, but to add that solution to a list and continue to find more solutions. This was done to allow all solutions could be found. To check for `completeAssignment()`, all variables in the problem instance contain 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.

```
1: procedure BRANCHFCLEFT(varList, var, val)
2:     assign(var, val)
3:     if reviseFutureArcs(varList, var) then
4:         ForwardChecking(varList - var)
5:     undoPruning()
6:     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, 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 variables 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 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 variable which acts as the key to an `UndoTracker`.

Algorithm 3 Branching right during forward checking.

```
1: procedure BRANCHFCRIGHT(varList, var, val)
2:     deleteVal(var, val)
3:     if ¬ empty(domain(var)) then
4:         if reviseFutureArcs(varList, var) then
5:             ForwardChecking(varList)
6:         undoPruning()
7:     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.

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 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.

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.

2.3 Maintain arc consistency (MAC)

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.

3.2 Metrics used

Multiple metrics were used to empirically measure the performance of different heuristics

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, but it is not used as a metric in the reported results. This is because of time benchmarking issues in Java due to the JVM [1] cause the time taken to be unreliable without a robust benchmarking framework.

To run the experiments, an `Experiment` class was developed that could take a list of heuristics

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.

4 Results and analysis

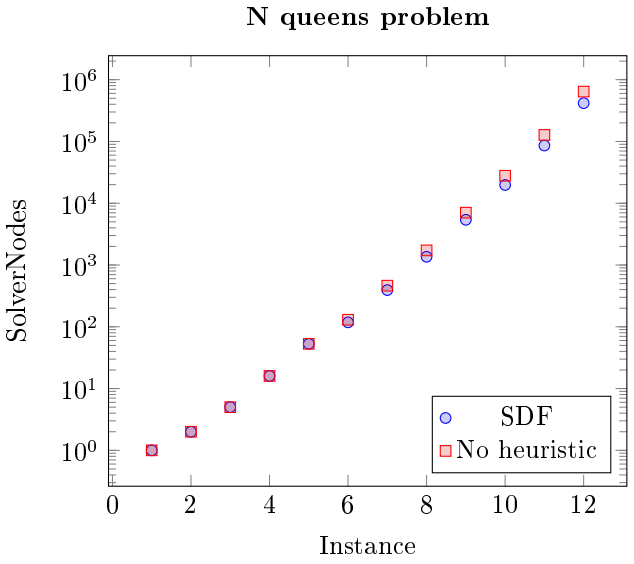

Figure 1:

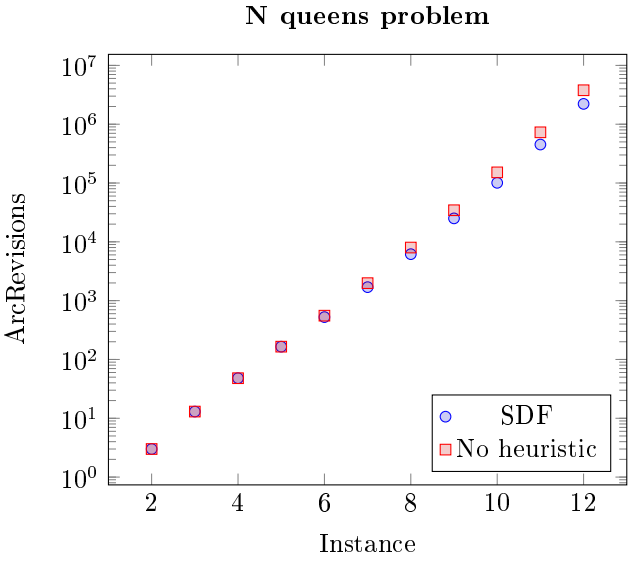

Figure 2:

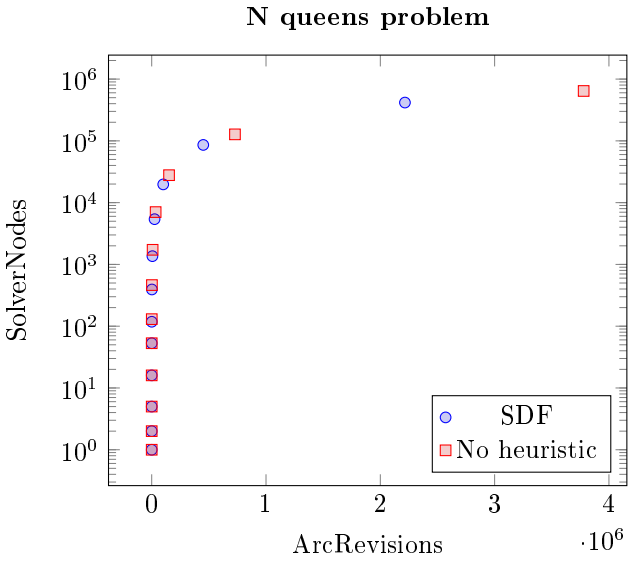

Figure 3:

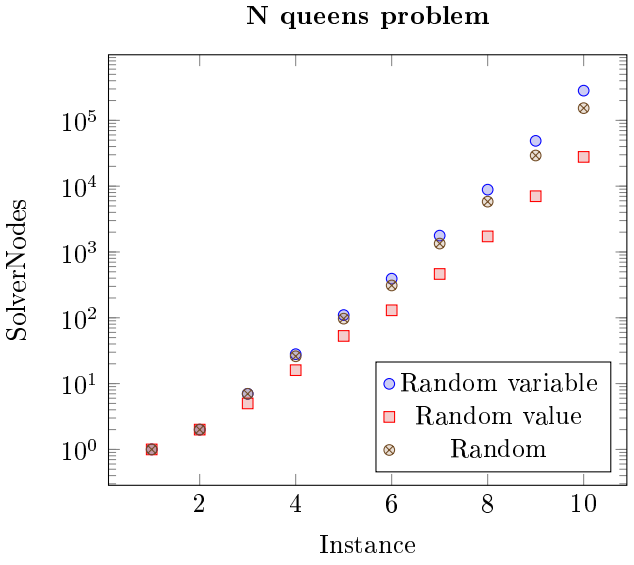

Figure 4:

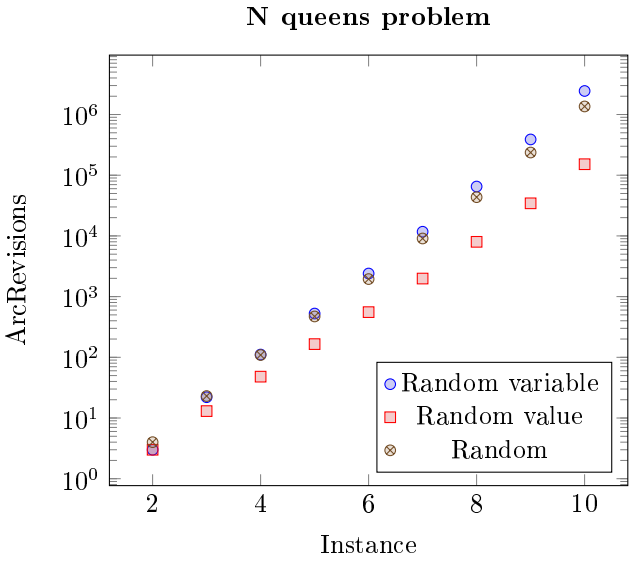

Figure 5:

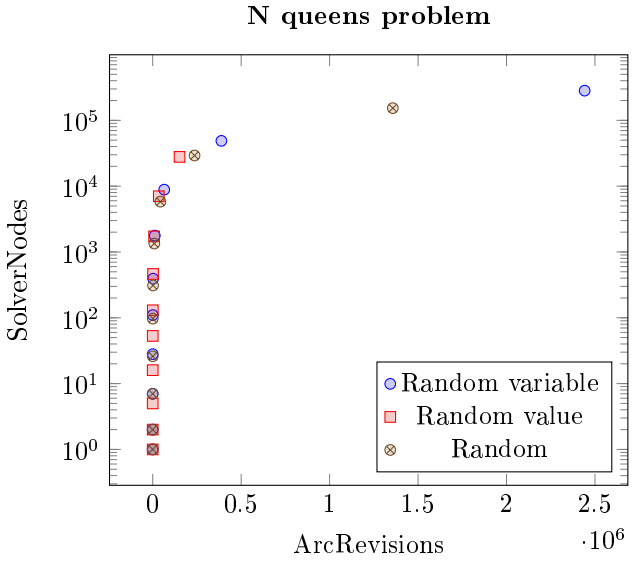

Figure 6:

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`.