University of
St Andrews

CS4402 Constraint Programming

Constraint Solver Implementation

April 18, 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 ←selectVar(varList)
7:         val ←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.

```
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 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.

```
1: procedure branchFCRight(varList, var, val)
2:     deleteVal(var, val)
3:     if $\neg$ 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. 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.

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.

The three metrics were all chosen for various reasons:

- **Solver nodes** - The number of solver nodes tells us how large of a search tree had to be explored in order to find the solutions. If different heuristics give different numbers of search nodes, it tells us that the heuristic had to do more or less searching by giving the size of the search tree. Because all solutions are found in our experiments, the search does have to be exhaustive. However, a more effective heuristic would prune more domain values earlier in forward checking and therefore have a smaller search tree with less nodes.
- **Arc revisions** - The reason the number of arc revisions was measured is similar to why solver nodes is measured. The number of arc revisions give an even more detailed look at how much work is being done at each node of the search tree. Had an algorithm like AC3 been implemented, the number of arc revisions would also help in differentiating the different between forward checking and maintaining arc consistency, as it would cost more revisions to maintain arc consistency, but search less nodes.
- **Time taken** - The time taken was chosen because ultimately an important aspect of solvers is how long it takes. A complicated algorithm could solver constraint satisfaction problems in fewer revisions and search nodes, but take a lot of time, so it is still valuable to measure the time needed.

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. Furthermore, because the time taken was also measured, the additional runs were needed to find an average time. Because of time benchmarking issues in Java due to the JVM [**jvm-benchmark**], 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

From running the solver on the various constraint satisfaction problems provided, the time taken, number of solver nodes and number or arc revisions were measured and compared. It made little sense to compare them in the same graph as each metric gave values on different scales, however, they are plotted beside each other to see if there is any difference between them. Additionally, the three metrics are plot against each other to show their relationship to each other. The graphs are all plot on log scales. As the size of an problem instance grows, the time, nodes and revisions taken grow exponentially, making it difficult to determine their relationship on a normal scale. Moreover, each of the three constraint satisfaction problems tested (N-queens, langfords and sudoku) are analysed separately. Because the problems are completely different problems, it made little sense to put them into the same graphs. However, connections are drawn between any similarities and differences between the problems.

4.1 N-queens problem

For the N-queens problem, we first look at how the time, nodes and revisions grow as the problem instance becomes larger (larger N). Comparing a few of the heuristics, we can see the largest domain first and random heuristic perform significantly worse compared to the other heuristics. This is expected and makes a lot of sense. The largest domain first heuristic is close to the worst case, as the variables with the most values left in its domain are always selected, making it more difficult to prune. This can be seen to be worse than randomly choosing on every step, which already performs poorly in comparison to all other heuristics. The random heuristic also performs poorly as expected as it contains no pattern of selecting variables in a smart way.

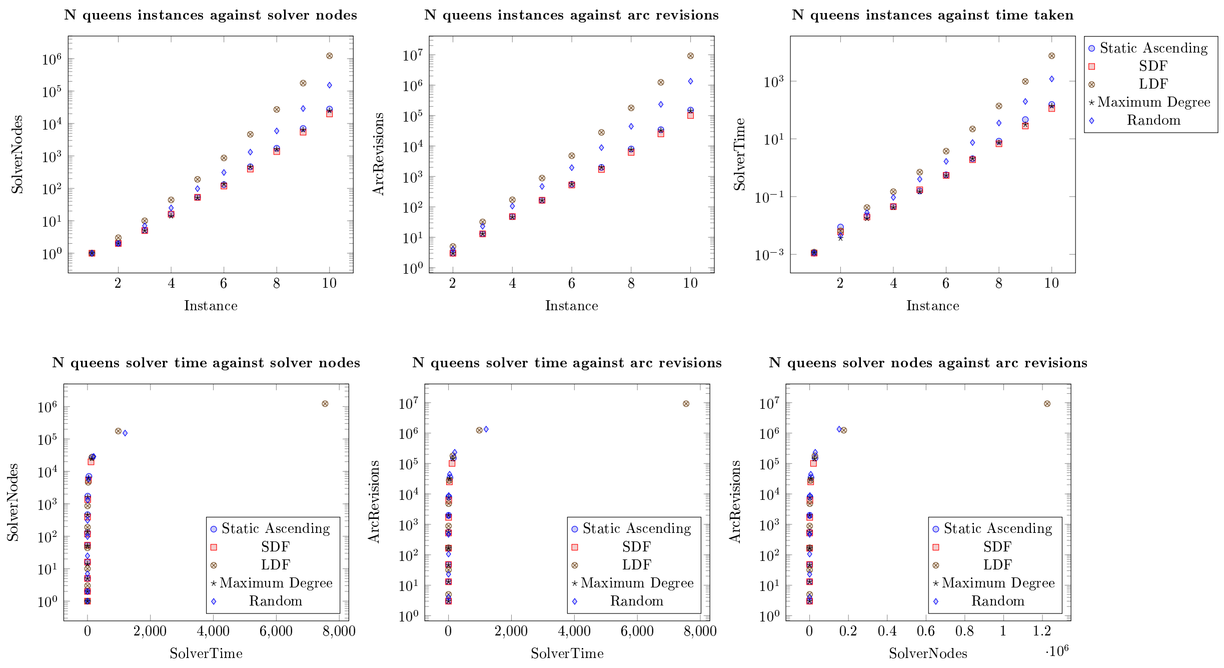


Figure 1: N queens problem performance with different metrics and heuristics.

The log graphs here show that though there are small differences between the 3 metrics, they are all comparable to each other, as we get a log curve on the log scale.

Next we can observe the heuristics in more detail by looking at one specific N-queens problem instance. The static ascending and descending heuristics perform very similarly while smallest domain first and maximum degree heuristic perform better. This was again expected as both SDF and maximum degree take into account the difficulty of the problem. SDF performs better as it adjusts dynamically during exploration while the ordering for maximum degree was determined at the beginning. We can still see that a clever static ordering with maximum degree was able to achieve better performance than a simple ascending/descending ordering. Further, static orderings that are not very smart such as alternating odd/even variables and minimum degree perform worse. It is slightly surprising there is so little difference between the three metrics for the different heuristics. The best heuristic performs the best for all three metrics and vice versa for the worst performing heuristic. Though this shows the heuristics performing consistently, it should be possible that some heuristics solve less nodes with more time if a lot of extra work has to be done. Because the heuristics used here are not very complicated, not much difference between the three metrics can be seen.

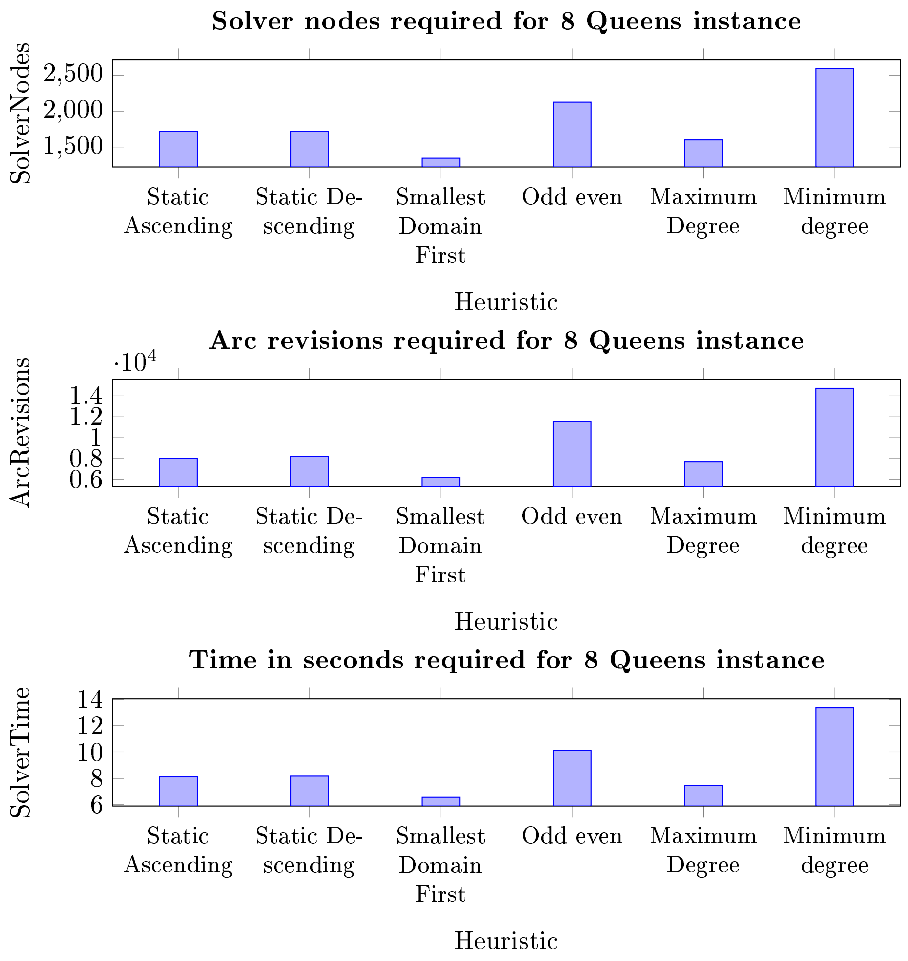


Figure 2: Performance of different heuristics on the 8-Queens problem. Heuristics that performance significantly worse such as largest domain first and random and excluded here.

It was briefly touched on earlier than the random heuristic performed worse than all other heuristics, but not as badly as largest domain first. What is interesting to find out next is if there is a big difference between different random heuristics. This will further show which aspect of the heuristic is of more importance as different specific aspects are randomised.

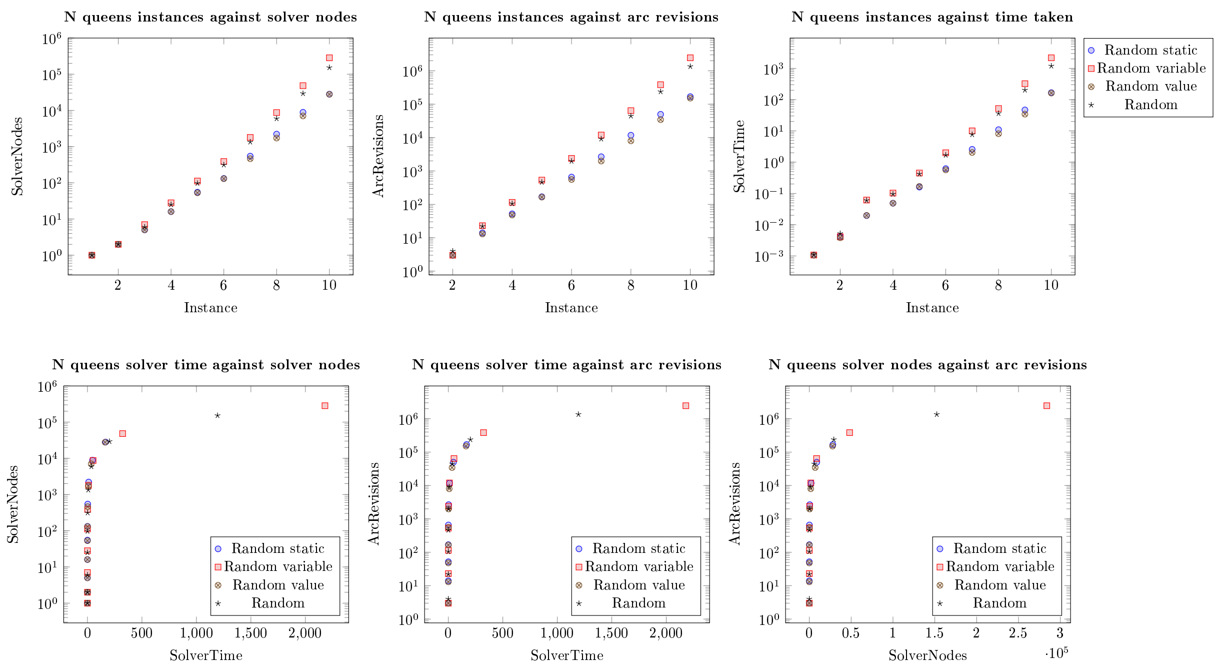


Figure 3: N queens problem performance of random heuristics

From the four random heuristics tested, it can be seen clearly that the completely random heuristic and random variable heuristic perform worse. This shows two things:

1. Variable ordering is more important than value ordering in general - This is shown from the improved performance of random value ordering compared to the other three random heuristics, all of which included random variable ordering. The random value ordering was simply a static ascending order and its performance was not heavily impacted by the random value selection, as it took almost the same number of solver nodes and arc revisions as the completely static ascending heuristic.

2. Having some form of order, even if a random order is better than always choosing randomly - By using a random static order, the same order is kept throughout solving the problem. The metrics measured for this approach is closely comparable to the odd even variable heuristic, showing how the two are almost equivalent, even from mixing up the order of the variables differently. Of course this may not apply to every problem, especially if a problem took advantage of an odd/even ordering, however, if it doesn't matter then a static random ordering is just as good.

4.2 Langfords

For langfords problem, multiple runs were not able to be completed easily due to the difficulty of the harder langfords (n = 3) problems taking too much time. As such, the time taken and random heuristics could not be measured for langfords problems where n $\neq$ 2. Because the langfords problems for n = 3 are more difficult problems that take more search nodes and revisions, it is interesting to see how the heuristics are affected when the problems are more complex.

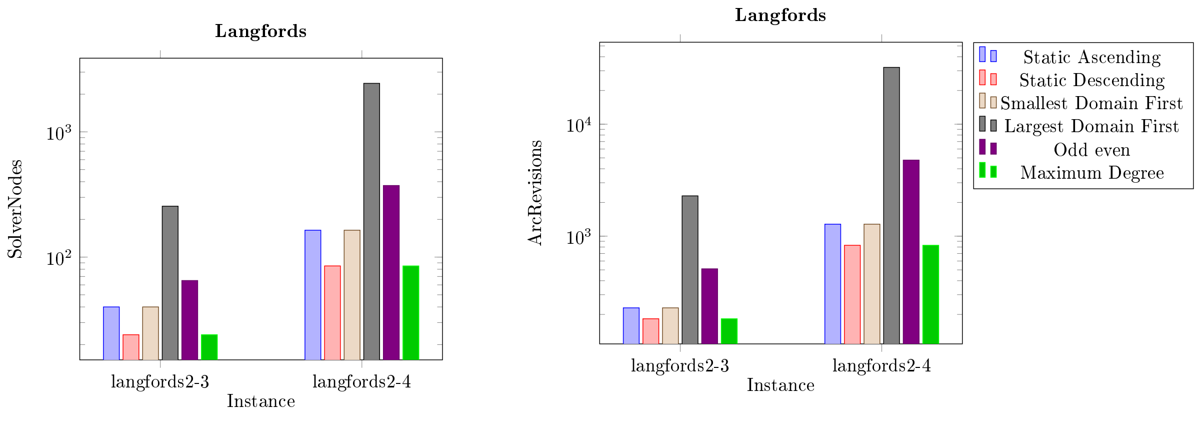


Figure 4: Langfords problem with different metrics and heuristics.

For the smaller langfords problem, there is different pattern seen between the different heuristics. The best heuristic is a static descending order and the maximum degree ordering. This is different from the N-queens problem, where smallest domain first performed much better than all other heuristics. From this result, we can see that it is not always the case that dynamic heuristics like smallest domain first perform the best and static heuristics can be better depending on the problem and problem size. Here, the descending static order is able to perform well because the last variable is the hardest to search for. This is due to the properties of langford's problem, where the arrangement of the digits is important. Finding what the last digit is greatly helps reduce the domains needed to search which is why descending works well. Maximum degree is able to find the same performance because the last variable is the most constrained and so it also solves in descending order. Smallest domain first alone does not help as each variable has the same domain and the heuristic cannot infer any more information that can help it improve.

The graphs of the random heuristics show a similar pattern to the N-queens problem. The variable assignments can be seen to have a greater impact, as fully random and random variable ordering is much worse than a random static and random value ordering.

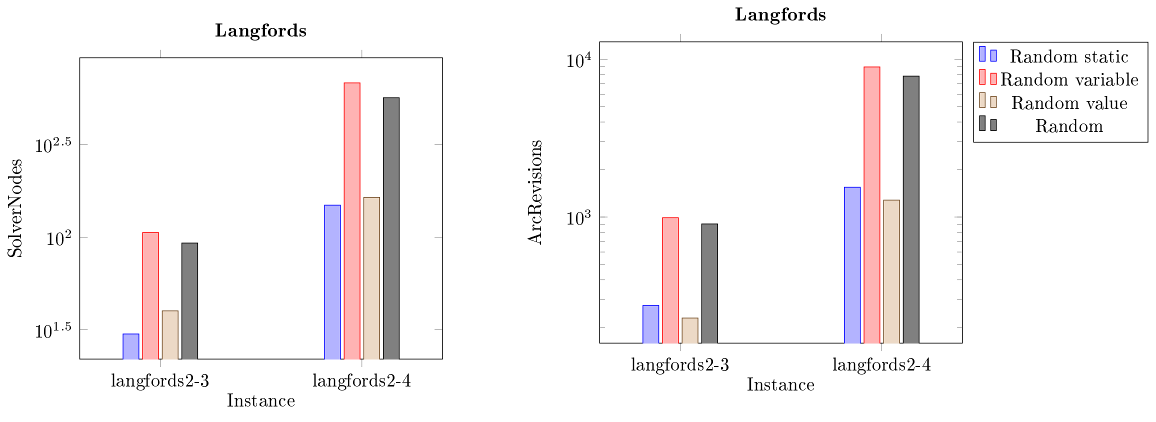


Figure 5: Langfords problem with random heuristics.

The data for langfords of n = 3 further supports the fact that the descending static ordering performs better. However, running the maximum degree heuristic does not finish in time, which is strange. We can see the static descending order is significantly better than smallest domain first and ascending static order. The time taken is not very reliable as only one run was done, but the difference can clearly be seen. The strange issue is why the maximum degree heuristic was not able to finish in time. Perhaps this had to do with memory issues, or overhead before the solver started. However this does show that even if the heuristic should theoretically perform better, it is not always the case in practice due to other forms of overhead.

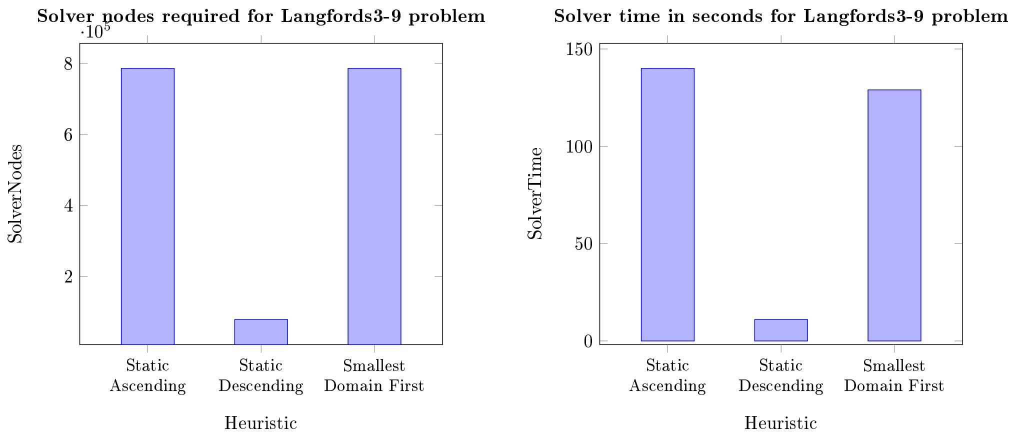


Figure 6: Comparison of heuristics for Langfords3-9, showing the significant improvement of a static descending heuristic.

4.3 Sudoku

The two sudoku problems (SimonisSudoku and FinnishSudoku) are two of the harder problems to solve. In particular, many heuristics could not solve the FinnishSudoku problem in any reasonable amount of time, long enough that little comparison could have been made. Like the Langford's problem, because of the time needed to solve the sudoku problems, multiple runs for experimentation could not be done, and so the random heuristics and time taken are not analysed. For FinnishSudoku, no heuristic except for smallest domain first was able to finish in reasonable time which shows the strength in that particular heuristic.

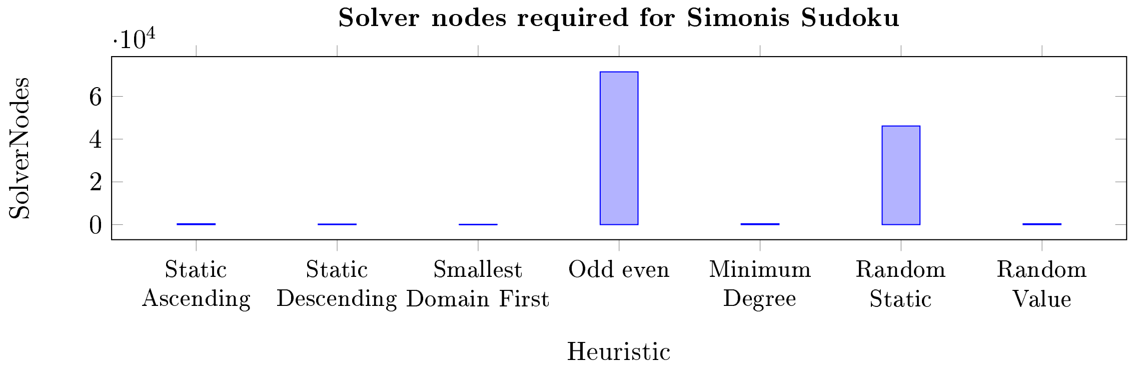
Furthermore, the following heuristics did not complete the SimonisSudoku in any reasonable time:

- Fully random and random variable heuristics
- Maximum degree ordering
- Largest domain first

Perhaps for the sudoku problem, the least constrained variables are better suited to being solved first, which would explain why the maximum degree could not finish when the minimum degree ordering finished quite quickly.

Simonis Sudoku

In the graphs for the SimonisSudoku problem instance, it can be seen that the odd/even static ordering took considerably more nodes and arc revisions than all other heuristics. This is due to the way the sudoku problem is modelled. Solving all odd then even variables makes it difficult to solve entire rows and columns. In fact no full row or column will be completed until all odd variables are assigned, as the odd even heuristic will go through the variables in a diagonal pattern, causing a lot of backtracking and wipeouts. Further, the random static heuristic also performed poorly, as its order was random, meaning some rows and columns may not be solved or finished until much later. In comparison, all the other methods which stuck to a ascending or descending order got much better results.



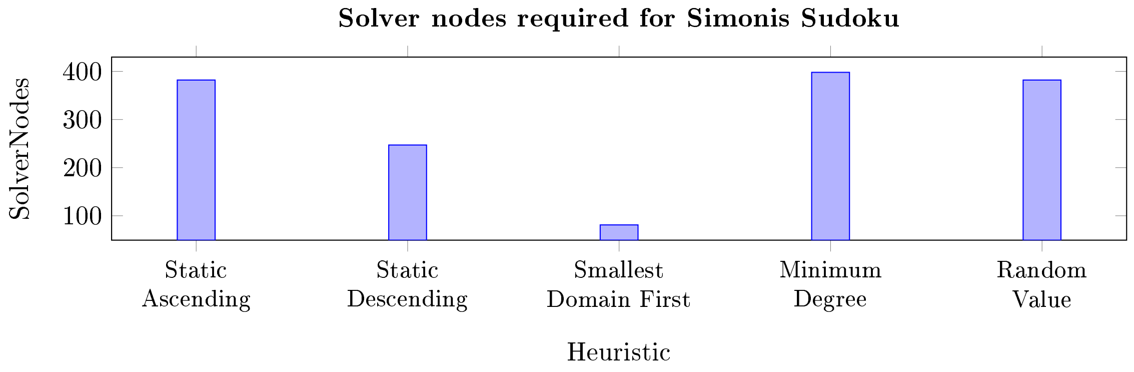
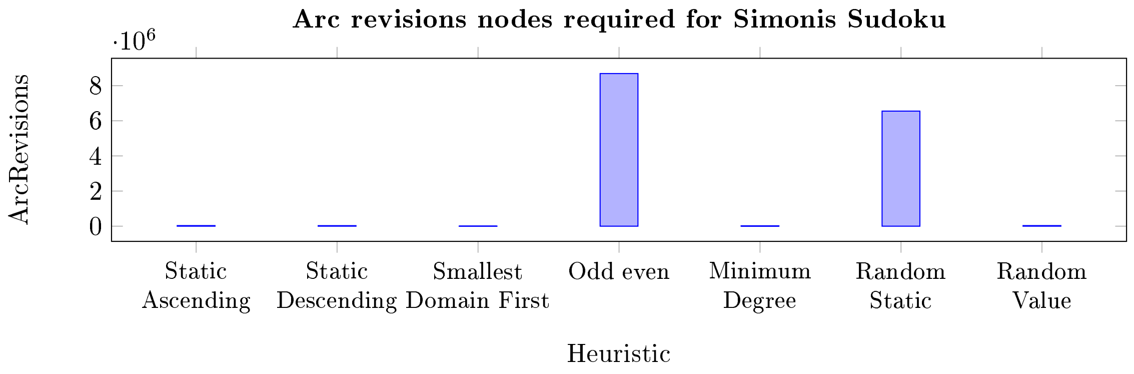


Figure 7: Solver nodes for Simonis Sudoku. The second graph does not show the odd even and random static heuristics for better comparison of the other five heuristics.



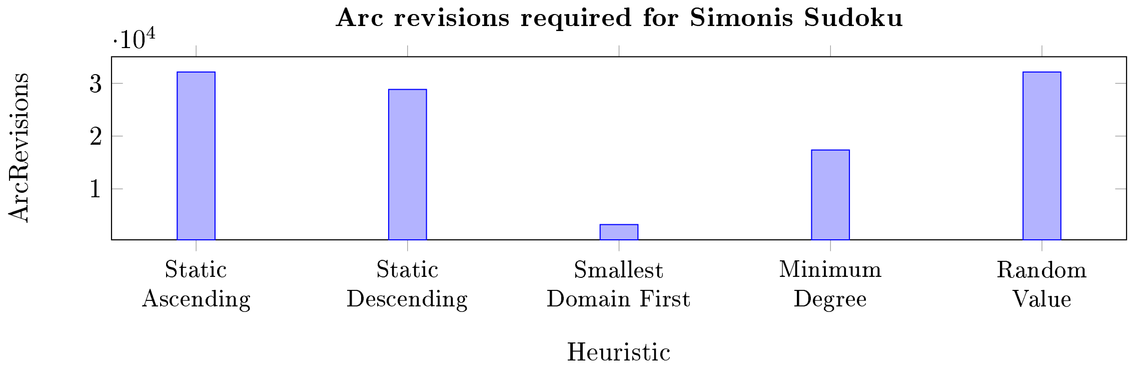


Figure 8: Arc revisions for Simonis Sudoku. The second graph does not show the odd even heuristic for better comparison of the other four heuristics.

There are two more interesting points in the results for Simonis Sudoku. First, smallest domain first used the least number of revisions and solver nodes. This is similar to the N-queens problem, but contrasts with Langford's problem. This shows us that the performance of static heuristics can be very problem dependent, more so than dynamic heuristics. The is especially true the more specific a static heuristic is to a problem class, or even to a problem instance, as a static ordering can give the exact ordering of variables needed that gives the optimal search. Second, the minimum degree ordering gives more solver nodes relative to the other heuristics, but less arc revisions. This is the first case in our results where the number of nodes and revisions do not tell the exact same results. This is interesting as it means the minimum degree heuristic has a lower arc revision to search node ratio, so less revisions are done on average per branch, but more branching is done in general. This is good as it shows just looking at revisions or solver nodes it not necessarily a good enough indicator of performance. Moreover, the means the minimum degree heuristic has pruned less values, becomes more nodes had to be searched, but the additional search nodes did not require much revision.

5 Conclusion

In conclusion, a forward checking constraint solver was implemented. The solver was implemented to be able to take in any generic heuristic for variable and value section. Tests were written on the provided constraint satisfaction problem instances to ensure the correct number of all solutions could be found, verifying the correctness of the solver.

Next, different heuristics were implemented for the solver to use, which included a dynamic variable heuristic (smallest domain first) and static variable heuristics such as maximum degree. Further, random heuristics were tested to show the different affect of variable and value selection.

Finally, the heuristics were run on the various different problems where the time taken, number of solver nodes and number of arc revisions were measured to show the difference between the heuristics, especially the difference between a dynamic heuristic like smallest domain first, and a static heuristic like maximum degree. When looking at the N-queens problem, it appeared that static heuristics did not perform as well as dynamic heuristics. However, the Langfords' problems showed that this was not always the case, as a descending static order was able to perform significantly better than a smallest domain first heuristic. This demonstrated that a good static order built for specific problems could perform very well, but would be highly problem dependent while dynamic heuristics like smallest domain first could work well in most general cases. TODO sudoku