# CS4402 - Practical 2
# Constraint Solver

150025289

April 2017

## 1 Introduction

The aim of this project to implement a generic Forward-Checking (FC) based constraint solver. The solver will consist 2-way branching mechanism and be able to work with different heuristics (static or dynamic ones).

In our project, we first implemented d-way branching FC first then implemented 2-way branching FC. In addition to those two, we extended our FC by MAC. We also wanted to achieve a generic solver which can work on different problems so, we designed our solver accordingly and tested our solver on 3 different problems (graph colouring, sudoku and futoshiki).

On one of the problem cases (sudoku), we made an empirical evaluation to analyse the effectiveness of different heuristics and 2 different types of solver.

## 2 Design

### 2.1 General

We designed our project in an object-oriented way and used `python 3` as programming language to implement it. Our solver project consist the integration between different set of structures. These are;

- a problem,
- variables,
- constraints and
- heuristics.

We are going to explain these terms first and then define our solver mechanism.

### 2.2 Problem

A problem consists the given input, the required constraints to satisfy and wanted output. In our system, we designed every input and output as *variables* and created *constraints* between them.

## 2.3 Variables

The variables consist a value (which can be empty as well) and the domain which has the possible values can be assigned. We mentioned that we are using variables for input values and output values so, when an input value is given we also set the domain to have that only value in the beginning of its creation. This will happen before sending variable to the solver, so whatever happens this input values will only be able to take this only value. For instance in Sudoku, for a given non-empty cell in the specification, we set the value and the domain of that variable to the given value directly.

To identify variables, we added an `id` field to them. Since we are using Sudoku as our main problem, we used a tuple with two integers as id. By doing this, we can mark the positions by row and column such as $(0,0)$ or $(4,2)$. Instead of using an integer id, using a tuple of two integers as id is not a limitation for other problems. For instance, if we want to use our variables on graph colouring problem without any orientation of row or columns, we can simple use one of the integers and set the other one to 0 for all the variables. Since the solver only use the ids to access the variable, only restriction about them is they should be unique.

Variables also have a heuristic value which is calculated by heuristics calculator and this helps them to have order when they are sorted in a list.

## 2.4 Constraints

FC Solver only works on binary constraints and our problem instances only have binary constraints since we also designed the input values as variables. So, we can define constraints as two constraints and the operation between them. In this manner, we can use two variables $(v_1, v_2)$ tuple as the key to a mapping of an operation. This tuple will be also unique since every variable is unique and we will be able to access this two variable and the two way arcs between them by having this constraint entry.

We wanted to avoid duplicate constraints which can occur as the form of $(v_2, v_1)$ in addition to $(v_1, v_2)$. To solve this possible issue, during the creation of the problem instance we only create constraints of $(v_i, v_j)$ where $i < j$. By doing this, we avoid these possible redundant constraints. For sudoku example, while we have $v_{(0,0)} \neq v_{(0,1)}$, we don't have $v_{(0,1)} \neq v_{(0,0)}$ as constraint defined in problem.

## 2.5 Heuristics

In our project, we used different variable assignment heuristics which can be static or dynamic. For static ones (in which the order of the assignments are set before solving the problem), we used;

- Maximum Degree and
- Maximum Cardinality.

Implementation of these two heuristics are pretty straight forward. For maximum degree heuristics we went through all the constraints and counted the edges for each variable.

For Maximum cardinality, we started from a random point and iterate through the variables with the most connection to the assigned group.

For dynamic heuristics (in which the order may change depending on the variables' situation on solver run-time), we implemented;

- Minimum Domain and
- Brelaz.

We create our heuristics specifically on the problem instance and depending on its static or not it works differently. Static heuristics directly calculate the heuristic value when they are initiated and write the heuristic value to the variable. In the other hand, dynamic heuristics calculate initial heuristic values for variables but the solver ask the heuristic calculator to update the values. For this purpose, we implemented a Dynamic Heuristic indicator (flag) for each heuristics so that, solver can identify which heuristics is dynamic and depending on that, it can ask to update the values to the heuristics.

## 2.6 Solvers

We have 2 solvers in our design. The first one is the main requirement for the project and the other one is implemented as an extension.

### 2.6.1 FC Solver

Our main solver is FC Solver which takes a problem instance and heuristics calculator (optional) and can run 2-way or d-way branch solving procedure. The algorithms are implemented based on the lecture slides and pretty much the same as described over there. They have a recursive structure which they can call itself after each assignment (or non-assignment in right branches). Only on arc-revision and pruning/un-pruning, some implementation decisions are taken.

In FC, when we assign a value (or cut-out one value from the domain in right branch in 2-way), we revise every arc$(v_f, v_c)$ in where $v_c$ is the current assignment (or non-assignment) and $v_f$ is the possible unassigned future variables. In our implementation, since we defined our constraints with $(v_i, v_j)$ where $i < j$ structure, we need to make sure that we find the constraint by searching the constraint in the right order and when we find, we need to make sure that we need to prune the correct one. This can be handled by searching the constraint in both ways and setting up a flag for which order is found. Additionally, even the problem instance consists redundant constraints for same constraint, the algorithm will find the last one and execute the constraint operation to prune the future variables domain correctly.

Another decision we took is about how to store prunes and how to undo pruning. To do this, we decided a store a prune map between variables and pruned domain so far. When a future variables domain is pruned, it accesses the prune map to find the old pruning domain list for that variable. If found, it adds the new pruned value to the list, otherwise it creates a new array to add this new value.

In case the recursive algorithm failed to find any solution or consistency of arcs show no domain for some variable the solver will execute the next line of code which indicates undo-pruning for each entry and re-initialisation of the prune map.

#### 2.6.2 MAC Solver

MAC Solver is actually pretty similar to FC Solver. However, there is a difference in arc-revision process. Thu, we can use our FC Solver as parent class and create our MAC Solver and only override required revision methods.

In FC, when we branch into one node, we only prune the arcs related to the assignment but in MAC, we iteratively go through every arc which might be affected by every pruning. For this, we keep a `set` in python and put variable tuples while paying attention to the which one is to be pruned.

## 3 Problem Cases

We have 3 problem instances. These are;

- Graph Colouring
- Futoshiki
- Sudoku

### 3.1 Graph Colouring Problem

Graph colouring problem was a initial point for our project to test our solver in smaller instances. We implemented this problem by mapping colours to integers and since the problem's upper domain bound in not fixed (minimisation optimisation problem), we implemented a mechanism in our solver to solve the graph colouring problems by finding the minimum $k$ value which indicates the number of colours which are used to colour the graph nodes.

The solving procedure of this minimisation optimisation problem is pretty simple. The solver starts from $k = 1$ and tries to find a solution. After every unsolvable result, it iterates with increasing k by 1. In the end, it will find a solution with the minimum $k$ value.

### 3.2 Sudoku

Sudoku is a very well known problem for constraint solvers and it is actually a specified graph colouring problem with fixed number of colours and some fixed colours for given vertex.

### 3.3 Futoshiki

Our solver accepts binary constrains and it can work on inequality constraints as well. To show this, we use Futoshiki. Futoshiki is usually $5 \times 5$ board with domain size of 5 for the variables. It requires different values for each column/row and additionally there are some $<$, $>$ constraints between some variables to take into account.

## 4 Test Cases

### 4.1 Graph Colouring Problem Cases

For graph colouring problem, we created one problem instance with 5 vertices and 6 edges. It can be seen in the figure 1. We created this instance by ourselves.

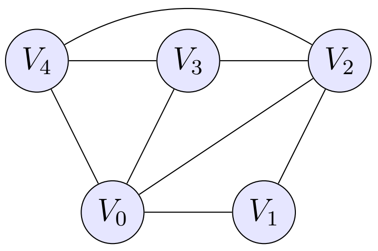

Figure 1: Our graph colouring problem instance with 5 vertices and 6 edges between them.

### 4.2 Sudoku Cases

For Suduko we used a total number of 47 instances which has been classified from easy to hard and has been used in previous researches [1] in addition to one empty board instance we created by hand. We did our evaluation using these 47 Sudoku instances. We acquire the mentioned instances in plain text format from the researchers' webpage[1] and parsed to create our problem instances to send to the solvers. These instances can be found in the `sudoku_instances` folder in the main directory of the project.

### 4.3 Futoshiki Cases

Since Futhoshiki is not our main interested problem, we only created one Futoshiki problem instance as we did in the graph colouring problem. This problem instance ($5 \times 5$) can be seen in the figure 2. We acquired this problem directly from the Wikipedia page of Futoshiki[2] and created the instance of it.

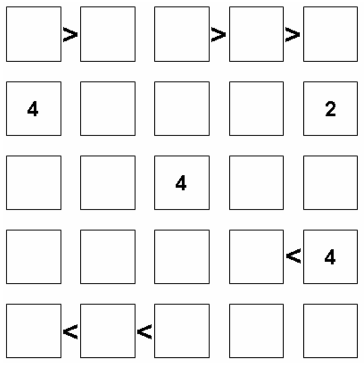

Figure 2: Our one and only Futoshiki instance to test our solvers capability of dealing with $<$ and $>$ type of constraints.

[1]`http://lipas.uwasa.fi/~timan/sudoku/`

[2]`https://en.wikipedia.org/wiki/Futoshiki`

## 5 Results

### 5.1 Graph Colouring Problem Results

We solved our only problem instance instance using FC with dynamic variable heuristics Brelaz and it found out that 4 colours is the minimum number for this instance of minimisation optimisation problem. The result can be seen in the figure 3.

```
Min solution is found in 4

V_1 = 2
V_3 = 2
V_0 = 1
V_2 = 0
V_4 = 3
```

Figure 3: The result output for our only graph colouring instance.

This experiment can be reproduced by running `runner_graph.py`

### 5.2 Futoshiki Results

We solved our only problem instance instance using FC with dynamic variable heuristics Brelaz and it found a satisfactory solution which can be seen in the figure 4.

```
5 4 3 2 1
4 3 1 5 2
2 1 4 3 5
3 5 2 1 4
1 2 5 4 3
```

Figure 4: The result output for our only Futoshiki instance.

This experiment can be reproduced by running `runner_futoshiki.py`

### 5.3 Sudoku Results

For 47 different Sudoku instances, we executed our FC and MAC solvers on 4 different heuristics we mentioned. For evaluation we used 3 different metrics;

- Node size,
- Time taken and
- Number of arc revisions.

For an easy Sudoku instance, the results of node size can be seen in the figure 5. If we compare the heuristics first before differentiating the solvers, as we can see Maximum Degree Heuristics and Maximum Cardinality Heuristics shows worse results compared to the dynamic variable heuristics Min-Domain and Brelaz. We need to point out that, since every variable has the same degree in sudoku, to overcome randomness which can cause abysmal results, we broke the equality by ordering equalities by the id of the variable.

Since it's an easy instance node size for Brelaz and Min-Domain is 81 which indicates no backtracks. For this instance Brelaz takes more time than Min-Domain since it has less calculation per node. Brelaz also calculates the future degree values which takes a little bit of time and in the same node size environment Min-Domain takes less time because of that. The number of arc revised is also really less for the dynamic ones compared to the static ones.

If we compare FC with MAC, we can see that MAC does so many arc revisions however, since it's an easy case it doesn't make node size less and the time taken is also disadvantaged in this case.

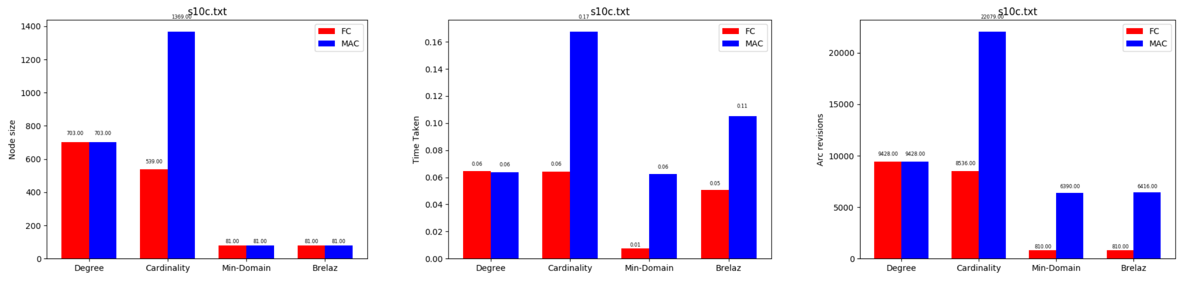

Figure 5: The results for the easy Sudoku instance `s10c`.

For a hard instance, we can see the results in the 6. In this example, Brelaz performed better in node size but it didn't save up enough time to beat Min-Domain heuristics. Maximum Degree Heuristics and Maximum Cardinality Heuristics performed worse than dynamic heuristics ones.

If we compare FC with MAC for this example, MAC actually makes the the node size smaller in this case but time FC is still time efficient. The number of size is more in MAC as expected.

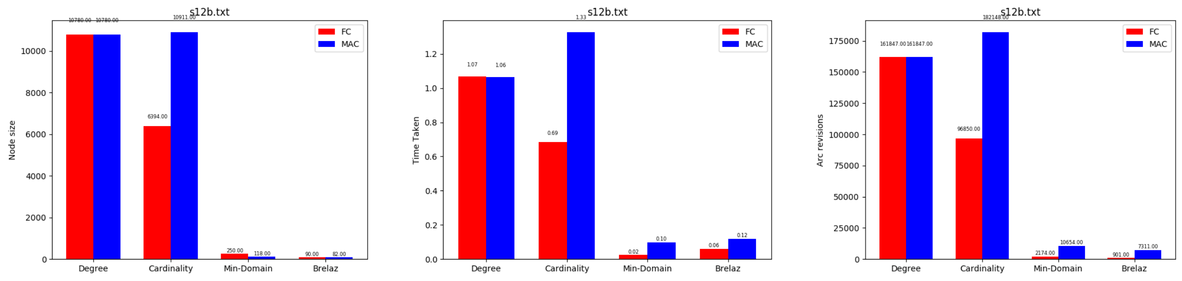

Figure 6: The results for the hard Sudoku instance `s12b`.

When we take the median average of the all Sudoku problem instances and plot the graphs, we end up with the results in the figure 7. We used median instead mean average to favour the values in the values which are similar, disfavour some extraordinary cases and reduce their effect on the results.

In FC, the average results show that Maximum Degree heuristics performs the worst. This is really expected because Sudoku has no differentiation in degree values and this is basically the same as the no heuristics at all. In average, Maximum Cardinality perform slightly better than Maximum Degree in FC since it tries to choose the new variable depending on its number of connection to the group which are already assigned and tries to find the most constraint ones in future in static way. The dynamic heuristics Min-Domain and Brelaz perform really well on average and they managed to help solving the problems under 100 nodes on average.

If we want to make a comparison between Min-Domain and Brelaz, we can indicate that the nodes traversed are actually quite same but Brelaz spends more time in calculations so we can safely point out that, Min-Domain is actually better choice for Sudoku solving.

Comparing between FC and MAC, on average, MAC performs better in node size but it doesn't mean that it's better in the terms of time. The average number of nodes explored is reduced by MAC but there is no gain of time contrarily there's an extra time requirement. This time requirement can be pointed out in the number of arc revisions which MAC does a lot much more compare to FC. Depending on these results, we can

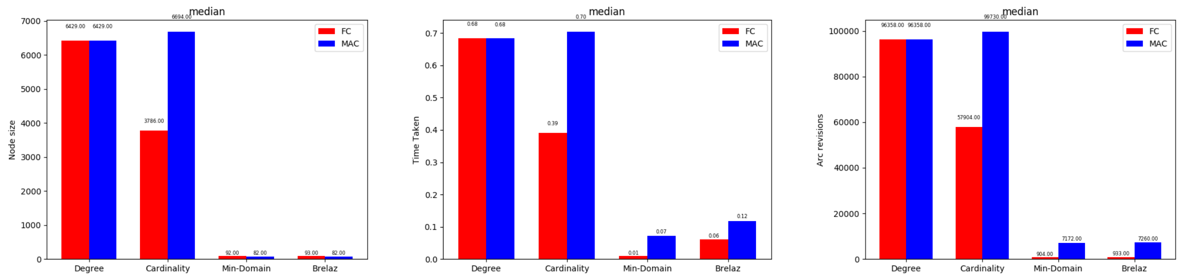

Figure 7: The median average results for the all the Sudoku instances.

say that the usage of Brelaz instead of Min-Domain is not necessarily better and the gain of MAC doesn't affect too much on node size that in the terms of time it's less efficient. This applies on Sudoku problem only of course. Additionally, these differences can be caused by the decision choices on implementation.

All of the results can be re-generated using `runner_sudoku.py`. Additionally, the results are also dumped using pickle to `results.dat`. This object can be loaded and the results can be acquired in python interpreter. We also printed out the results in `results.txt` and plotted out all the tests using matplotlib (check `plotter.py` for the code) and saved to `sudoku_plots` folder so that, results can also be seen without any python requirement.

In addition to all those, it is possible to run only one sudoku instance with Brelaz and FC. To do this, `runner_sudoku_one.py` can be executed. I will work on `s12a` hard instance.

## 6 Conclusion

In conclusion, this project was about creating a reliable constraint solver with the flexibility of using different set of heuristics and even different algorithmic procedures. We implemented Forward-Checking algorithm with 4 different heuristics and compared the results achieved by them on Sudoku instances. We also extended our solver by implementing MAC and compared it with FC to see the different outcomes.

We also showed that our solver is capable of working on different set of problems. We demonstrated the usage of inequality constraints on Futoshiki problem and the minimisation optimisation problem solving procedure with Graph Colouring Problem.

## References

[1] Timo Mantere and Janne Koljonen. Solving and analyzing sudokus with cultural algorithms. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 4053–4060. IEEE, 2008.