Fordward Checking and MAC3

Hendrik Walter

August 22, 2019

Contents

1 Introduction

This is the project report for the second practical for CS4402. The second practical demanded to develop and test two different algorithms to solve CSP. This is Forward Checking (FC) and Maintaining Arc Consistency (MAC3). The report goes into detail on the setup and the design chosen to solve the problem.

In 2.1 the design of the forward checker is discussed. 2.2 covers the details on MAC3 and 4 covers additional efforts on the solver. The project is summarized in 5.

2 Implementation

Before deep diving into the specifics of the Forward Checker and the MAC, some points should be mentioned about the general setup of the system. The following classes are part of the delivery:

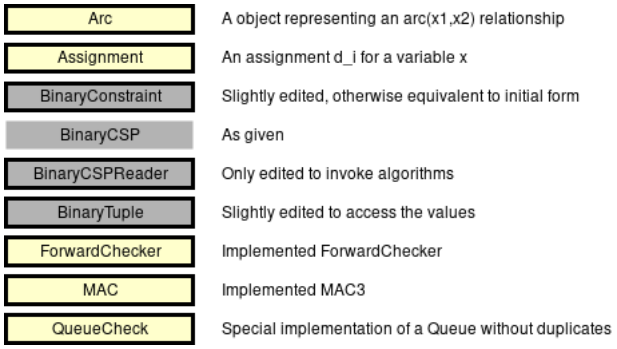

Figure 1: The java classes in the submission (without Generators)· The thick black lines indicate Java files in which there was a change, the files marked yellow are the new files

Please Note: *that the input variables have changed for command line use. Instead of just the file in args[0], the program also expects "MAC" or any other string (falls back to FC in this case) in args[1].*

The workflow and the stack for the two algorithms is as follows:

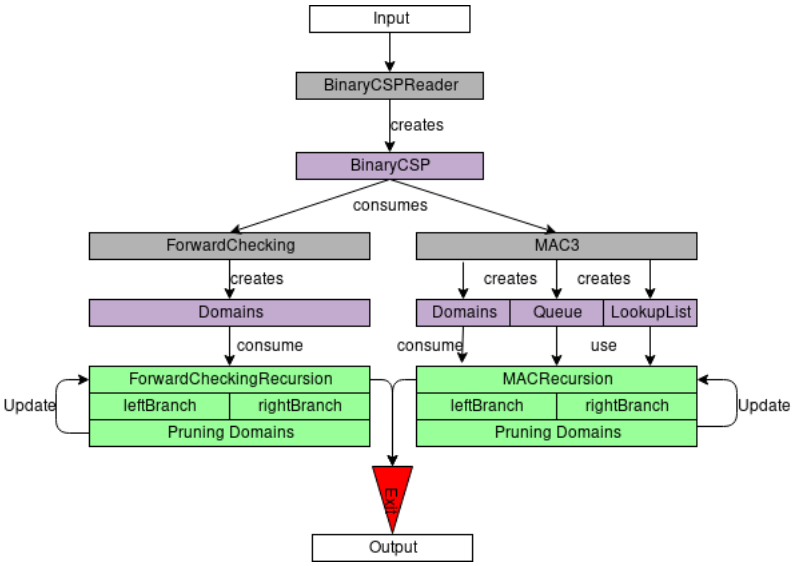

Figure 2: The general Workflow

The graphics does not contain all details and all classes in the overview but gives a coarse overview of what to expect in the algorithm section.

Some arguments and design decisions are equivalent for both algorithms, hence they will be covered in this section. Please also note that some functions are duplicates in the different algorithms. While this can be criticized as violating the "Don't repeat yourself" principle in coding, the student decided to do this as the algorithms should stay independent from each other and now everything is contained in one file.

For both algorithms a recursive design was chosen as the steps are the same during every iteration, equally it allows to backtrack more easily. Prior to calling the worker function, both algorithms have a function managing the recursion and setting up some values prior to initiating the process. To maintain the domains a list of sorted sets was chosen, in practice, they are an ArrayList of TreeSets carrying Integers. Each variable has an Integer name between 0 and n, so the list has exactly this length, such that *.get(n)* obtains the domain of variable n. This list of domains can be found in both implementations.

Further, the functions for assigning and unassigning variables can be found in both algorithms. Both classes have a private field which contains all assigned variables and their respective assignment. This allows to quickly check if a solution is complete and prevents unnecessary passing between functions.

2.1 Forward Checker

The Forward Checker (FC) as implemented in this submission is based on the lecture slides. It compromises a few steps and a few functions not specified in the pseudo code of the lecture slides which were introduced to allow a better overview. One example is that the class carries two fields, a list of Assignments and a Set of Integers. The former is used to ensure that all assignments are saved across all function calls and the latter is used to allow a quick check if a

value has already been assigned. Appearing like double work, it allows fast and easy look-up times to check attributes to the current assignments. In particular the list allows to keep track of the order in which the variables were assigned, which is important in case there is a sophisticated variable selection process in place.

Starting FC: In general, the FC creates the domains for each variable in the first step prior to entering the recursion. This domain model, consisting of a list of SortedSets, will be passed on through the whole computation and is one of the key ingredients for a successful computation. The other central object is the binaryCSP object. Both are passed to the recursion function initiate the computation. The first step is checking if the assignment is already complete, this can be done by comparing the number of variables to assign with the size of the assigned list. If successful, the machine can exit by printing the result. However, in most cases some computation needs to be done.

Selecting a variable: The first step is selecting a variable, which is done by iterating through the domains, checking each domain and noting the size. The smallest size is saved and the respective variable is selected. This fulfills the requirement that the model checker is *smallest domain first*. After this is complete, a value needs to be selected, this is again based on the list of domains, in fact this operation is extremely fast, as the domains are all SortedSets, so the first value in the domain will be the smallest value. This complies with the requirement for the solver to use *ascending* value ordering.

Left Branch: At this stage both the variable and the value are selected and the recursion algorithm proceeds into the left branch of the computation. The left branch formalizes the earlier assignment by persisting it into the assignment list of the solver and adds the variable to set of assigned variables, further, it creates a deep copy for rollback purposes of the domain object and reduces the domain of the selected variable to exactly this domain and starts the revision process. The revision process takes the binaryCSP, the domain and the assigned variable and revises all future Arcs. It does so by iterating through all unassigned variables (e.g. skipping the assigned variables) and passes them into the revise function. This revise function takes the constraints of the model, the domains and the current variable of the set of unassigned variables. This functions finds the appropriate constraint and filters for all values which are still supported by obtaining a new set of this revised domain. This revised set is used to retain only those values from the Master list of domains which has been passed into the model. A change here will impact the model at higher levels too as the reference to the object has been passed along all calls. If the revised domain is empty, it returns from the process and the revision of future arcs failed, if all domains have a size greater than 0, the left branch loops into the next recursion. Upon failure, the domain gets rolled back using the deep copy earlier and the assignment gets undone by amending the fields in the FC object. It then returns to the recursion function it was called from, which invokes the right branch.

Right Branch: At this stage it is known that the selected variable doesn't have the selected value. Hence, the domain shrinks by exactly that value. This

can be done in one simple call, because the domain can be obtained using the domain list (reminder for variable i, get(i) returns the domain) and removing a value from a set is equally simple. If the domain is empty now, the right branch adds the value back to the domain and returns, if it is not, the current domains are cloned and the future arcs are revised. This step is equivalent to the process described above and hence will be skipped in this description for simplicity. If all domains still have a size greater than zero, the next level of recursion is started. Otherwise, the domains are rolled back (including the initially failed value) and the right branch returns.

By only choosing all unassigned variables for the pruning process, this solver only looks into the future assignments and only iterates through them once, not considering that some changes made to domains at a later stage might also impact the domains from earlier arc checks. This is the *forward checking* requirement of the practical.

The requirement of *2 way branching* is fulfilled due to the fact that the right branch also loops into a deeper level of recursion by denying a value. Hence, the left branch tests for a claim $x_i = d_j$, while the right branch tests for $x_i \neq d_j$.

2.2 Maintaining Arc Consistency

The MAC3 algorithm tries to improve some of the known weaknesses of the FC. In particular, it reconsiders pruned domains if they might be affected by pruning processes at later stages. This should allow to spot dead ends quickly when entering a stack of recursion calls. Many of the steps are similar to the ones described above, so this section will only outline the differences where appropriate. Please note again, that many of the functions are duplicates, which is done in full awareness, and should simplify the process to understand the flow of the checker without jumping between different files. MAC also does not have dedicated left and right branch function, but they are marked by comments in the code.

Additional Fields: In addition to the fields of the FC Object, the MAC object also contains an ArcQueue and a LookupList. The ArcQueue is a simple queue of Arc objects. Arc objects carry two variables and identify that there is an arc relationship between the two variables. The special arrangement of the queue is that it doesn't allow duplicate values in the queue. During the revision process this becomes handy, because it allows the application to simply add elements without having to ensure that there are no duplicates, reducing computational complexity. The LookUpList serves as a dictionary. For slot n in the list, a set of Integers is returned which consist of the variables n has some form of constraint with.

Starting the algorithm: The algorithm is started, equally to the FC, with generating the domains. It also generates the ArcQueue and the Lookup list, completing the set up of the object before starting the recursive computation. The recursion itself also checks is the assignments are complete by comparing the size of assigned variables with the number of variables on the binaryCSP object. Afterwards it selects a variable and a value in similar fashion as FC.

Left Branch: After assigning the value to the variable and cloning the domain, it starts to revise the arcs based on the AC3 algorithm. This relies on the domains and the constraints and is a more elaborate view of the domains and arcs than FC. The AC3 algorithm first generates the Queue again, so all arcs are included. Until this queue is empty, it selects the first arc in the queue and finds all values for which there are support by using the revise function. In this revise function, a set is created with all values of the selected domain. By iterating through all values of the domain and all possible allowed values by the binaryCSP object, this is guaranteed to yield the desired output. By removing all values which don't have support, the domains are cleared out. The revise procedure returns one of three values. -1 for an empty domain and an early exit of the AC3 algorithm. 0 for an unchanged domain, implying that no new arcs need to be added to the queue. 1 for a changed domain. In this case the AC3 algorithm iterates through the set returned by the LookUpList and adds all required Arcs. A reminder, that the implemented version of a queue will only add an element if it is not already in the queue. This is based on values and not by object pointers. Finally, the AC3 algorithm reports a successful execution if all domains have still size greater than 0. The left branch will succeed into the next level, otherwise the domains will be restored and the right branch will be computed.

Right Branch: The right branch will first delete the assigned value from the domain of the variable and will then proceed to clone the domain and launch the AC3 algorithm. This is equivalent to the procedure described for the left branch. The right branch will only succeed to the next recursion if there all domains still have size greater than 0. If not, it will restore the domains and finish up the function call.

The described procedure implements all the requirements as laid out in the document. It is *2 way branching*, *ascending order* and *smallest domain first* for the same reason as FC.

3 Evaluation

This section will touch on the evaluation of the two different algorithms. There are different ways to measure the performance, especially when problems increase in size. This can include the number of nodes in the search tree 3.1, the time taken to perform the test 3.2 or also the arc comparisons 3.3. All experiments will be executed using the same problem class to allow an easier comparison between the two algorithms. A problem which is fairly easy to conceptualize in terms of increasing complexity with increasing problem size is the nQueen problem. It will serve as a basis for all experiments in this section.

Prior to the experiment, the expectations are that there will be less nodes in the search tree for MAC compared to FC, because it should be able to stop dead ends earlier. The time taken might be higher because of the repetitive revisions of the arcs. Further, it probably has more arc comparisons than the FC algorithm because it always checks all changed domains upon every domain change. It has more arc comparisons per node.

Note: While the other problems are not included in this report, they were still checked, to ensure that the algorithms worked as expected. All adjustments made for measuring the performance have been commented out.

3.1 Nodes in the search tree

To measure the nodes in the search tree, there will be an additional counter in both algorithms which is increased with every call of a left or right branch. In this context an assignment of a variable to an unsuccessful value accounts for two nodes, leading to *lucky* configurations in which a larger problem size could in fact have a smaller number of nodes. This can also be seen in the experiments for the nQueen Problem. The table below shows the number of nodes in each algorithm for the experiments:

n *Queen*	MAC	FC
4	6	12
6	18	46
8	46	134
10	22	54
12	100	276
14	84	228

This is visualized in the following graphics:

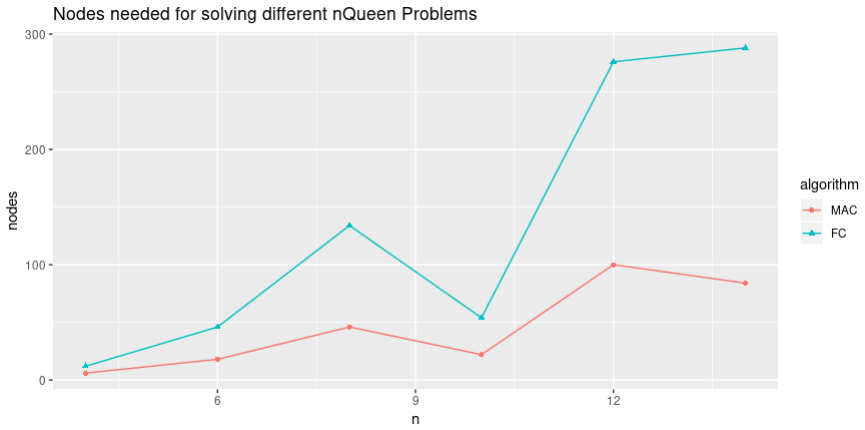

Figure 3: The nodes needed from MAC and FC for different sized NQueens Problems

Two things are striking in the table and in the plot:

1. MAC **always** has less nodes
2. With more difficult problems with number of nodes doesn't necessarily go up.

Observation one is less surprising. The reason behind using the MAC algorithm was to spot dead ends earlier, this allows the algorithm to avoid to create

too many nodes leading to a dead end. An example for this is given in the graphics below:

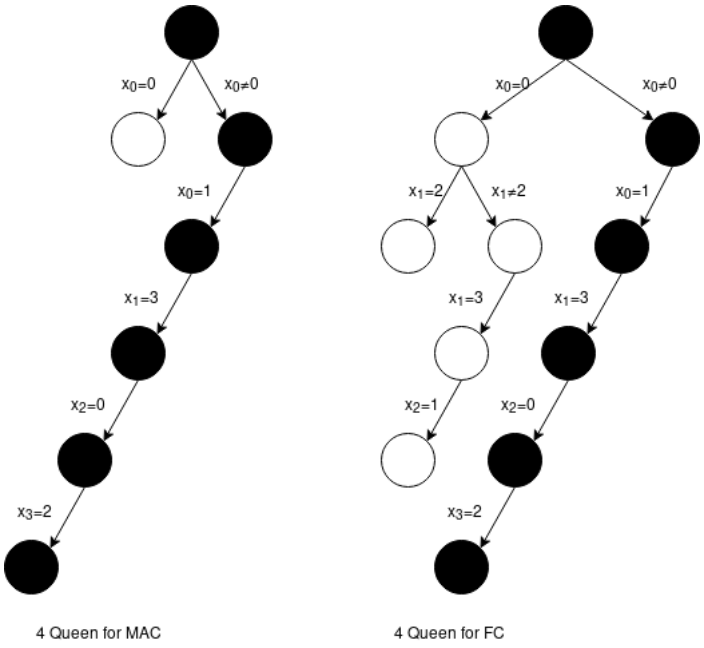

Figure 4: The nodetrees for 4Queen problem

On the right hand side we can see that there is a large arm leading down with many nodes not actually needed in the final assignment (marked in black). MAC catches on this very early and only has one miss.

The second observation might be more surprising. This observation can be explained by looking at the values, especially the selection of the variables and the their values can make a big difference on the number of nodes. This will be investigated at a later stage in the extension part of this submission.

3.2 Time Taken

While measuring the number of nodes is a good indicator of how well the algorithms perform on a theoretical level, the node saving might come at an expense caused by iterations through additional objects and hence a higher computational complexity.

The time is measured for both algorithms after the initiator function was called and then the time stopped once it was decided that the algorithm found a solution. The time is presented in ms and are averaged out over multiple runs.

n Queen	MAC (μ)	MAC (σ)	$FC(\mu)$	FC (σ)
4	5.1	1.48	3.1	0.8
6	20.72	5.77	9.77	2.5
8	84.72	50.3	19.66	2.98
10	25.74	8.69	15.02	2.799
12	62.89	11.93	30.94	3.59
14	80.21	23	32.41	4.65

Table 1: Time in ms

From the table one can deduce that the times are fairly equal. That the FC executes significantly fast than the MAC algorithm. While the differences are fairly small for smaller problems, the times start to diverge for larger problems. This becomes also visible in the plot below.

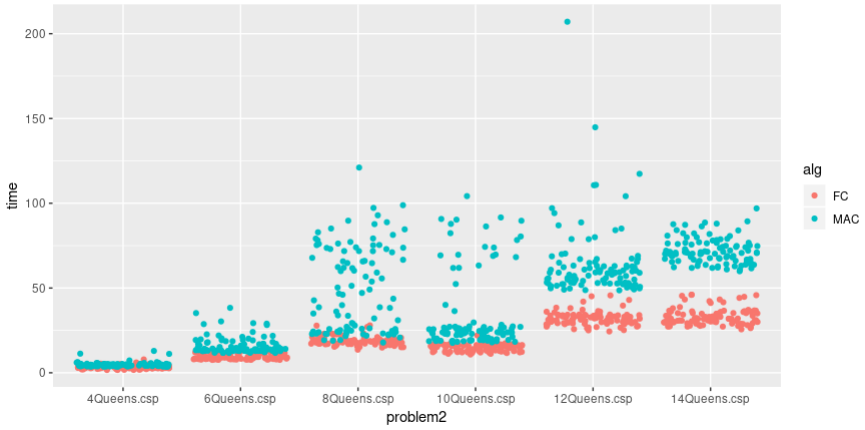

Figure 5: The different time points for different nQueen Problems

In fact, the plot shows also fairly well how MAC3 always performs slower than FC. The observation that MAC always has less nodes might come at the expense of an extended iteration period over the arcs. This observation might also be due to the implementation which is heavily dependent on iterating. Hence, the slower performance of MAC might be caused by the implementation and not the algorithm itself.

3.3 Arc Revisions

The arc revisions are measured by counting each call to the revise function. The results are measure for the nQueen problem again. Please consider the table below:

n Queen	*MAC*	*FC*
4	72	18
6	478	96
8	2018	317
10	1761	164
12	10331	864
14	11585	820

Table 2: Number of calls to the revise function for different nQueen Problems

As suspected the number of revise calls are significantly higher for the MAC algorithm than for the FC algorithm. This is due to the fact that the AC3 procedure considers all arcs effected by a change in domain, even thought the change has not impacted the support for the enqueued arcs. This is the price to be paid for a lower amount of nodes in the search tree.

4 Extension

The proposed extension of mine covers a static variable ordering based on user input and a performance comparison (based on search tree size) with the smallest domain selection criteria. Furthermore, this section makes a proposal for improved performance on average.

4.1 Variable Selection

In the above implementations of the algorithms a dynamic heuristic for selecting a variable is used - smallest domain first. It is known that this does not always yield the best choice of variable. The best choice is here defined as "lowest number of nodes in the search tree". To understand how the nodes are distributed for a different variable selection, one can run an experiment as follows: 1. Implement static variable selection, 2. generate all possible orders of variable selection, 3. compare with the nodes for lowest domain first selection.

This experiment has been implemented for this report.

1. The static variable selection is implemented using additional fields and only for MAC3 to show the proof of concept. The fields are an ArrayList with the static order, a boolean indicating that static ordering is used and a index indicating the current selection in the order. The use of the static ordering is simple. The user passes the order into the program by simply writing the order of the variables, e.g. "java BinaryCSPReader 4Queens.csp MAC 0 1 2 3". This will let the program convert the "0 1 2 3" into the correct format and pass it on to the program. Whenever selectVar is called now, the program will check the boolean flag and instead return the respective variable from the arraylist. The counter is increased/decreased whenever a variable gets assigned/unassigned. This accounts for assignments being removed and re-selection of variables if a right branch is taken.

2. To generate all possible instances, a R script has been used. It generates the instances by selecting the numbers from 0 to n-1 for a nQueen problem

in random order and then passes them on to the program. The program then counts the number of nodes in the search tree and reports back to the script. The script collects the data and computes the averages, means and also allows to plot densities for the last step.

3. For the comparison, please see below.

Considering the amount of combinations for the order of the variable (n!), it is not sensible to conduct this experiment for larger problems. All cases are likely to be covered for the 4 Queen and 6 Queen problem ($4! = 24$ and $6! = 720$), for the case of $n = 8$, it is already less likely as $8! = 40320$. However, for demonstration purposes it a subset of those combinations has been included.

In the plot below all three violin plots are displayed:

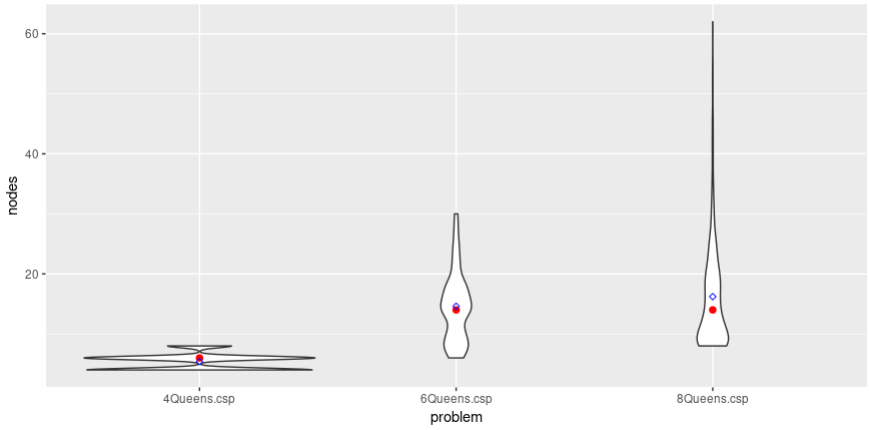

Figure 6: The violin plots for the nodes for all different variations of input orders. Median in red, mean in blue

First of all, there are different shapes in the plot. The 4Queens problem has very defined extensions, e.g. areas with a lot of measurements, while nearby nodes basically didn't occur. In contrast, the other two problems have a strong peak, in particular the 8Queens problem has a very distinct peak. The plot allows to make the following statements:

1. For the 4Queen a lot of the orders lead to similar sized node trees. A phenomena which can be explained considering the limited number of possible values available, there must be a minimum of 4 nodes in the tree, however the upper number if capped at 30 (binary branching, only four values available for each variable). Hence, the possible domain of tree sizes is limited.

2. The strong peaks in 6 and 8 Queens problem lead to the assumption that there are *terrible* choices for the order, leading to enormous tree sizes, because very few variables can be pruned effectively.

3. Depending on the order of the variables, the search three for a 6Queen and 8Queen problem are very likely to be the same, in fact it seems that the median and mean are almost identical. Checking with the data, the

median for both experiments was 14, the mean for the 6Queens problem 14.6 and for the 8Queens problem 16.21. This seems counter intuitive as the problem size grew, while the median stagnated. A possible explanation would be to argue that the 8Queen problem is solvable in a manner such that a selected assignment prunes the domains for future assignments so neatly such that very few assignments remain for testing.

Having seen that there are overlaps for node tree sizes and that there can be terrible choices for order selection, the question remains how well the smallest domain selection performs in relation to a perfect order?

A graphics containing an answer to this question is given below:

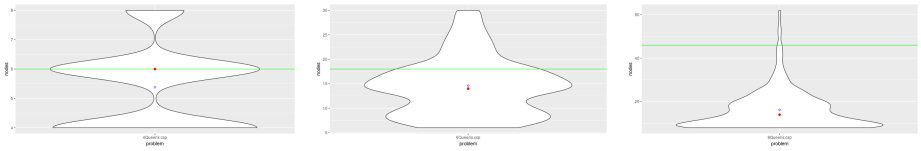

Figure 7: Median in red, mean in blue, smallest domain selection in green. Left to Right 4/6/8-Queen problem

It is quite striking that the smallest domain selection (SDS) avoids the worst cases for all problems, which is fairly comforting. However, it is also not hard to see that SDS is far from performing well. In all cases it is worse than the mean number of nodes and in all cases is equivalent to the median number of nodes or worse. In particular for the 8Queen problem SDS performs terrible. Achieving the lowest of the possible nodes is not realistic, as this would require some form of oracle, but considering and interpreting the percentiles shows why more sophisticated measures might be required.

n Queen	*MAC with SDS*	*Percentile*
4	6	87.7
6	18	81.4
8	46	98.7

Table 3: Percentiles (Values in the sample equal or smaller than the one reported)

All percentiles are very high. Very striking is the value for the 8Queens problem - 98.7% of all possible search trees are smaller than the one observed using SDS. This means that randomly assigning an order is only worse than SDS in 1.3% of all cases. This results together with the oberlap for the search tree sizes between the problems now also explains why the search tree for 8Queen was bigger than the one for 10Queen (as seen in Evaluation). The situation regarding the percentile is similar for the other problems, even though not as extreme. One needs to take care with the percentile result for the 4Queens problem because of the limited node possibilities. The percentile for 5 is already 47.7%, hence there are a lot of search trees with six nodes (hence the wide body).

From this section one can see that the SDS does not perform very well compared to random assignments. If one is interested in the mean performance

over a multiple problems, one is better off selecting a random assignment rather than smallest domain variable selection.

5 Conclusion

This report presented FC and MAC3, two algorithms to solve binary CSP problems. The implemented classes fulfill the requirements as set out the project description and solve the problems adequately. The classes and design decisions have been explained and justified as required. The empirical evaluation is included, showing that MAC always has less nodes, does more ARC revisions and takes longer to compute. Finally, an extension covering static variable ordering as well as an experimental approach to assessing the performance of the selection a variable based on the smallest domain has been included in this report.