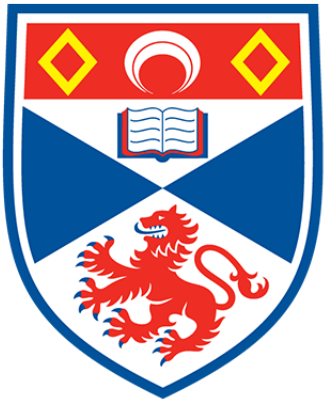

University of
St Andrews

CS4402 Constraint Programming

Constraint Solver Implementation

April 14, 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

2.1 2 way forward checking

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

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)
```

Algorithm 2 Algorithm for revising an arc.

```
   procedure Revise(Arc(var1, var2))
2:     val ←var1.assignedValue()
       constraints ←getArcConstraints(Arc(var1, var2))
4:     if empty(constraints) then return False
       else
6:         for each constraint c in constraints do
               for each value v in domain(var2) do
8:                 if ¬constraintHolds(c, v) then dropVal(var2, 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

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 the time taken metric. Although a robust benchmarking framework was not used, warm up iterations were added as part of the experiments due to benchmarking complications on the JVM [1].

The number of search nodes and revisions did not require multiple iterations because they are deterministic. However a "random" heuristic that random chooses the next variable to assign would require the extra runs to determine the average nodes and arc revisions.

4 Results and analysis

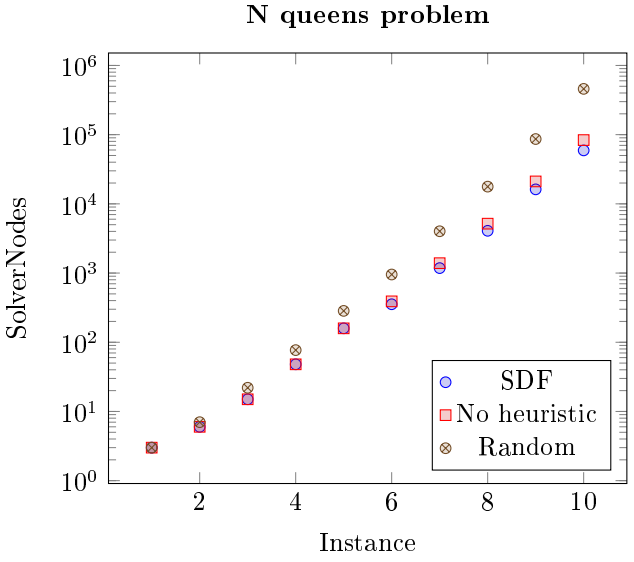

Figure 1:

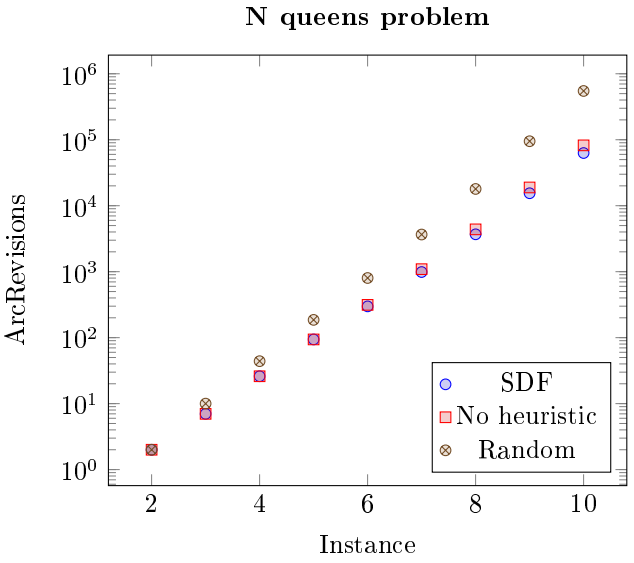

Figure 2:

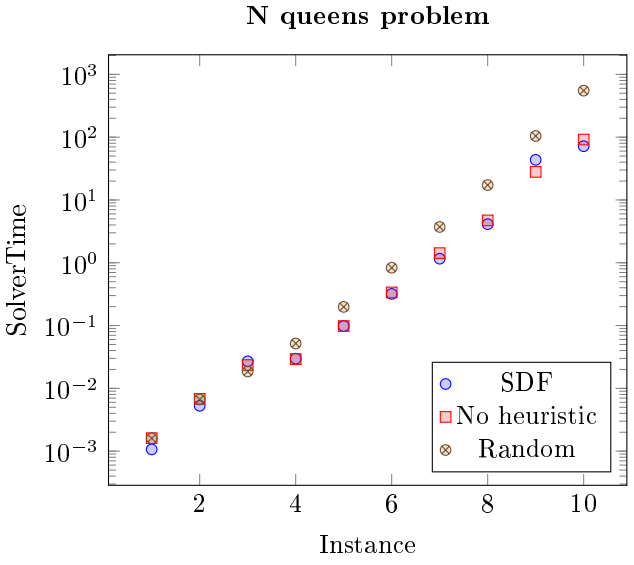

Figure 3:

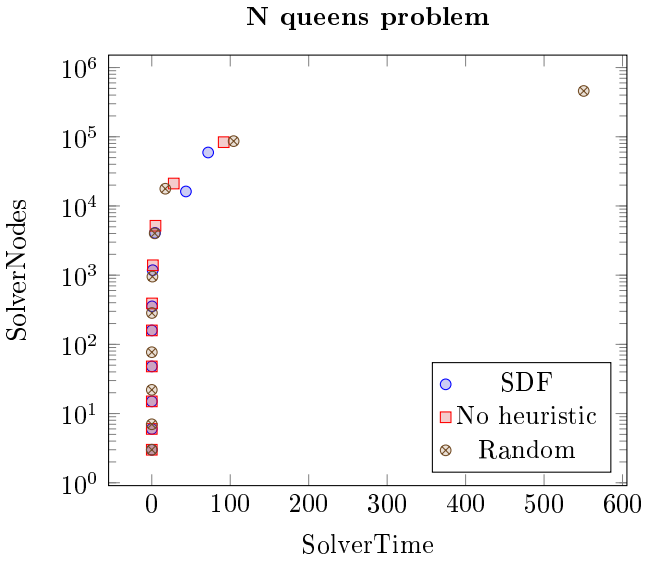

Figure 4:

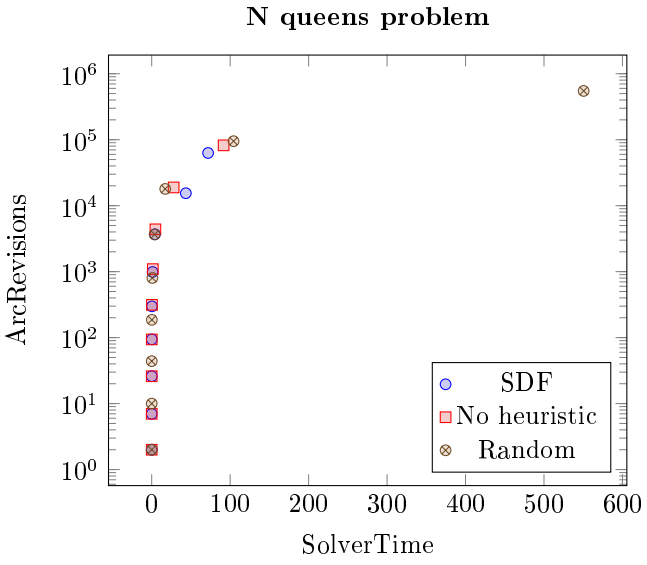

Figure 5:

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`.