CS4402 - P2

200007413

November 2023

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Aims and Objectives

The aim for this submission was to integrate the foundational concepts and optimizations required to create a constraint solver.

The objective therefore was to create a working constraint solver implementing forward checking, maintenance of arc consistency, variable and value heuristics, whilst using binary constraints and 2-way checking.

1.2 Code Structure

This submission uses the Java packaging tool maven. This required using a nested folder structure in the `src` directory, with a dummy com.example domain. Inside this `src/com/example` folder is the generators, along with the `cspsolver` package containing all of the required files for the solver.

Along with those given in the starter code, there are three new files. The first is a the `Domain.java` file containing a class used to represent decision variable domains. The second is `BinaryARC.java` containing a class used to represent a singular dircted ARC.

Finally, the `BinarySolver.java` file contains the main solver class for this submission. It includes the main function allowing users to run the solver with a given *.csp* configuration, along with the MAC and forward checking solving functionality.

Along with the Java code, the root directory of this submission contains the files `generator.sh` and `evaluator.sh` which were used to generate the empirical evaluation tables used in

chapter 3.

Chapter 2

Design and Implementation

The specification outlines certain design decisions that have already been made. The most important of these is the use of binary constraints. Binary constraints are a simpler, but somewhat out-dated approach to CSP representation, where all constraints involve exactly two decision variables.

However, as has been taught in lectures, every non-binary constraint problem definition can be converted into a binary constraint instance. Therefore, the set of problems this solver can theoretically solve is not reduced.

Additionally, there are some beneficial implications of limiting to binary constraints. Enforcing node consistency is no longer required. Node consistency is a pre-cursor for maintaining arc consistency, and is defined as all singleton constraints being satisfied. As none of these exist, all CSPs defined for this submission will always be globally node consistent.

2.1 Domain Representations

The choice of how to represent decision variable domains was the very first design decision made for this submission. Research on the implications of different domain implementations suggests that each representation comes with it's own positives and negatives in terms of the time and spacial complexity for the different operations that are often done on the domain during constraint solving [1].

For this implementation, the decision was made to implement sparse-sets. These require storing two arrays and a size value for each domain. The first array *domain* contains every value in the domain. The second array *map* is indexed by the domain value, and

gives the index into the domain array for that value. The first size elements of the domain array make up the current state of the array. To remove a value, it is swapped in the domain array with the value at index (size - 1), and then the size value is decremented, effectively pushing it just out of range of the current domain.

This removal procedure is extremely efficient for backtracking. During pruning, x number of elements will be removed from the domain, and will be the x number of elements adjacent to the first *size* elements of the domain. Therefore, the pruning can be undone by just storing the *size* value before pruning, and then restoring that value upon backtracking.

2.2 Maintaining ARC Consistency (MAC)

Maintaing global arc consistnecy is a technique for ensuring that the domains for all the decision variables are consistent with the constraints.

There are two situations where MAC is required: (1) before any solving is done, MAC is required to restrict the initially given domains so that only potentially valid values are in the domain; (2) after any variable value is assigned, the change needs to propagate.

A naive algorithm (AC1) could achieve this for both scenarios by adding all ARCs to the queue, and propagating from there.

However, for scenario (2) the number of ARC revisions can be optimized. As the CSP is initially made globally ARC consistent and after every variable value selection, the CSP is guaranteed to be globally arc consistent before a variable value is assigned. Therefore, assignation is only changing the domain of the one variable assigned. This means we can re-establish global arc consistency by initially en-queuing just the ARCs pointing towards the variable assigned.

2.2.1 ARC Representation

This implementation utilizes ARC revisions so that domain changes only require revising the domains of variables who have constraints involving the variable whose domain was updated.

```
public boolean check(int sourceVal, int supportVal) {
    if (isForwards) {
        return this.constraint.check(sourceVal, supportVal);
    } else {
        return this.constraint.check(supportVal, sourceVal);
    }
}
```

An ARC represents one half of a constraint. A constraint is an un-directed connection between two decision variables represented in this submission as a Class with a first and second variable in the order given in the configuration file. A constraint forms two ARCs, one pointing from the first variable to the second, and one pointing the other way. When ARC revision happens, the domain values in the source variable are checked to make sure there is a supporting value in the support variable domain. If not, the value is removed. The above code shows the code for checking a given value from a source variable domain and support variable domain for an arc. If the ARC is pointing in the reverse direction, then the constraint check must be reversed (as the source is the second variable, and support the first).

2.2.2 Queue Representation

The queue represents all of the ARCs that need to be revised to achieve global arc consistency. However, when revising arcs, additional ARCs related to the variable's domain that was updated are added. This means that having an accessible queue is important. Additionally, the queue should be a set, as if an ARC needs to be revised, but is already part of the queue, then adding it is redundant.

Initially considered was the `LinkedHashSet`. This represents an ordered set, maintaining the order the elements were added to the set, as well as automatically preventing duplicate elements. However, in the version of Java used, the only way to access these elements in order is to convert the set into an interator. These iterator instances cannot be added to in the same way with the new elements, meaning a new linked hash set would have to be created, making sure to not add ARCs that exist in the iterator to this new queue.

Instead, it was decided that just using an `ArrayList` would be sufficient. This maintains the order elements were added, and allows elements to be added and removed at any time. However, it required the additional overhead of searching through the list for duplicates when adding to the queue.

2.3 Heuristics

Variable Selection

The specification required two variable heuristics which are used to determine which unassigned variable to assign next when required.

The first of these is ascending, meaning variables are chosen in the order given in the

CSP configuration file.

However, as we are using 2-way branching, it was decided that this order would act as a queue, where initially the first variable is chosen, and if the value chosen leads to an empty domain, then the second variable is chosen. Once all variables are selected, the ordering is resetf.

To achieve this in implementation, an array list was used as a queue, where the first element of the array is removed, and if propagation fails, then added to the end of the array.

The second variable heuristic is "smallest-domain first". This looks through the domains of all unassigned variables, and always selects the variable with the fewest elements. This is to hopefully find domain wipeouts faster.

This implementation required looping through each unassigned variable, and checking the size attribute of the associated domain.

Value Selection

Only one value heuristic was required by the specification. This is "ascending" selection, which chooses domain values in increasing order. To implement this with the sparse-set domain representation used for this submission, an additional lower bound metric had to be maintained.

This is because the domain in sparse sets is not ordered (as the mapping array gives the positions of elements). To maintain this value, it is first initialized with the lower bound given by the CSP configuration. However, when a value is removed, the whole domain has to be searched.

Chapter 3

Empirical Evaluation

In this section, the results of an empirical analysis of the constraint solver will be shown and discussed. For the performance metrics for the instances given in the starter code, see Table A.1.

3.1 Problem Classes

Given in the starter code for this practical were CSP configuration generators for different classes of problems. The following is an analysis of how the different parameters and configuration options impacts the performance of the solver.

NQueens

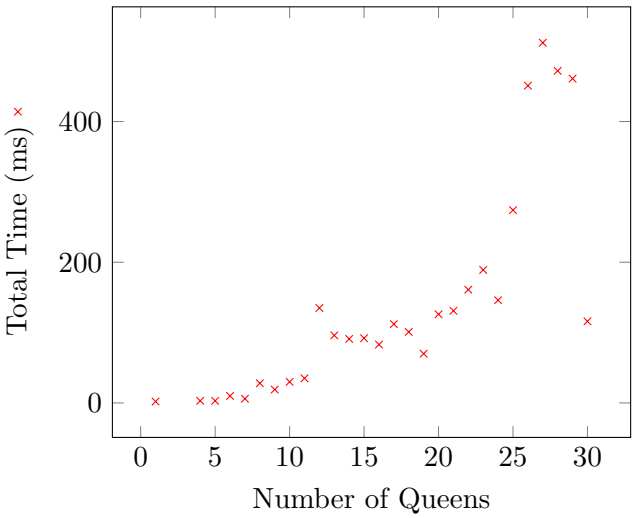

Figure 3.1: Graph to show the relationship between the number of queens in the N-Queens problem class and the amount of time taken to solve the instance.

As shown in Figure 3.1, in general, the more queens, the longer the solver takes to solve. This appears to be exponential, which makes intuitive sense, as adding an additional queen to the problem requires expanding the width and height of the board by 1, thus exponentially increasing the number of squares on the board (the decision variable domains size).

Additionally, another intuitive relationship is between the number of nodes the solver passed, and the number of ARC revisions made. Every time a variable is assigned a value, the solver must re-establish global arc consistency, thus requiring more ARC revisions. A node represents either a left branch where a value is chosen, or a right branch where a different value must then be chosen. The more nodes required, the more values had to be assigned, and so the more ARC revisions required.

The 30Queens instance is an interesting outlier, taking just as long as the 15Queens problem to solve.

3.2 Variable Heuristics

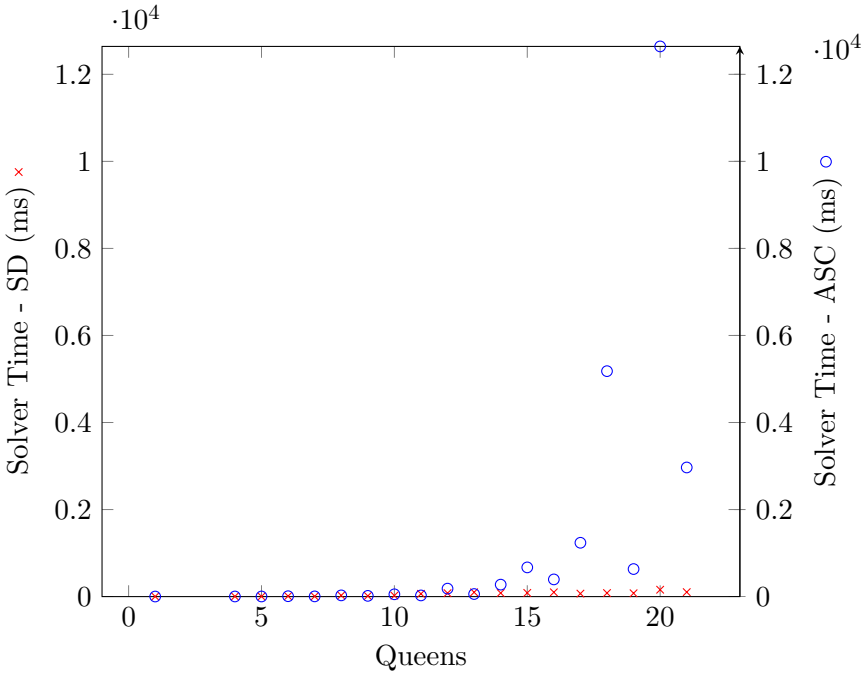

Figure 3.2: Graph to show the difference in performance between the smallest-domain first (SD), and ascending (ASC) variable heuristics.

Table A.3 and Figure 3.2 show that the choice of variable heuristic can have a huge impact on the performance of the solver. For the NQueens problem, choosing the variables with the smallest domains improved the total time by a factor of up to 100 for the larger instances, requiring far less ARC revisions and searching far fewer nodes.

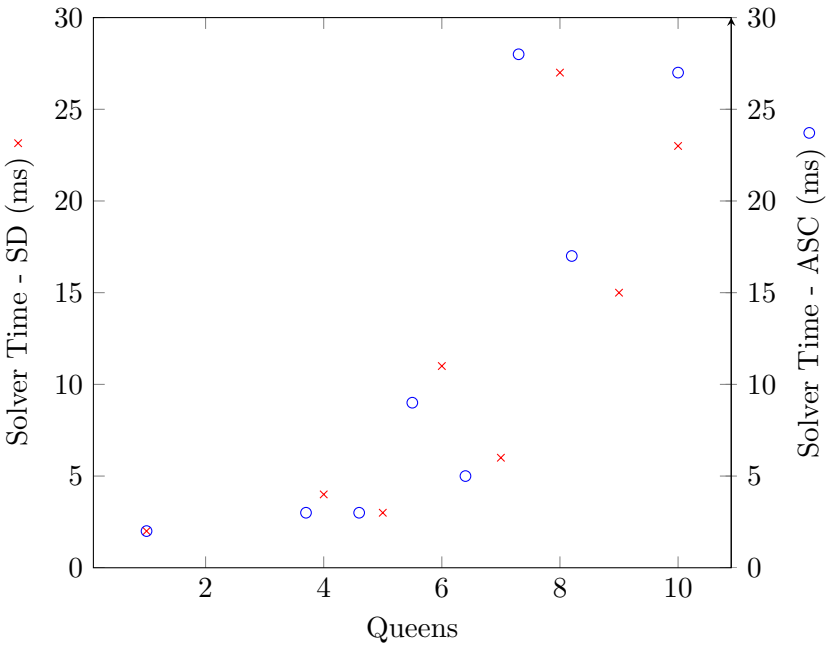

Figure 3.3: Zoomed in version of Figure 3.2, showing that the heuristic choice only becomes relevant once the instance becomes sufficiently large.

However, Figure 3.3 shows that the choice of variable heuristic is less important when the instance is small enough to be solved very quickly. There are most likely other external factors causing these results to vary, such as background CPU load, overwhelming the differences caused by the variable heuristic.

Chapter 4

Conclusion

Overall this submission successfully meets the required objectives for implementing a functioning binary constraint solver implementing the still relevant AC3 algorithm, forward checking, with a modern domain representation.

Further improvements to the code would be to introduce different value heuristics. The sparse-set representation of domains requires a lot more overhead when using the ascending value heuristic, which could be avoided if a different heuristic not requring the lower bound to be tracked was used.

Bibliography

[1] Vianney Le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. 2013. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*. Uppsala, Sweden, (Sept. 2013), 1–10. `https://hal.science/hal-01339250`.

Appendix A

Performance Metric Tables

Starter Code Example Performance

Table A.1: Table showing the solver's performance metrics with the instances given in the starter code.

Instance	Solver Nodes	ARC Revisions	Total Time (ms)
10Queens.csp	24	910	25
4Queens.csp	6	45	4
6Queens.csp	26	326	10
8Queens.csp	34	862	21
FinnishSudoku.csp	7,682	$6.59 \cdot 10^5$	2,095
langfords2-3.csp	6	105	3
langfords2-4.csp	8	260	8
langfords3-10.csp	4,606	$7.68 \cdot 10^5$	7,179
langfords3-9.csp	1,348	$2.06 \cdot 10^5$	1,951
SimonisSudoku.csp	81	5,053	71

Queens Class Performance

Table A.2: Table showing the performance metrics for the solver with the first 30 n-queens instances.

Queens	Solver Nodes	ARC Revisions	Total Time (ms)
1	1	0	2
4	6	45	3
5	5	68	3
6	26	326	10
7	9	199	6
8	34	862	28
9	18	569	19
10	24	910	30
11	46	1,679	35
12	93	3,539	135
13	111	5,165	96
14	54	3,344	91
15	17	1,780	92
16	25	2,591	83
17	51	4,137	112
18	48	4,737	101
19	28	4,059	70
20	98	8,602	126
21	35	5,049	131
22	26	5,127	161
23	46	7,623	189
24	40	7,937	146
25	175	22,618	274
26	166	23,826	451
27	54	11,802	512
28	28	10,223	472
29	29	11,318	461
30	30	1,505	116

Variable Heuristic Comparison (using NQueens)

Table A.3: Table showing the comparison of the smallest-domain first (SD) variable heuristic and acending (ASC) variable heuristic in terms of solver performance metrics.

Queens	SDNodes	SDARCs	SDTime	ASCNodes	ASCARCs	ASCTime
1	1	0	2	1	0	2
4	6	45	4	6	45	3
5	5	68	3	5	66	3
6	26	326	11	51	292	9
7	9	199	6	7	166	5
8	34	862	27	182	1,352	28
9	18	569	15	44	671	17
10	24	910	23	226	3,286	53
11	46	1,679	64	46	972	27
12	93	3,539	79	1,010	13,903	183
13	111	5,165	99	54	1,450	59
14	54	3,344	82	2,163	39,881	275
15	17	1,780	85	6,646	$1.29 \cdot 10^5$	671
16	25	2,591	99	2,625	61,831	395
17	51	4,137	69	11,255	$3.11 \cdot 10^5$	1,237
18	48	4,737	82	59,451	$1.75 \cdot 10^6$	5,180
19	28	4,059	77	2,998	91,468	633
20	98	8,602	158	$1.2 \cdot 10^5$	$3.96 \cdot 10^6$	12,641
21	35	5,049	103	19,265	$6.56 \cdot 10^5$	2,967