University of St Andrews
School of Computer Science
Senior Honours

CS4402
Constraint Programming

Practical 2: Constraint Solver Implementation

April 18, 2019

150003431

Contents

1

Introduction

1.1 Overview

This practical involved implementing a constraint solver supporting two algorithms, Forward Checking (FC) and Maintaining Arc Consistency (MAC). Given the number of problems translated as binary constraints, the solver is designed to be capable of executing the two efficiently with good scalability. The efficient OO design is sought with a test environment for empirical evaluation. The two methods are compared in terms of performance metrics. Beyond the basic requirement the project is extended to support various types of heuristics for variable and value assignment ordering with their implication discussed in their relative performance improvement. Additional problems are implemented for further analysis.

1.2 Instructions

Data

Any data files provided from the specification reside in src/test/resources. Any example outputs of the test environment are ready saved and will be generated in src/test/output. Any extra data files generated are also saved in extra/ and unsolv/ directories (to be discussed in detail in the evaluation section.)

Commands

The test suite is designed as a Maven project where source code and JUnit tests are located in relevant directories. Commands can be run on Terminal at the directory *runner* where "pom.xml" is located. Following commands will be used mainly to execute the tests.

```
$ mvn clean
```

Removes the current compiled class and output files for a new build.

```
$ mvn clean test -Dtest=SolverTest [-Dprint=<Y | N>]
```

Runs the solver for all 10 problems sources in the specification using both Forward Checking and Maintaining Arc Consistency under the default variable and value ordering as specified for the basic solver. This takes around 20 seconds to execute including the build time (16 solver runs).

For testing other problem instances other than those available in src/test/resources, create a new directory or use others as mentioned above, containing those files under src/test and pass the directory name as following: ("extra" directory is an example of this.)

```
$ mvn clean test -Dtest=SolverTest -Ddir=<dir_name>
```

-Dprint=<Y | N>

- **Y** : generates csv statistics output (with no solution), with every problem output as a row entry (includes headers: problem name, type of algorithm used, time taken in ms and s, depth (number of nodes), extra count, type of variable heuristics, type of value heuristics)
- **N** : generates txt file output of statistics and solution

-Ddir=<dir_name>

directory located under src/test where problem instances to test reside. (default is src/test/resources)

As an extension, other types of heuristics are implemented and these can be using the following test class. (would recommend also reading empirical evaluation and extension section prior to the usage).

```
$ mvn clean test -Dtest=AdvSolverTest [-Dorder=<all | allVal | basic |
    def (dynamic)>]
```

As with previous test class data directory can be customised to solve new instances, so does print type which can be adjusted using -Dprint argument. -Dorder is the important argument to use. The test class can run with 3 settings.

-Dorder=<all | allVal | basic | dynamic (def) >

- **all** : will run under all 5 variable order heuristics, both static and dynamic, and 2 value order heuristics. (includes 2 static variable ordering)
- **def | dynamic** or omitted : will run under all 3 dynamic variable order heuristics with 1 value order heuristics. (default)
- **allVal** : will run under 3 dynamic variable order heuristics with 2 value ordr heuristics.
- **basic** : same as *SolverTest*.

Both *SolverTest* and *AdvSolverTest* support additional optional argument -Dm=<type> where you can specify to run only with one algorithm type, as the default set up is to use each one in solving the given problems.

-Dm=< b | fc | mac>

- **b** : to run on both (can be omitted for being a default option)
- **fc** : to run only with Forward Checking
- **mac** : to run only with Maintaining Arc Consistency

2

Basic Solver

2.1 Design and Implementation

For the Basic Solver, the key design decisions made are the following:

- The implementation of two algorithms superficially follows the pseudo code provided from lectures as inspiration: both algorithms are designed to be recursive methods analogous to tree branching operations. The following will describe how they are implemented.
- The Basic Solver is designed in the class *Solver* which is capable of solving the provided Binary csp problem using both algorithms. The class contains all methods required where some functions are being shared as appropriate in the running of the algorithms. Type of an algorithm to use for solving is determined by a boolean parameter passed to *solve* (true for FC, false for MAC). *Solver* accepts a BinaryCSP instance as a parameter with key information saved in the initialisation steps.
  - Given the domain bounds, domain values for variables are populated in a 2D ragged array which will be used in the algorithm duration.
  - Variables are represented in a HashMap as key, with indices to access the domain array as values. This is to ensure that the solver does not assume that the input csp will always takes variable values from 0.
  - Constraints are saved from the BinaryCSP instance and this is used to extract the information which is saved in the attribute *connections*: it records the list of variables which one is connected to for every variable.
- The key idea to be implemented for both is **to support domain pruning via arc revision.** This is designed to be supported as follows.
  - Prior to the method call of *revise*, which conducts domain pruning, methods *reviseFA* for FC and *AC3* for MAC initialise an empty map for <binary tuple (binary constraint), list of binary tuples> and push to a stack. This is a step where it is allocating the space to be used by *revise*.

  - During the execution of *revise*, whenever it encounters incompatible domain values to be pruned, for each value, it collects the corresponding tuples which are no longer supported from the related BinaryConstraint from constraints list, hence prunes the list. Pops the previously pushed map and saves the pair and push back to the stack. This is followed by removing the value from *domains* array by setting the value to be [-1]. Hence the implication of such operation is that domain pruned is saved along with its tuples had it supported if valid. By going through all the relevant domain values to check, the map with the collected pairs is the outcome of domain pruning of one specific *revise* call. The method revise hence will take the related *BinaryConstraint* instance containing all valid binary tuples for the two variables as a parameter at the time of calling.
  - Maintaining the pruned outcomes in a stack is suitable for undo-pruning process at backtracking. By removing the map at the top, the latest values pruned, these will be reflected back to both constraints list and *domains* array. Note that the value removed and the variable whose domain the value is pruned from, are deduced from the map. There is a boolean indicator first which is saved for every key binary tuple which denotes the direction to which constraint should be read as an arc, which helped in AC3 implementation of arc revision. The value is restored in the domain by entering in the *domains* array. Having the uniform method of domain pruning and undo-pruning implied that they can support both algorithms.

- Given the above implementation of domain pruning, the subsequent method calls in the algorithms will always use the newly revised domain values for the smallest-domain-first heuristics. The basic implementation of *sortVarList* handles arranging the variables in the ascending number of available domain values for the basic solver.
- The solver has method *printSol* which is used to print out solution including the statistics. In order to ensure that the solver can be restored back to its initial state with fully initialised problem attributes, the method *reset* will read from the BinaryCSP instance again to restore from the solution state, for it to be reused. In addition, the same solver instance can be used to solve a different BinaryCSP problem. by passing a new instance of BinaryCSP to *setNew*.
- The solver is robust in dealing with problems with no solutions, by checking the final assignment status given the end of the solving method call. The outcome of the search, hence, will be indicated in the output statistics files including the usual statistics of time and depth of the search.

The following describes algorithm-specific method implementations.

2.1.1 Forward Checking

As with the pseudo code guideline, the method call to *FC* is divided to 2-way branching by branchFCLeft and *branchFCRight*, where each has nested call to *FC*: Left branch calls *FC* having decided on the variable and value assignment and Right branch calls *FC* having checked that given the value is removed, the resulting domain bounds are valid to proceed. The intermediate call of reviseFA is used for arc revision with all future variables and this is the method which initialises the map for storage to be used by revise.

The purpose behind the choice of general data structure used are:

- **Arrays** for domains, domain bounds, tuples - for compactness.
- **ArrayList** for *varList (list of unassigned variables), stack* - for flexibility in addition and removal.
- **HashMap** for *pairs (as mentioned above), variable-index mapping* - for constant time extraction of matching values.

2.1.2 Maintaining Arc Consistency

As with the method call to FC, The method call *doMAC* initialises the variable list. It also calls the initial AC method in order to enforce the first general arc consistency where any specific domains with bounded values pre-assigned can be reflected onto the list of valid tuples in the *constraints* list. Arc revision carried out by AC takes two arguments to initialise the queue differently depending on the time it is being called. At the initial call as described, the queue will contain all the arcs. Throughout the execution of *MAC*, the queue will be initialised with only all the incoming arcs to the variable interested (currently assigned or has its domain value being removed).

The queue is designed to be a Doubly linked list priority queue. The implementation is taken from the work on Stacks and Queues from CS2001, so did the implementation of the stack. At the start of *AC*, the map will be initialised to be used for storing all binary tuples pruned from binary constraints, hence, the domain values, at *revise*. Depending on the outcome of the method, arcs will be reintroduced in the queue. The previously stored binary relationship of every variable (*connections*) is used here to get the list of variables which are connected with the variable of interest using the method getConnectionVar. Hence, using the returned list, only arcs not in the queue will be added. getFail method will be called when fail private attribute of Solver is set by revise to indicate the incidence of emptied domain of a variable. This will be received by *AC* which propagates back to MAC to proceed to undo-pruning process. At the end of *MAC*, previously removed value from the variable is reassigned to the domain given that the value is known.

2.2 Heuristics

As previously mentioned, smallest-domain-first is implemented by *sortVarList* method which is called at the start of every method call of *FC* and *MAC*, as one of the options in selectVar method. Value selection is done in ascending order by sorting the domain array and selecting the first non negative value. This is further extended to support other heuristics which will be discussed in the later section.

2.3 Empirical Evaluation

In order to assist the empirical evaluation process, *Counter* class is designed to take care of statistics measure, recording and output. The relative performance of two algorithms in terms of their space and time-wise complexity are compared. For performance evaluation, the following metrics are measured and recorded:

- The time elapsed: start time is recorded at the call of the first method call with the end time being the time which it lands at *completeAssignment* when a solution is found. When the solution is not being found, the method call will return after the exhaustive search thereby time will recorded at the end of the search.
- The number of nodes: for Forward Checking there are two values which are measured, one for the total number of method calls to FC, branchFCLeft, branchFCRight and the total number of calls to FC only. This is to analyse the performance of the solver in terms of both width and depth. It is speculated, however, this will be double the amount of the first measure given it searches in balance. For Maintaining Arc Consistency, the total number of calls for *MAC* method is counted. Hence the the last two values are comparable to understand the depth of the search.
- The number of calls to arc revision: for MAC, the number of times *revise* is called inside *AC3* method is recorded.

General Testing Output

As described in the instructions, empirical evaluation is delivered by JUnit test environment with results generated in a comprehensible format. The first analysis on the performance of the solver using the two algorithms provides following observations:

- FC and MAC show comparable performance where they work under similar time scale where FC wins over MAC to some extent. The Queens and langfords problems up to 3_9 show how MAC finds the solution with fewer number of recursive calls in comparison to FC, with extreme cases by half or a third. There would be an offset made by its time performance, where it proceeds ARC revisions at each call which may involve some unnecessary iterations.

Table 2.1: *Summary Output of Problems Solved*

	Problem	FC.time (ms)	MAC.time (ms)	FC.depth	MAC.depth	MAC.revisions
1	4Queens	0.760	0.510	9	6	48
2	6Queens	2.450	1.400	27	11	243
3	8Queens	9.550	5.100	72	26	937
4	10Queens	16.020	14.250	33	15	790
5	langfords2_3	0.500	0.300	13	6	119
6	langfords2_4	1.200	0.550	33	8	297
7	langfords3_9	530.950	854.630	1,894	1,206	242,069
8	langfords3_10	2,915.340	4,370.970	6,677	8,065	997,714
9	SimonisSudoku	6.780	256.630	82	81	12,412
10	FinnishSudoku	923.540	1,671.210	7,244	9,917	696,824

- The performance can also be analysed in terms of its implementation. For both algorithms, the method *revise* and undoPruning involves double loop iterations across domain of the assigned variable and the other variable (future variable for FC). The method *assign* and *unassign* also takes linear time for removing and adding elements for it being designed to be an array. *revise* also has *checkMatch* which is executed to check if the given pair of variables are connected by a constraint in the given problem. This loads further iteration. This is the major area where FC method will consume its time, whereas MAC involves Arc revisions via a queue additionally. Hence, it is considered that MAC will have a higher time complexity than FC.

In order to understand the scalability of the two algorithms and the implementation, the solver is tested on Queens problems with a increased size of N. This is executable using QueensTest:

```
$ mvn clean test -Dtest=QueensTest [-Dlower=<> -Dupper=<>]
```

where optional arguments *lower* and *upper* denotes the lower and upper bound of N (inclusive) to run. The default is set as [5, 50] as shown in the plot below.

- The outcome shows there is a strong correlation between the number of nodes and the time elapsed. This suggests that the time mostly reflects the computation required at each node. Hence, this implies that although better implementation of the algorithm to minimise time complexity may help reduce the time consumed, it may have little benefit for problems which necessarily requires the computation steps for the set series of variable and value assignment.
- As the size of the problem increases, both algorithms show increased time and space complexity. MAC consistently wins over FC in terms of depth while the winning is not reflective on the time spent.

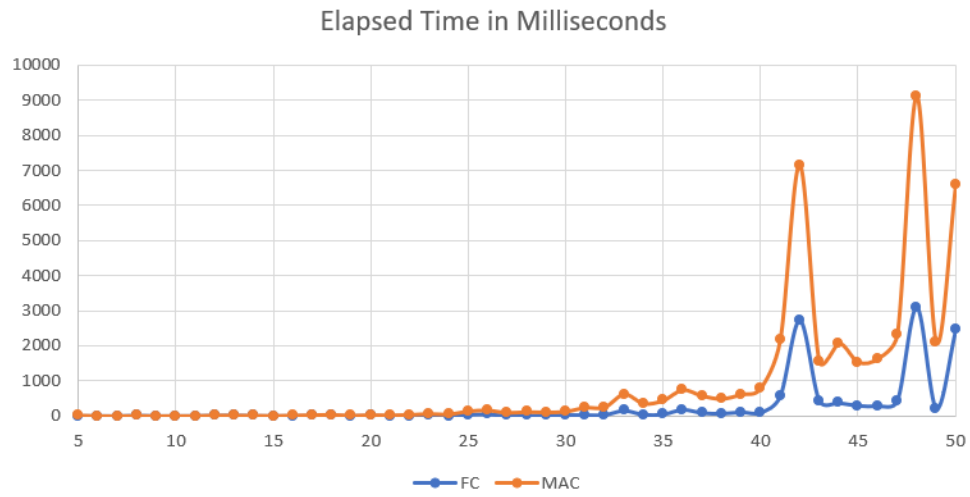

Figure 2.1: *Plot of Execution Time for N-Queens Problem*

- The outcome also suggests that for this specific problem, an increase in the problem size may not imply a *monotonic* increase but rather problem-specific: for problem of size N = 42 and 48, both solvers required drastically increased number of search in order to find the solution. This suggests that the outcome may also highly depend on the order in which it searches the parameter space, which can be investigated further by using different types of variable heuristics.

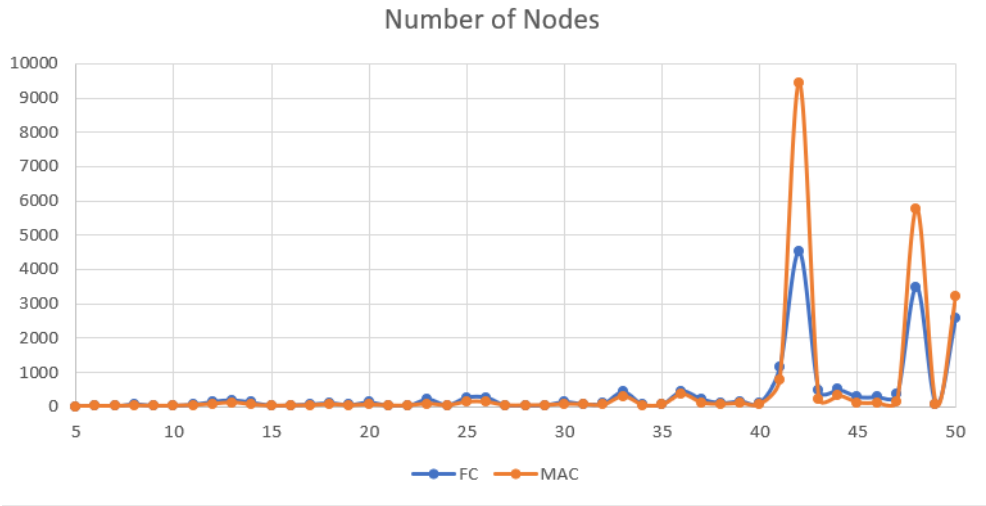

Figure 2.2: *Plot of Number of Nodes (depth) for N-Queens Problem*

Additional experiment conducted to compare the two methods is for exploring their ability in identifying unsolvable problems. Some notable instances from Langfords Number problems are generated which are known to have no valid solutions. As shown in Table 2.2, we can see that MAC is more powerful in identifying problems with no solutions. Not only does it wins by the execution time, it also identifies the result early on by AC3

arc revisions which prunes the domain at every execution of MAC. The depth drops by a factor of 10 with an extreme case for Langford's L(4,5) problem, where MAC finds this at its first method call. This exemplifies where FC falls behind and how it requires additional steps in order to find the dead-end on every angle via a series of backtracking.

The difference in their strength is more apparent as the problem becomes large. For Langfords L(4,9) problem FC couldn't resolve within the time frame, hence, recorded to last at least 5 minutes.

Table 2.2: *Summary Output of Solver Given Unsolvable Langfords Problems*

	Problem	Type	Time..ms.	Depth	FC.total	MAC.revisions	Var	Val
1	Langfords2_5	FC	173.42	375	1125		sdf	asc
2	Langfords2_5	MAC	87.81	44		3573	sdf	asc
3	Langfords3_6	FC	479.73	1228	3684		sdf	asc
4	Langfords3_6	MAC	195.85	54		15100	sdf	asc
5	Langfords4_5	FC	352.91	461	1383		sdf	asc
6	Langfords4_5	MAC	125.7	1		4599	sdf	asc
7	Langfords4_9	FC	300000+	X	X		sdf	asc
8	Langfords4_9	MAC	6024.38	922		556856	sdf	asc

3

Extensions

3.1 Adv. Solver - Heuristics

The implementation is extended to support various heuristics. Given enum variable *HEURISTICS* each options are initialised accordingly to be recognised by the solver. The choice of variable and value heuristics is required at Solver instantiation. Any relevant variable ordering heuristics are handled by a separate class *Control* whose instance is introduced as a private attribute of *Solver* to be called. By this new addition, Smallest Domain First (SDF) is included in variable selection method. Hence, variable selection call is divided into two methods, one for initial call at solver initialisation and another for dynamic variable ordering - written as *getVarList* and *selectVar* respectively.

Both static and dynamic variable ordering assignment are implemented where for the former the *getVarList* is suffice to introduce the fixed ordering to be used throughout searching. For the dynamic variable ordering heuristic: *BRELAZ* and *DOMDEG* are implemented in *Control* which also have some helper functions written for these two heuristic calculations. *BRELAZ* is called by passing the varList and *var* which is the variable to be selected under SDF ordering. Hence, the method will compute and compare its first variable with this *var* Additionally, for value assignment ordering, min-conflict are included, which is written directly in *Solver*.

3.1.1 Empirical Evaluation

Using the test class *AdvSolverTest*, all outputs are evaluated and compared with the default setting. The following command will use dynamic ordering heuristics with min-conflicts value assignment ordering on 10 problems instances and will generate a csv output of statistics:

```
$ mvn clean test -Dtest=AdvSolverTest -DprintType=T
```

The example output is summaried in the Table 3.1, where test run on 10 Queens problem is ranked according to the time taken. The additional columns included in the

table shows the type of variable and value assignment ordering heuristics used. The column "Count2" represent different value depending on the type of algorithm used: for FC, it represents the total number of calls to FC and its sub branching methods and for MAC, it denotes the number of arc revisions by the method call *revise*.

Table 3.1: *Summary Output for 10 Queens ranked by Time Elapsed*

	Type	Time1 (ms)	Depth	Count2	Var	Val
1	FC	0.350	11	21	domdeg	minconf
2	FC	0.360	11	21	brelaz	minconf
3	MAC	0.670	10	491	brelaz	minconf
4	MAC	0.670	10	491	domdeg	minconf
5	MAC	0.910	10	491	sdf	minconf
6	MAC	0.950	15	790	brelaz	asc
7	MAC	1.440	15	790	domdeg	asc
8	FC	1.790	11	21	sdf	minconf
9	FC	1.830	33	87	brelaz	asc
10	FC	1.900	33	87	domdeg	asc
11	MAC	15.230	15	790	sdf	asc
12	FC	17.420	33	87	sdf	asc

- Table 3.1 exemplifies the importance of a choice of ordering heuristics for the optimal performance. As ranked by the least time taken for the solver, more advanced variable heuristics including *domdeg* and *brelaz* have enhanced the performance largely. Given that the default setting of using SDF and ascending value ordering for both algorithms exhibited the worst performance, the former choices would be preferred.
- In comparison to another problem shown in Table 3.2, the outcome suggest that more sophisticated variable assignment ordering would improve the performance in general while the value assignment ordering technique is more likely to be problem-dependent. This is the case for achieving the optimal time. However, for finding the solution with the smallest number of nodes, the use of *min-conflicts* indeed outperforms ascending value ordering.

The result from Table 3.2 also suggests that there may not be a large time consumed due to the arc revision operations for MAC. As highlighted, note that unlike the initial observation of depth and time correlation, using *dom/deg* with *min-conflicts* generated solution by only searching half the number of nodes as used by *sdf* and *asc (ascending ordering)*: but the change in time is small, with halved number of arc revisions. The outcome consistently shows that combinations of more advanced heuristics will outperform by large even disregarding the discrepancies that may exist due to being implementation-dependent. For FC on the other hand, the amount of searching required was the same.

Table 3.2: *Summary Output for Langfords3_9 ranked by Time Elapsed*

	Type	Time1 (ms)	Depth	Count2	Var	Val
1	FC	512	1,894	5,653	brelaz	asc
2	FC	520	1,894	5,653	sdf	asc
3	FC	572	2,139	6,388	domdeg	asc
4	FC	669	988	2,935	brelaz	minconf
5	FC	835	988	2,935	sdf	minconf
6	MAC	845	632	150,906	domdeg	minconf
7	MAC	853	1,206	242,069	sdf	asc
8	MAC	1,113	1,203	231,141	domdeg	asc
9	MAC	1,170	811	196,775	sdf	minconf
10	MAC	1,232	811	196,775	brelaz	minconf
11	MAC	1,372	1,206	242,069	brelaz	asc
12	FC	1,611	2,073	6,190	domdeg	minconf

Another testing is delivered to reflect the initial observation made in the previous section regarding the underperformance of the solver for the instance 48Queens problem. See Table 3.3 for its output. It is again tested with various combinations of heuristics to consider its implication. The results fully explains the previous observation: the problem instance gives worst case performance for each algorithm under the default variable and value assignment ordering. The optimal combination has saved more than half of the time from the depth of the tree - it is pruned by factor of 5 and 10 above for FC and MAC respectively. Hence, this implies how the problem is significantly influenced by the choice made. Additionally, the constrast in the results generated from using *brelaz* and *sdf* is

Table 3.3: *Summary Output for 48Queens ranked by Time Elapsed*

	Type	Time1 (ms)	Depth	Count2	Var	Val
1	FC	1,951	3,490	10,420	brelaz	asc
2	FC	2,445	694	2,032	brelaz	minconf
3	FC	2,536	694	2,032	sdf	minconf
4	FC	2,755	3,490	10,420	domdeg	asc
5	FC	2,954	694	2,032	domdeg	minconf
6	FC	3,109	3,490	10,420	sdf	asc
7	MAC	3,763	378	119,872	brelaz	minconf
8	MAC	3,906	378	119,872	sdf	minconf
9	MAC	4,237	378	119,872	domdeg	minconf
10	MAC	7,164	5,777	677,196	brelaz	asc
11	MAC	7,401	5,777	677,196	domdeg	asc
12	MAC	9,120	5,777	677,196	sdf	asc

significant to emphasise. It is significant given that the former only differs by breaking the ties in the variables with the same number of domain values left by choosing the variable

that influences the sub-graph of future variables the most.

3.2 More Problems Classes

As another extension to the project, other problem class such as QuasiGroup Completion for length N 4, 5 and 18 are attempted (stored in src/test/extra) directory, and runnable using both *SolverTest* and *AdvSolverTest* where the latter will allow the use of extended heuristics implementation. Run the following:

```
$ mvn clean test -Dtest=AdvSolverTest -Ddir=extra -Dorder=dynamic -Dprint
    =T [-Dm=<fc|mac>]
```

QuasiGroupGenerator is written to read in 2 types of instance files from 2 different sources:

- *read()*: first method reads in problem instance as provided in CSPLib. All the problems are of size N=30, and the solver has managed to solve a minority suggesting that the large problem size imposes time constraint and also the outcome was problem-specific.
- *read2()* : another read method reads in problem instances provided in lsencode where the above 3 problem instances are generated.

The instances can be solved in the following options:

- *SolverTest* - default set-up : 26 seconds (for 6 runs)
- *AdvSolverTest* - "dynamic" (default) : around 100 seconds (for 18 runs)
- *AdvSovlerTest* - "allVal" : 142 seconds (2 min 22 seconds) (for 36 runs)
- *AdvSolverTest* = "all" : not recommended due to very bad performance of static variable ordering heuristics.

which will generate the csv output as shown in the Tabular format in Table 3.4. Similar to the previous results, the choice of ordering heuristics was important to some extent to save time while it had little impact on the depth of the search for small N. It was more significant as N becomes large (N=18), where ascending value ordering was better for FC while min-conflicts gave better result for MAC. The problem is efficient solvable for solvers which support allDifferent constraints on rows and columns. Hence, the binary constraint solver may not give the optimal time.

3.3 Discussion

In terms of the implementation, there are still some limitations that exist. Due to the design which does not use tree structure directly, it has a larger space complexity compared to fully tree based design. The solver is also specialised to find a single solution if one exists. Hence, improvement can be made to extend the solver to support this functionality. Nonetheless, it supports the key functionalities and extension has allowed to

be more flexible and powerful to conduct empirical evaluation by using various heuristics. The testing environment is designed to be user friendly in terms of use with new problems and output generation for post analysis.

3.4 Conclusion

In conclusion, the practical delivery included design and implementation of a binary constraint solver supporting two solving algorithms. The key design involves the use of recursive method design with helper functions that fully exploits the given information sourced from a csp problem instance. Empirical evaluation is delivered by structured performance metrics and reporting process under a user-oriented testing environment.

The comparison has shown that the algorithms are comparable in terms of time and depth of the search while MAC leverages Arc Consistency Enforcement and outperforms FC for robustness and identifying satisfiability of the problem. The outcome has also shown that the result is also dependent on the variable and value assignment ordering heuristics, which are also evaluated using various problem instances.

Table 3.4: *Summary Output of 3 Instances of Quasigroup Completion Problem using dynamic variable ordering and value ordering heuristics, ranked by time and size of N*

	N	Type	Time1 (ms)	Depth	Count2	Var	Val
1	4	FC	0.13	17	33	sdf	minconf
2	4	FC	0.13	17	33	domdeg	asc
3	4	FC	0.14	17	33	domdeg	minconf
4	4	FC	0.16	17	33	sdf	asc
5	4	FC	0.18	17	33	brelaz	asc
6	4	FC	0.19	17	33	brelaz	minconf
7	4	MAC	0.23	16	336	brelaz	minconf
8	4	MAC	0.23	16	336	domdeg	asc
9	4	MAC	0.24	16	336	brelaz	asc
10	4	MAC	0.24	16	336	domdeg	minconf
11	4	MAC	0.32	16	336	sdf	asc
12	4	MAC	0.39	16	336	sdf	minconf
13	5	FC	0.23	26	51	domdeg	asc
14	5	FC	0.24	26	51	brelaz	minconf
15	5	FC	0.25	26	51	sdf	minconf
16	5	FC	0.27	26	51	brelaz	asc
17	5	FC	0.31	26	51	domdeg	minconf
18	5	FC	0.34	26	51	sdf	asc
19	5	MAC	0.60	25	672	brelaz	minconf
20	5	MAC	0.61	25	672	domdeg	asc
21	5	MAC	0.70	25	672	brelaz	asc
22	5	MAC	0.70	25	672	domdeg	minconf
23	5	MAC	0.77	25	672	sdf	minconf
24	5	MAC	0.93	25	672	sdf	asc
25	18	FC	1,845.90	563	1,363	domdeg	asc
26	18	FC	1,996.39	563	1,363	sdf	asc
27	18	FC	2,001.62	592	1,450	brelaz	minconf
28	18	FC	2,146.38	563	1,363	brelaz	asc
29	18	FC	2,285.97	592	1,450	sdf	minconf
30	18	FC	2,463.06	592	1,450	domdeg	minconf
31	18	MAC	14,933.38	364	106,083	sdf	minconf
32	18	MAC	14,950.14	364	106,083	brelaz	minconf
33	18	MAC	14,973.80	400	109,148	domdeg	asc
34	18	MAC	15,254.47	400	109,148	brelaz	asc
35	18	MAC	15,445.22	400	109,148	sdf	asc
36	18	MAC	16,267.06	364	106,083	domdeg	minconf