CS4402 Practical 2: Binary CSP Solvers

Creating Two Binary Constraint Satisfaction Problem Solvers

190018469

University of
St Andrews

Computer Science
University of St Andrews
November 2023

1 Introduction

The specification detailed the creation of two binary constraint satisfaction problem solvers, one using Forward Checking (FC) and the other Maintaining Arc Consistency (MAC). Both solvers employ 2-way branching and were augmented with two variable and value ordering strategies to choose from. The performance of the solvers and selection strategies were evaluated against instances of Langford's problem, the n-queens problem and Sudoku. Finally, it is observed how solver performances varies among different instances of the three problems.

Usage Instructions

To compile the solvers, run the makefile provided with the submission. After this, the solvers can be run with the following command:

```
java BinaryCSPSolver <file.csp> [solverType] [solutionsToFind] [varSelectMode] [valSelectMode] [debugMode]

file.csp: The path to a problem instance to solve.
solverType: The solver type to use (MAC / FC).
solutionsToFind (Optional): The number of solutions to find before stopping. 0 will attempt to find all solutions.
varSelectMode (Optional): The mode to use when selecting a variable to assign (0 = Ascending, 1 = Min Domain).
valSelectMode (Optional): The mode to use when selecting a value to assign to a variable (0 = Ascending, 1 = Min Conflicts).
debugMode (Optional): Whether to log additional information to show each step taken by the solver (True / False). Useful for debugging and full understanding.
```

Listing 1: Usage Instructions

2 Design

2.1 Solver Overview

An abstract solver class was designed to hold functions and variables that both forward checking and maintaining arc consistency use. This includes a basic solving framework, support for state management, assigning and unassigning variables and values, generating and revising arcs as well as printing solver information. The abstract class is extended to create the full forward checking and maintaining arc consistency solvers.

2.2 Representing constraint satisfaction problems

Code was supplied to allow the solver to read in a constraint satisfaction problem. This code was extended and holds information on the constraints of a problem, the domain of each variable and the variables left to assign. The data structures chosen for the variables list and domains were LinkedList and TreeSet. Initially, LinkedHashSets were used instead. However, while they did provide increased performance overall, the alternative data structures were required to support ascending order choice.

2.3 Solving Framework

The maintaining arc consistency and forward checking algorithms largely share the same solving framework. In 2-way branching, a variable and value are selected to assign, followed by propagation and recursive selection of additional assignments. This is known as the left branch. If an assignment fails, then it is "unassigned", making the opposite choice before propagation and recursion as before. This is known as the right branch. Together, these left and right branches are what make the solver be 2-way branching. The key difference between the two algorithms lies in their propagation steps, or more precisely how they enforce local consistency. Forward checking checks arcs to the current variable, while maintaining arc consistency checks all arcs linked with a variable. Furthermore, MAC also enforces global arc consistency before making the first assignment. To offer a solving framework while supporting these differences, the core `BinaryCSPSolver` class holds `solve` and `recursiveStep` methods that call the abstract `prepareSolver` and `enforceLocalConsistency` methods. An image showing how forward checking solves 4-queens is shown in Figure 1.

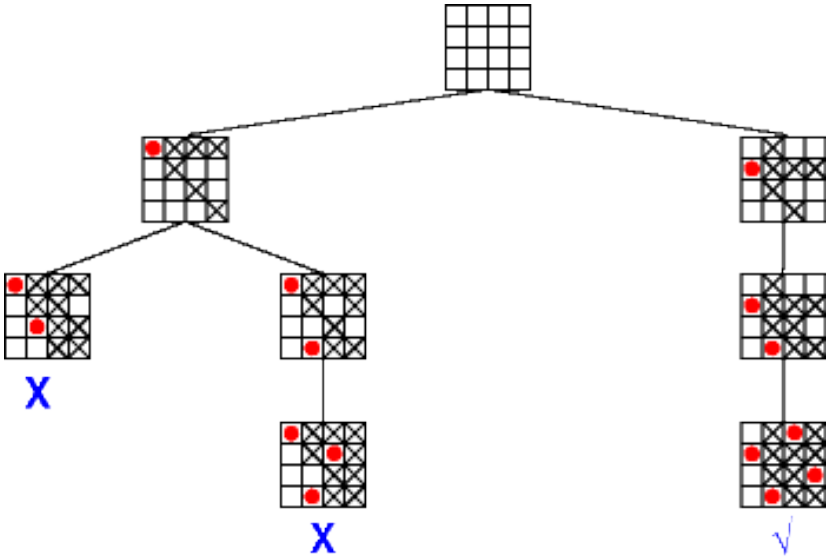


Figure 1: Forward Checking Solving 4-Queens [1]

2.4 State management

Whenever an assignment is made, a new state is entered, pushing a state change to the stack of all state changes. This records the assigned variable and any domain and constraint tuple pruning performed. When an assignment is undone, this information is used to restore the state of the solver.

2.5 Assigning and unassigning

For a given variable, an assignment prunes all the other values not chosen. Furthermore, any constraints that rely on the pruned values are also removed. An unassignment performs the opposite, reverting to the previous state before pruning only the value that was previously assigned.

2.6 Selection strategies

The solver supports two variable and value selection strategies. Variables can be chosen in ascending order or by smallest domain. Similarly, values can be chosen in ascending order or by minimum conflicts. Ascending order chooses the first value each time a choice is made. Restoring supports ascending order by replacing variables at the head of the LinkedList and using sorted TreeSets for domains. The smallest domain choice looks through the domain sizes of all variables using a simple minimum algorithm to return the variable with the smallest domain.

The minimum conflicts choice generates Geelen pairs for each possible domain value. This records the number of other domain values left and lost upon making a certain choice. After each pair is generated, another simple minimum algorithm can be used to return the value causing the fewest conflicts.

2.7 Generating and revising arcs

The core solver holds code to generate arc pairs for a specific variable. This involves searching through constraints and generating an arc pair for each constraint that contains the variable. Arc revision goes through each value in the domain of the arc's first variable, checking whether it has support in the domain of the arc's second variable. This is done by checking the constraint for a tuple that matches the two value choices. The fact that each constraint is only listed once in the table of constraints in ascending order provides additional challenges. Extra care must be taken when checking constraints as variables on the left side of an arc and in assignments are not always on the left side of a constraint. In these cases, it is checked whether the constraint should be checked in reversed order or not.

2.8 Tracking and printing solver information

The core solver also contains variables to track the number of solutions found, nodes explored and revisions done. These are updated when printing a solution, performing an assignment or unassignment and making a revision respectively. Once the solver has finished, this information can be printed to analyse its performance. Each specialised solver's main method also contains a variable to track the time taken.

3 Testing

3.1 Debug Logging

The solver can be run with an optional argument to switch on debug mode. This prints solver logic to standard output, showing each step as it is taken. This can be used to analyse the solver's behaviour and verify that each step is as expected. Furthermore, it can be useful to gaining a full understanding of each part of the algorithm and to spot the differences between MAC and FC.

```
 ws/Practical\ 2_310c49b2/bin BinaryCSPMACSolver instances/n-queens/small/4Queens.csp MAC 0 0 0 True
Set var 0 = 0
EmptyDomainException: Domain wipeout when pruning domain! (1)
Set var 0 != 0
Set var 0 = 1
Set var 1 = 3
Set var 2 = 0
Set var 3 = 2
Found solution!
1
3
0
2

EmptyDomainException: Domain wipeout when pruning domain! (2)
EmptyDomainException: Domain wipeout when pruning domain! (2)
EmptyDomainException: Domain wipeout when pruning domain! (2)
Set var 0 != 1
Set var 0 = 2
Set var 1 = 0
Set var 2 = 3
Set var 3 = 1
Found solution!
2
0
3
1

EmptyDomainException: Domain wipeout when pruning domain! (2)
EmptyDomainException: Domain wipeout when pruning domain! (2)
EmptyDomainException: Domain wipeout when pruning domain! (2)
Set var 0 != 2
EmptyDomainException: Domain wipeout when pruning domain! (2)
Found 2 solutions!
Explored 18 nodes!
Performed 80 arc revisions!
Time taken: 15ms
```

Figure 2: Debug Mode in MAC 4-Queens

3.2 Breakpoint Testing

Visual Studio Code comes with support for breakpoint testing. This allowed points to be placed in the code at which the solver would pause. During this pause, the state of each data structure in the solver could be checked, showing the state of the solver more precisely than the debug logging.

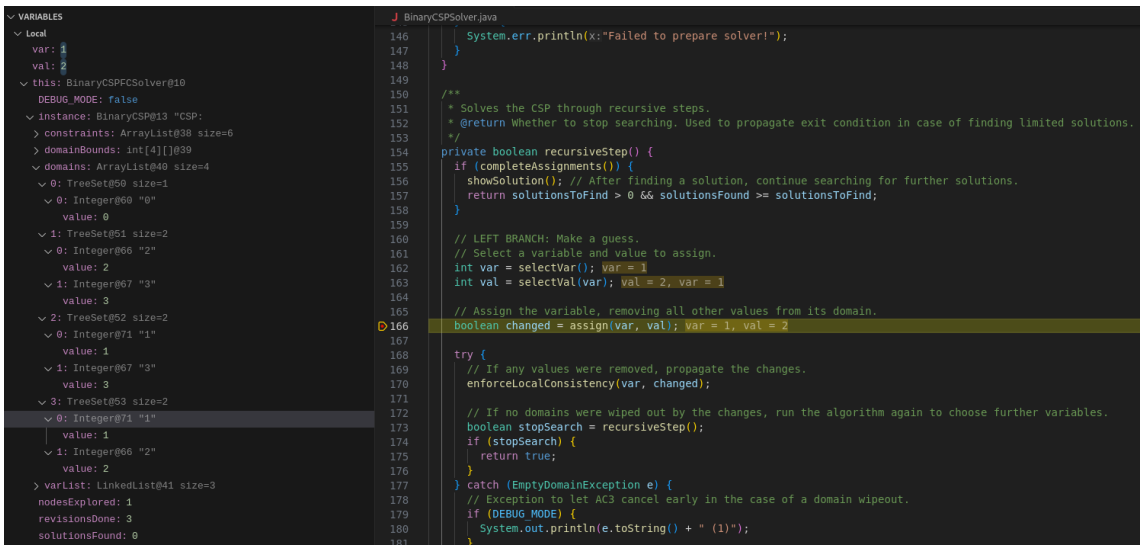
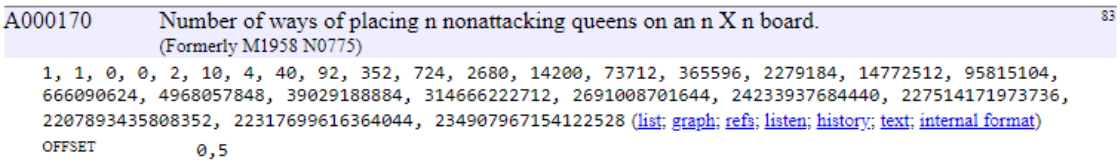


Figure 3: Breakpoint Testing in FC 4-Queens. Identical state to Figure 6.

3.3 Accuracy Verification

The accuracy of the solvers was tested by comparing solver output to given solutions. For example, the number of solutions found for a range of n-queens problem was compared to the known number of solutions as shown in Figure 4 [2]. Similarly, the output from the solvers for Sudoku found online could be compared to the given solution to ensure they match 5. One tool that was instrumental in testing the solver

during development was the BrainBashers website [3]. This contains an interactive version of the n-queens puzzle as shown in Figure 6, which allowed for a visual representation of the expected solver state after each assignment.


A000170 Number of ways of placing n nonattacking queens on an n X n board. 83
(Formerly M1958 N0775)

1, 1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, 73712, 365596, 2279184, 14772512, 95815104, 666090624, 4968057848, 39029188884, 314666222712, 2691008701644, 24233937684440, 227514171973736, 2207893435808352, 22317699616364044, 234907967154122528 (list; graph; refs; listen; history; text; internal format)

OFFSET 0,5

Figure 4: N-Queens Solutions Sequence

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 5: Sample Sudoku with Solution

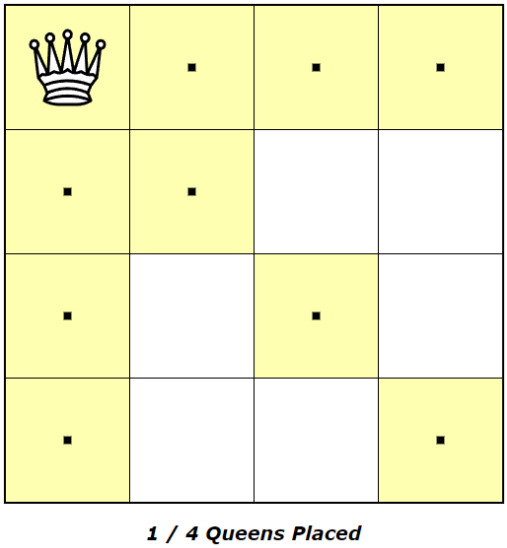
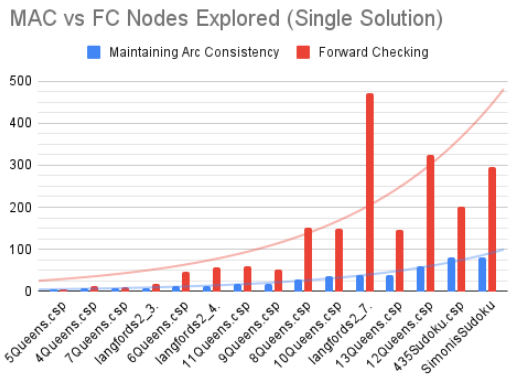


Figure 6: BrainBashers Interactive 4 Queens Puzzle

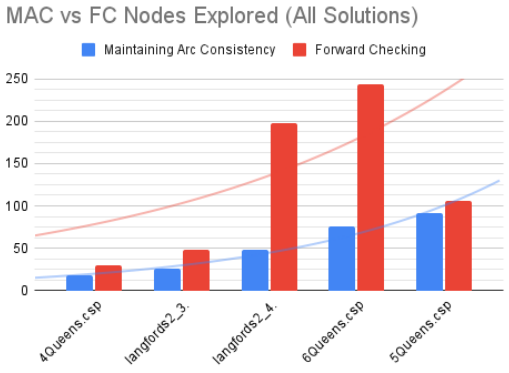
4 Evaluation

4.1 Solver Types

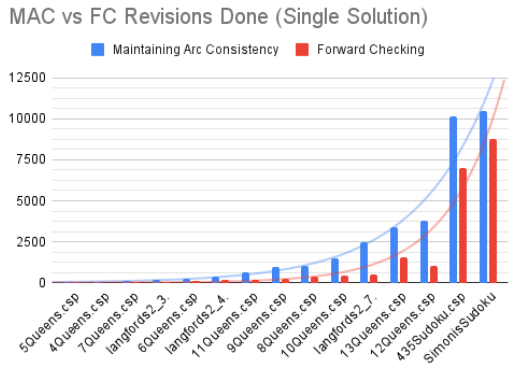
MAC was found to consistently explore fewer nodes than FC as seen in Figures 7(a) and 7(b). This is a result of the additional revisions performed in AC3 removing additional fruitless parts of the search tree, making pruning more effective. As a result of using AC3, MAC consistently performs more revisions than forward checking as seen in Figures 7(c) and 7(d). The reduction in time taken from exploring fewer nodes generally exceeds the additional time from making more revisions when running MAC, giving it a faster performance than FC overall as seen in Figures 7(e) and 7(f). However, for sufficiently small problems, FC may on occasion have greater performance as seen in the result for `langford2_3` and `8Queens` in 7(e), as well as `langford2_4` and `5Queens` in 7(f).



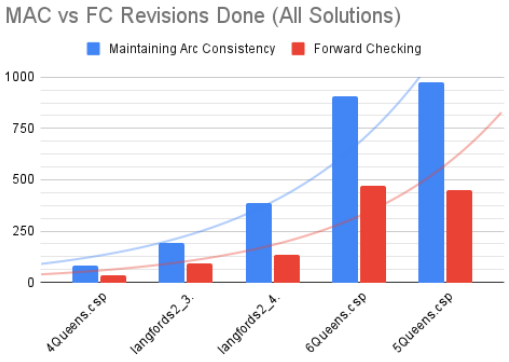
(a) Nodes Explored (Single Solution)



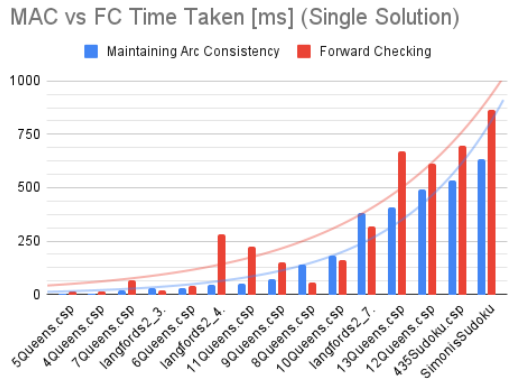
(b) Nodes Explored (All Solutions)



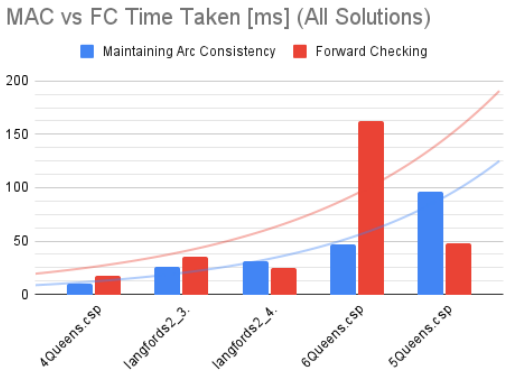
(c) Revisions Done (Single Solution)



(d) Revisions Done (All Solutions)



(e) Time Taken (Single Solution)



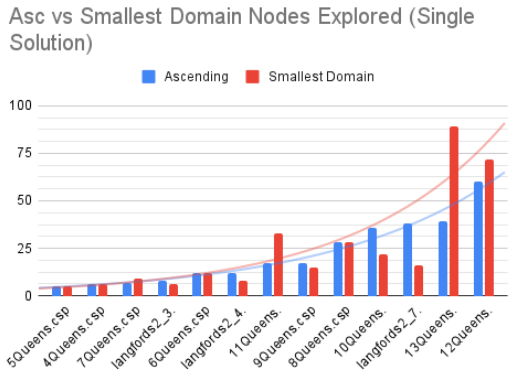
(f) Time Taken (All Solutions)

Figure 7: Maintaining Arc Consistency vs Forward Checking

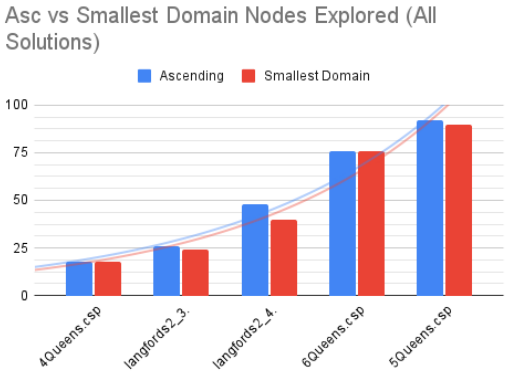
4.2 Variable Selection Strategies

Using the smallest domain heuristic gave mixed results when finding a single solution and slightly positive results when finding all solutions as seen in 8. In some cases, such as the 15-queens instance, using this heuristic resulted in a tenfold decrease in the number of nodes explored and revisions done. The time taken was decreased by

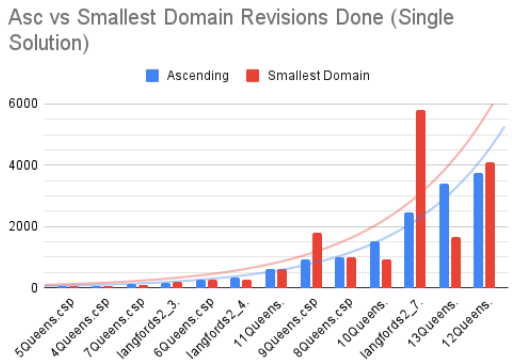
roughly half. This lesser, but still significant decrease, is likely due to the additional cost of having to check the domain size of each variable left to assign before choosing it. However, the opposite can be observed too. In Figure 8(a) smallest domain is shown to double the nodes explored in `13Queens`, while in Figure 8(c) it results in double the number of revisions for `langfords2_7`.



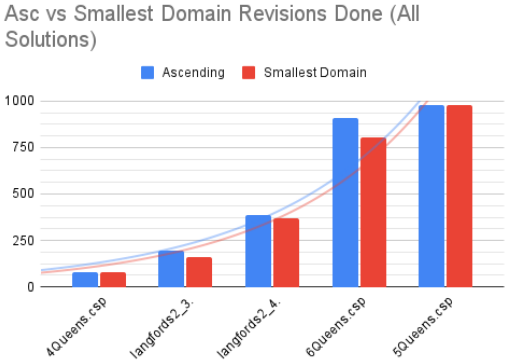
(a) Nodes Explored (Single Solution)



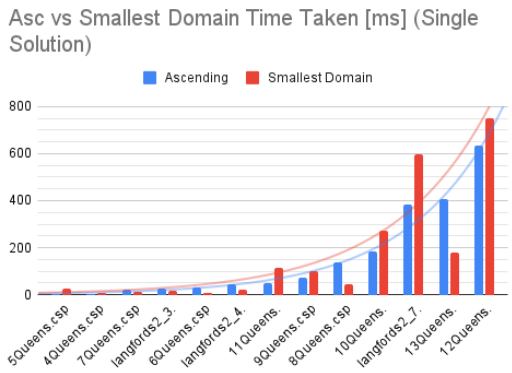
(b) Nodes Explored (All Solutions)



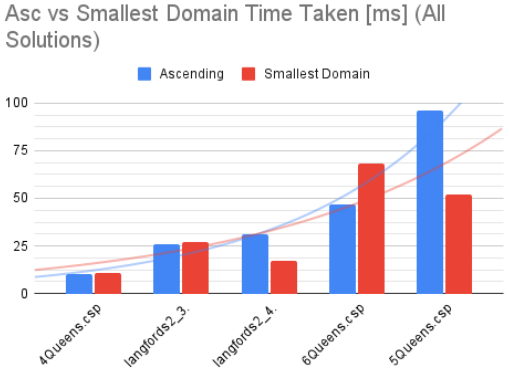
(c) Revisions Done (Single Solution)



(d) Revisions Done (All Solutions)



(e) Time Taken (Single Solution)

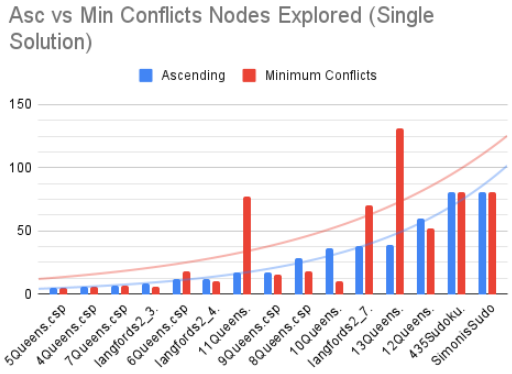


(f) Time Taken (All Solutions)

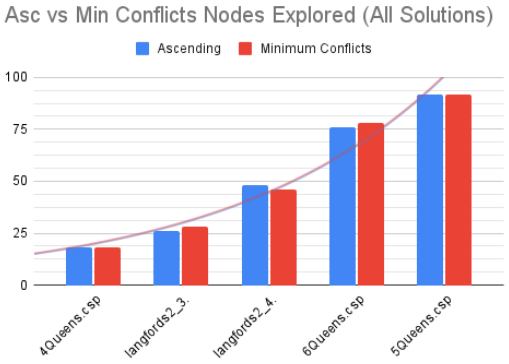
Figure 8: Ascending vs Smallest Domain Variable Ordering

4.3 Value Selection Strategies

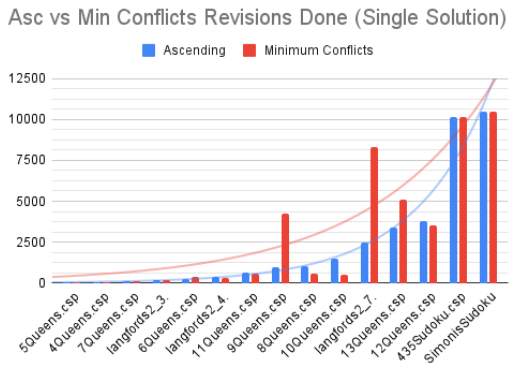
The minimum conflicts heuristic also gave mixed results. In some instances, such as `8Queens` and `10Queens`, the heuristic resulted in a significant decrease in nodes explored and revisions done as seen in 9(a) and 9(c). However, as seen in 9(e), `10Queens` still requires more time despite these savings. Despite the noticeably worse results in smaller problems, the trendlines seen in Figure 9 suggest that larger problems benefit with time saved compared to ascending order.



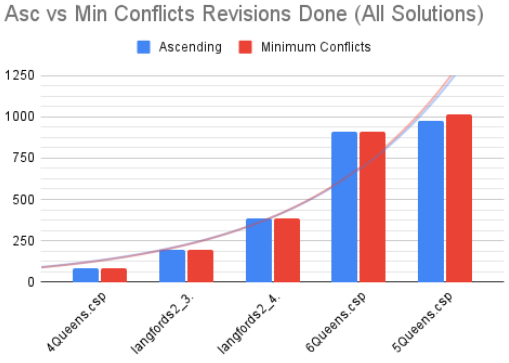
(a) Nodes Explored (Single Solution)



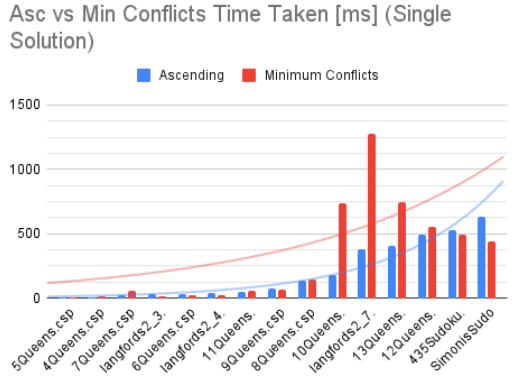
(b) Nodes Explored (All Solutions)



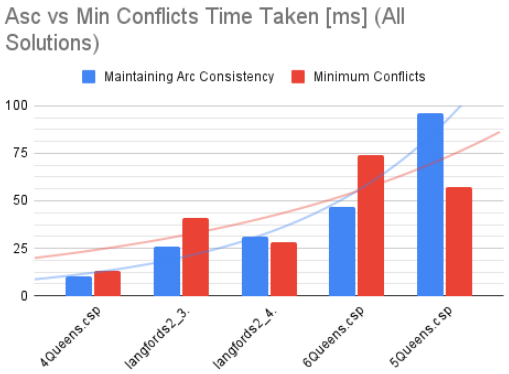
(c) Revisions Done (Single Solution)



(d) Revisions Done (All Solutions)



(e) Time Taken (Single Solution)



(f) Time Taken (All Solutions)

Figure 9: Ascending vs Minimum Conflicts Value Ordering

4.4 Problems

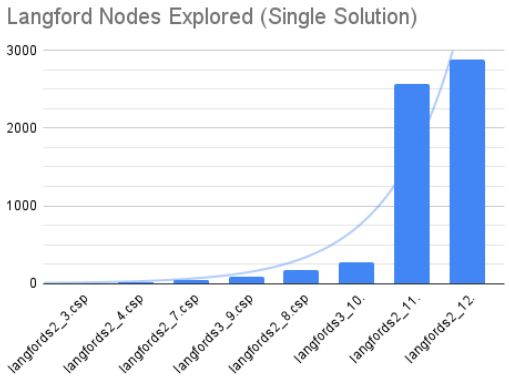
The performance of the MAC solver at tackling all problems with the smallest domain variable ordering heuristic and ascending order value ordering is shown in Figure 10.

Instance	Solver Type	Solutions To Find	Variable Ordering	Value Ordering	Solutions Found	Nodes Explored	Revisions Done	Time Taken
10Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	22	929	115
11Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	33	1791	270
12Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	72	4086	752
13Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	89	5798	596
14Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	50	4252	740
15Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	17	1873	430
16Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	24	2666	403
17Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	27	3884	430
186Sudoku.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	81	10126	713
18Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	46	5816	668
198Sudoku.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	83	12253	744
435Sudoku.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	81	10127	514
489Sudoku.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	105	16206	837
4Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	6	48	25
5Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	5	66	8
6Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	12	258	19
7Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	9	200	13
8Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	28	1000	99
9Queens.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	15	606	20
langfords2_11.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	54	11334	820
langfords2_12.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	42	10737	707
langfords2_3.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	6	104	8
langfords2_4.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	8	276	44
langfords2_7.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	16	1664	181
langfords2_8.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	88	7370	438
langfords3_10.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	2960	856730	4403
langfords3_9.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	813	224796	2072
SimonisSudoku.csp	MAC	1	SMALLEST_DOMAIN	ASCENDING	1	81	10488	678

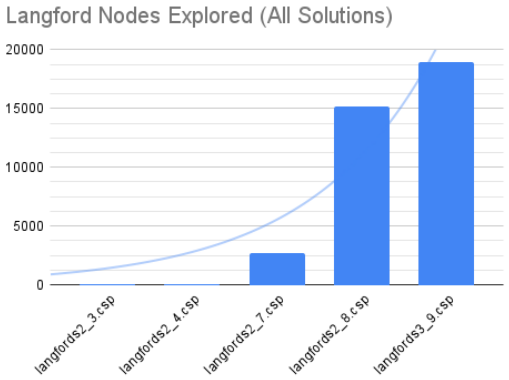
Figure 10: All Instances Data

4.4.1 Langford's Problem

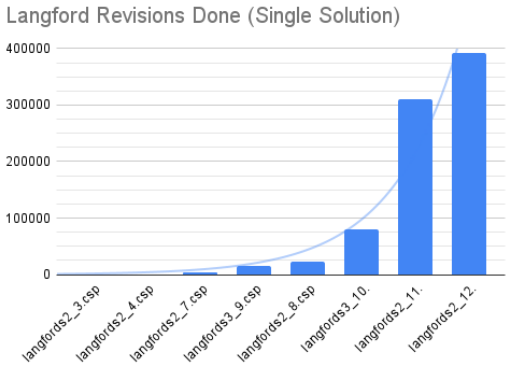
When generating Langford's problem instances, it was found that, for problems where $k = 2$, n had to equal 0 or -1 when performing $MOD4$. When defining the position of any number l to be a_l, then for $k = 2$, its twin will be in position b_l, where $b_l = a_l + l + 1$. When these positions are summed, we get the equation $\sum a_l + \sum b_l = 1+2+3+\cdots+2n$. This can be reformulated as an arithmetic sequence and rearranged to get $\sum a_l = (3n^2 - n)/4$. From this, we can see that $(3n^2 - n)$ must be divisible by four, giving the modulo pattern [4], [5]. Looking at the results shown in Figure 11, it can be seen that the domain range n (right number) seems to have a larger effect on problem difficulty than entanglement k (left number).



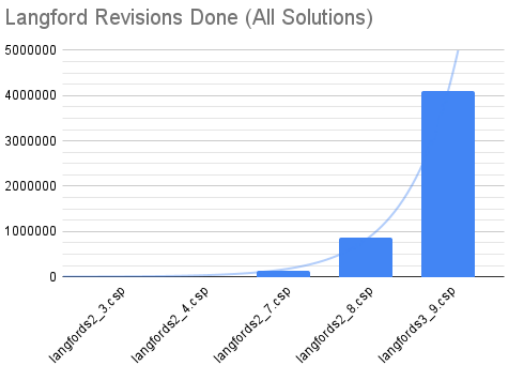
(a) Nodes Explored (Single Solution)



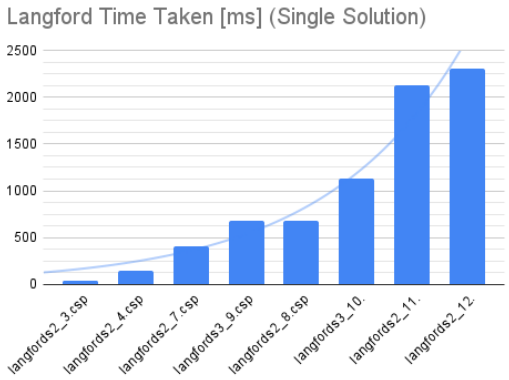
(b) Nodes Explored (All Solutions)



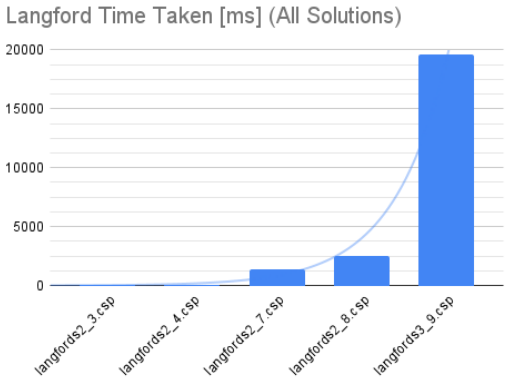
(c) Revisions Done (Single Solution)



(d) Revisions Done (All Solutions)



(e) Time Taken (Single Solution)

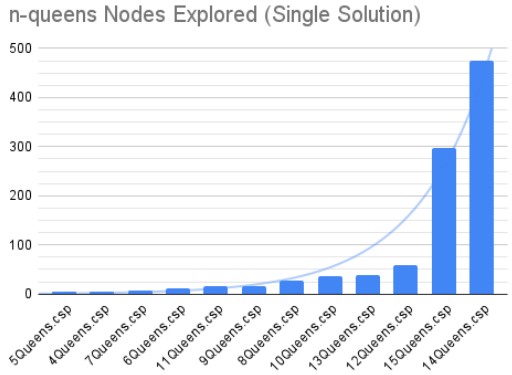


(f) Time Taken (All Solutions)

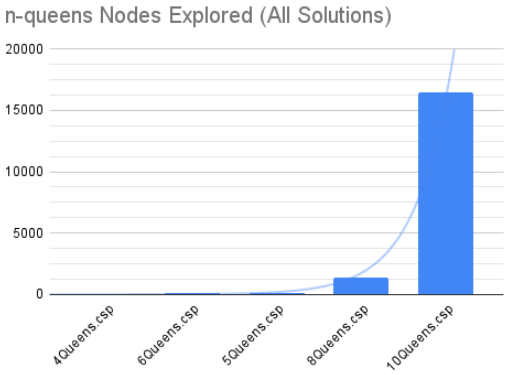
Figure 11: Langford's Problem Data

4.4.2 n-queens Problem

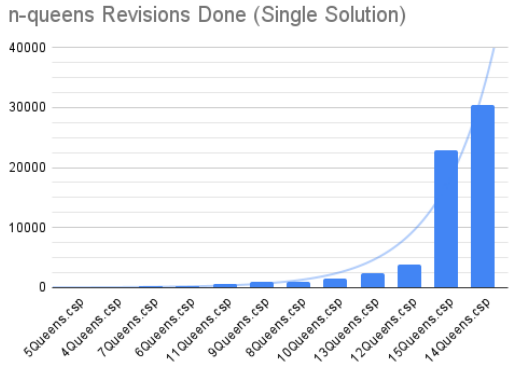
As the number of queens is increased, the challenge to the solver is increased. This is as expected, as a larger board requires more variables to be assigned and a larger search space to be explored. However, the order of difficulty does not seem to match directly with the number of queens for finding a single solution as one might expect.



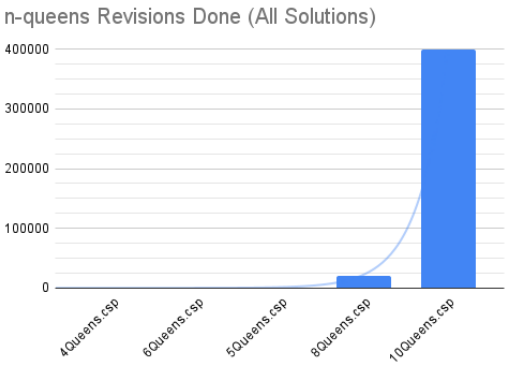
(a) Nodes Explored (Single Solution)



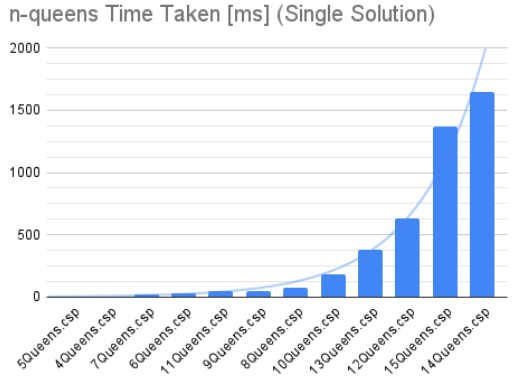
(b) Nodes Explored (All Solutions)



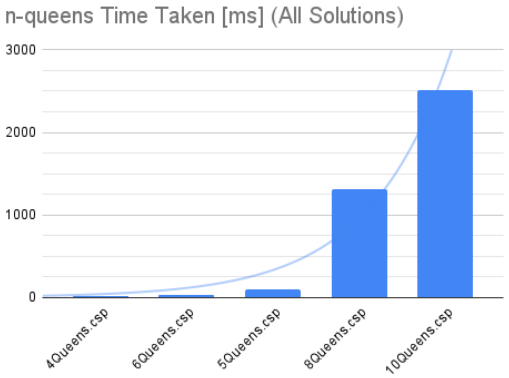
(c) Revisions Done (Single Solution)



(d) Revisions Done (All Solutions)



(e) Time Taken (Single Solution)



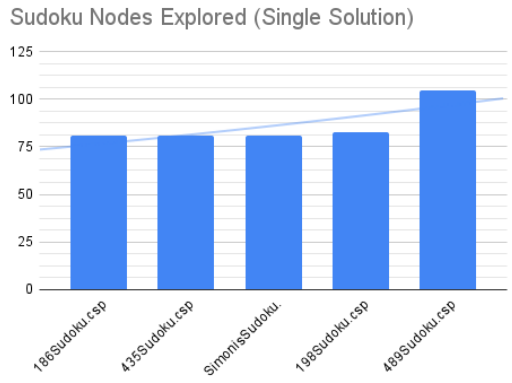
(f) Time Taken (All Solutions)

Figure 12: n-queens Data

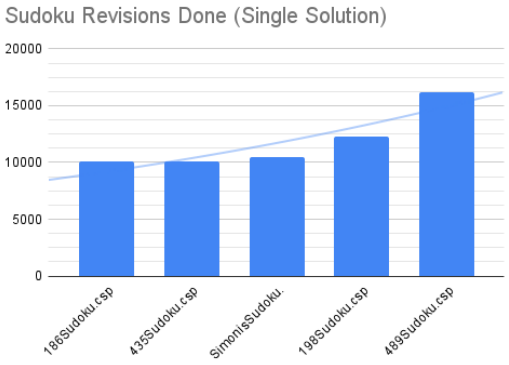
4.4.3 Sudoku

When attempting to solve Sudoku, it was found that easy Sudoku with additional clues were easier to solve than hard Sudoku with fewer clues. This makes sense as the search space on Sudoku with more clues is smaller. Interestingly, it was found that a class of medium difficulty Sudoku existed, where problems were too difficult

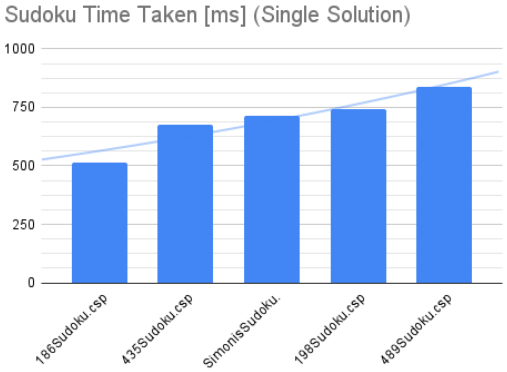
for the solvers to solve in reasonable time when using ascending variable ordering. However, when using smallest domain variable ordering, these problems were trivial for the solvers to beat, highlighting the heuristic's usefulness. These problems were `186Sudoku`, `198Sudoku` and `489Sudoku` as shown in Figure 13.



(a) Nodes Explored (Single Solution)



(b) Revisions Done (Single Solution)



(c) Time Taken (Single Solution)

Figure 13: Sudoku Data

5 Conclusion

The final implementation offers Forward Checking (FC) and Maintaining Arc Consistency (MAC) solvers with two variable and value ordering strategies to choose from. Overall, MAC was found to be the most reliable solver, with ordering strategies having mixed impact on performance. These were evaluated against instances of Langford's problem, the n-queens problem and Sudoku. For Sudoku instances, the smallest domain variable ordering heuristic was found to be vital to maximising performance. With additional time, solver code could have been polished further to maximise performance. Furthermore, additional evaluation could be done, specifically on how ordering strategies affect the performance of the solver when tackling large problems and how constraint pruning affects performance.

References

[1] R. Barták, *Constraint propagation*. [Online]. Available: https://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html (visited on 21/11/2023).

[2] N. J. A. Sloane, *Number of ways of placing n nonattacking queens on an n x n board*. [Online]. Available: https://oeis.org/A000170 (visited on 05/11/2023).

[3] K. Stone, *4 queens puzzle*. [Online]. Available: https://www.brainbashers.com/queens.asp?size=4 (visited on 05/11/2023).

[4] Data Genetics, *Langford's sequences*. [Online]. Available: http://datagenetics.com/blog/october32014/index.html (visited on 23/11/2023).

[5] J. Miller, *Langford's problem*. [Online]. Available: https://dialectrix.com/langford.html (visited on 22/11/2023).