Contents

CS4402 Practical 2

170011474

November 2020

1 Overview

This practical requires students to implement solvers based on forward checking and maintaining arc consistency in two-branch way. Empirical experiments are expected to demonstrate comparisons of 2 approaches.

2 Design and Implementation

2.1 Data Structures

2.1.1 Variable

Each `Variable` has its own id and its assigned value. It also records its possible domain. The domain has a type `LinkedHashSet` for integers, which would naturally keep elements' ascending order. In this way, ascending value assignment would be benefited from `LinkedHashSet`'s properties and only an iterator is required to return the next value from its domain.

2.1.2 Arc

An `Arc` connects between a dependent and a supporter. The dependent is normally a future variable to be pruned. In its function `prune()`, the future variable's domain is updated by removing unsupported values based on existing constraints.

2.1.3 Pruning

This class is responsible for each pruning process. It keeps a variable which is pruned and a `LinkedHashSet` to keep all removed values from its domain. The reason for this is when undoing a pruning is needed, the variable can quickly add those removed values to its domain.

2.1.4 BinaryConstraint and BinaryTuple

This class is provided by lectures but is extended. Given a domain, a constraint filters the domain based on constraint tuples it has. A `BinaryTuple` contains 2 values to indicate they can be assigned together.

2.1.5 BinaryCSP

It is a class to hold a CSP problem, including its constraints and domain bounds. This is extended to have a `constraintsMap` with a type `LinkedHashMap<Integer, LinkedHashMap<Integer, BinaryConstraint>>` as a "database". It can look up for a `BinaryConstraint` based on the constraint's 2 variable ids. It could also return a set of variables with associated domains initialised.

2.2 Base: Binary Branch Solver

The procedure to do 2-way branching:

```
Procedure solve():
    if(completeAssignment())
        printSolution()
        exit()
    nextVariable = selectVar(futureVars)
    nextValue = selectVal(nextVariable)
    branchLeft(futureVars, nextVariable, nextValue)
    branchRight(futureVars, nextVariable, nextValue)
```

For `selectVar()`, it could be ascending variable order or smallest domain first heuristic. For `selectVal()`, it directly uses ascending value order to assign. `branchLeft()` procedure:

```
Procedure branchLeft(futureVars, nextVariable, nextValue):
    assign(nextVariable, nextValue)
    if reviseFutureArcs(futureVars, nextVriable):
        solve()
    undoPruning()
    unassign(nextVariable, nextValue)
```

In `branchLeft()`, the `nextValue` is assigned to `nextVariable` where `nextVariable` is removed from `futureVars`. If `reviseFutureArcs()` is successful, then continue to a new `solve` node. At the end, recovering is need by undoing pruning and unassign. `branchRight()` procedure:

```
Procedure branchRight(futureVars, nextVariable, nextValue):
    removeFromDomain(nextVariable, nextValue)
    if (nextVariable.isConsistent()):
        if reviseFutureArcs(futureVars, nextVriable):
            solve()
        undoPruning()
    addToDomain(nextVarible, nextValue)
```

In this procedure, the right branch is the way without the `nextValue`, so removing it from `nextVariable`'s domain. Before the next step, check if the chosen `nextVariable` is consistent and then prune future arcs by `reviseFutreArcs()`. Afterwards, doing a recursive `solve()`. At the end, some recovering things are required: undoing pruning and add the removed value to its domain.

`reviseFutureArcs()` function is implemented in `ForwardChecking` and `MaintainingArcConsistency` respectively.

2.3 Forward Checking

The Forward Checking solver extends `Solver` and it only needs to implement `reviseFutureArcs()`. The procedure is:

```
Procedure reviseFutureArcs(nextVariable, prunings):
    Foreach futureVariable in futureVars where futureVariable != nextVariable:
        prunings.add(prune(Arc(futureVariable, nextVariable)))
        if(!futureVariable.isConsistent()):
            return false
    return true
```

For each future variable, construct a new `Arc` where the future variable is the dependent variable to be pruned. The removed variables from its domain would be kept in a new `Pruning` for future undoing. If there is any future variable not consistent (i.e. domain is empty) then return false.

2.4 Maintaining Arc Consistency

The `reviseFutureArcs()` in this solver is:

```
Procedure reviseFutureArcs(nextVariable, prunings):
    queue = all Arcs(x_i, x_j)
    while not empty(queue):
        prunings.add(prune(Arc())
        futureVariable = arc.getDependent();
        if(!futureVariable.isConsistent()):
            return false
        if(isPruned()):
            Foreach futureVariable in futureVars:
                queue.add(Arc(futureVariable, nextVariable))
    return true;
```

The `queue` holds all `Arc`s which connect to `nextVariable` to be pruned. For each `Arc` in the `queue`, we prune it and then check if the `futureVariable` (the `dependent` in the `Arc`) is consistent. If its domain is pruned / updated, then added all `Arc`s which are connecting to the `dependent` to the `queue` for further operations.

The `queue` is implemented by `LinkedHashSet` because it can keep `Arc`s' insertion order and also `Set` ensures that no duplicate `Arc` would appear. Besides, an `iterator` can be used to iterate over each `Arc` and remove the current element from the queue easily.

3 Empirical Evaluation

3.1 Methodology

This practical uses testing instances provided by CS4402 and also instances generated by generators, including Queens, Sudoku, and Langfords problems. Time taken to obtain the first solution, nodes in the search tree, and revision times are recorded as metrics to do comparisons and contrast of Forward Checking, Maintaining Arc Consistency with different variable assignment orderings: ascending variable ordering and smallest-domain first.The time for a solution would take the average number of 3 running times. For generated instances, testings are conducted by FC with smallest-domain first variable ordering strategy.

3.2 Results

Please notice that FC represents forward checking, and MAC represents maintaining arc consistency. In Variable Order, 1 is for ascending variable order, and 2 is for smallest-domain first order.

Results					
Instance	Solver	Variable Order	Solution Time(ms)	Nodes	Revisions
langfords2_3	FC	1	3	13	41
langfords2_3	MAC	1	6	8	101
langfords2_3	FC	2	3	13	41
langfords2_3	MAC	2	6	8	101
langfords2_4	FC	1	7	33	142
langfords2_4	MAC	1	11	11	312
langfords2_4	FC	2	7	33	141
langfords2_4	MAC	2	10	11	312
langfords3_9	FC	1	350	3537	38901
langfords3_9	MAC	1	213	61	16057
langfords3_9	FC	2	215	1662	19164
langfords3_9	MAC	2	143	29	6743
langfords3_10	FC	1	1469	15118	185121
langfords3_10	MAC	1	644	154	52671
langfords3_10	FC	2	824	6677	84546
langfords3_10	MAC	2	375	70	23028
4Queens	FC	1	1	9	18
4Queens	MAC	1	2	6	27
4Queens	FC	2	1	9	18
4Queens	MAC	2	2	6	27
6Queens	FC	1	5	27	96
6Queens	MAC	1	8	10	155
6Queens	FC	2	4	27	96

6Queens	MAC	2	8	10	155
8Queens	FC	1	9	81	366
8Queens	MAC	1	14	19	565
8Queens	FC	2	9	72	317
8Queens	MAC	2	14	19	565
10Queens	FC	1	8	81	415
10Queens	MAC	1	14	24	746
10Queens	FC	2	6	33	164
10Queens	MAC	2	11	14	460
SimonisSudoku	FC	1	26	189	2553
SimonisSudoku	MAC	1	103	178	9641
SimonisSudoku	FC	2	12	82	810
SimonisSudoku	MAC	2	64	82	6232
FinnishSudoku	FC	1	2034	109397	1635146
FinnishSudoku	MAC	1	8912	84248	5204602
FinnishSudoku	FC	2	317	10123	93475
FinnishSudoku	MAC	2	768	2918	313740

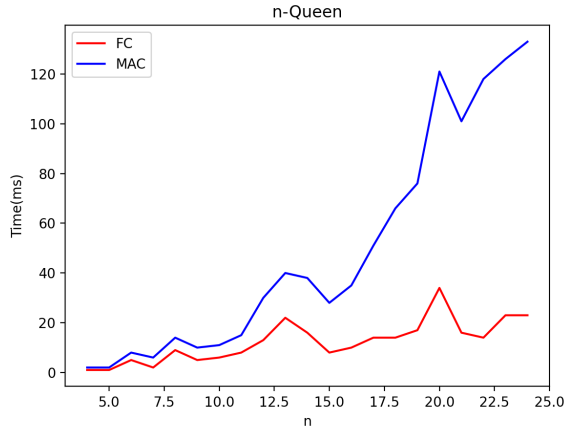
3.3 Evaluation

There are 2 main factors that would affect the solver performance most: variable domains and constraints. From the table, as the problem size goes up, the solver time would increase, nodes and revision times also increase dramatically.

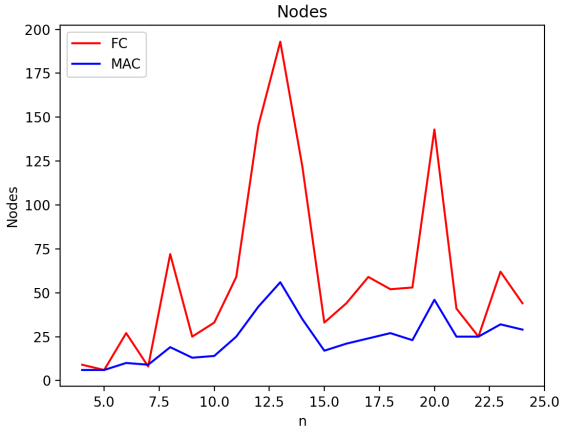
In terms of solution time, in problem class langfords, for instance `langfords2_3` and instance `langfords2_4`, the best solver is FC. For further difficult instances, the MAC with smallest-domain first variable strategy dominantly wins. In problem class n-Queens, the FC solver performs better than the MAC solver. For variable strategy, smallest-domain first variable ordering works better. In problem class sudoku, the FC with smallest-domain first variable ordering strategy dominantly wins the game.

An overall conclusion can be drawn from this table: *smallest-domain first domain strategy generally performs much better than the traditional ascending variable ordering in every problem class.* Besides, Forward Checking (FC) strategy generally performs better than MAC. To gain a deeper insight, for each problem class, I used generators to generate new instances.

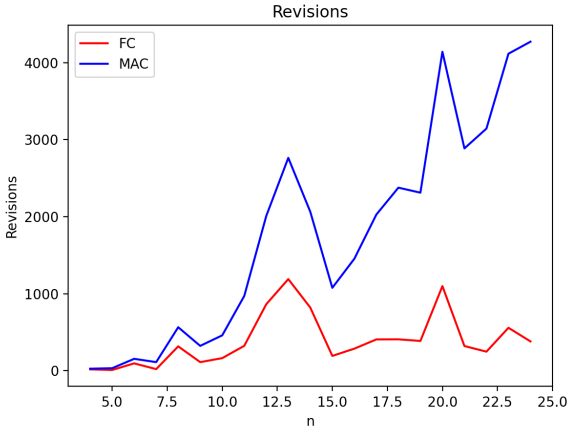
3.3.1 N-Queen Problem



(a) Time(ms)



(b) Nodes

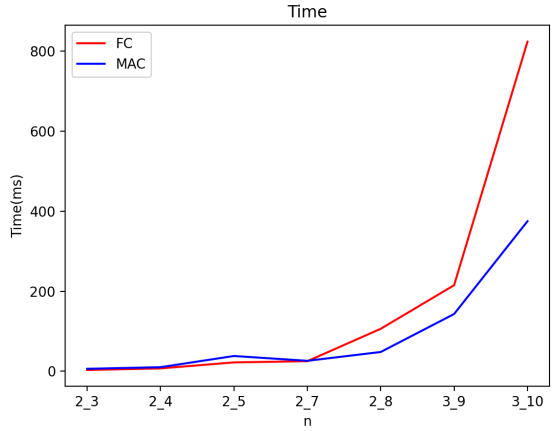


(c) Revisions

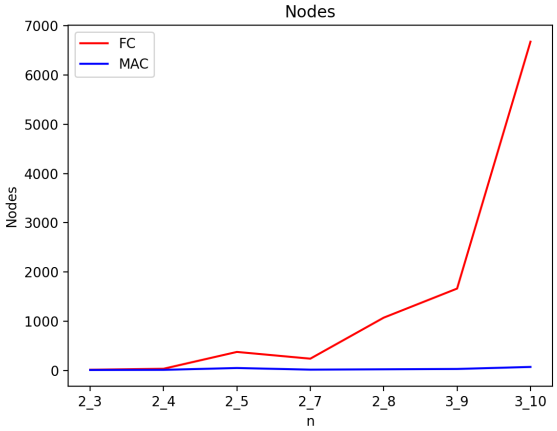
Figure 1: N-Queen Problem

Those figures show that, as `n` increases, MAC would take more time and more revisions, but FC would search more nodes. But overall, FC would give a solution with less time.

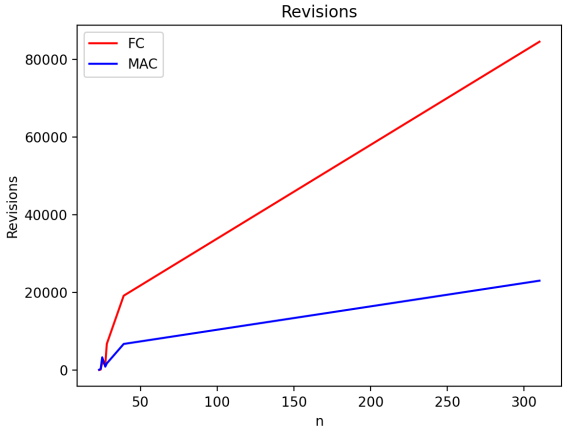
3.3.2 Langford's Problem



(a) Time(ms)



(b) Nodes

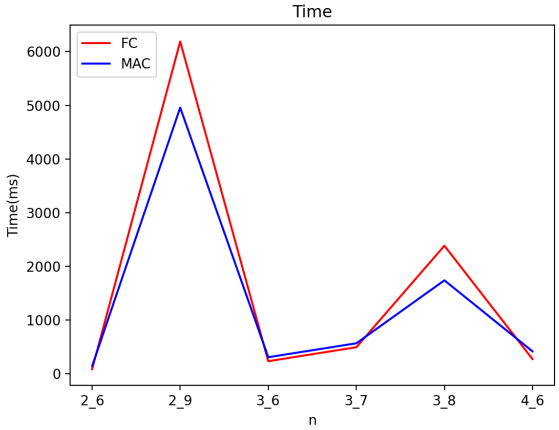


(c) Revisions

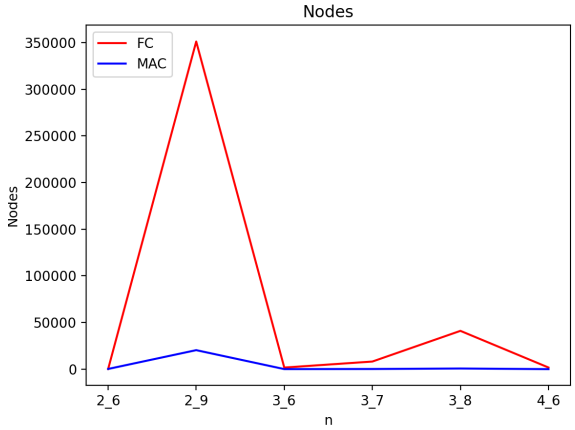
Figure 2: Langfords Problem Solvable Instances

As we can see from those figures, for solvable instances in Langford's problem class, if values' domains become very large or variables are well connected to each other, then MAC would take less time than FC. FC generally takes more nodes and more revisions.

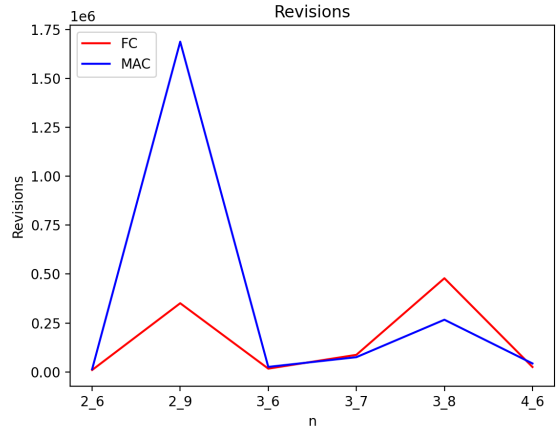
When instances are non-solvable, MAC takes much less nodes and much less revisions to find out dead-ends. FC sometimes would take more time to notice dead-ends. The reason is the same as the one mentioned above: when variables are well connected, then a pruning would significantly reduce revision times as well as nodes to be further searched.



(a) Time(ms)



(b) Nodes



(c) Revisions

Figure 3: Langfords Problem Unsolvable Instances

3.3.3 Sudoku Problem

The most difficult sudoku in the world is the instance FinnishSudoku. One of the most easiest sudoku instances can be the generated sudoku instance without any modification. Based on some online sudoku problem generators OpenSky Sudoku Generator, I have constructed some sudoku instances in csp files. By using FC with smallest-domain first strategy, the most difficult one is the FinnishSudoku and other sudoku instances would take about 10+ms. In other words, FinnishSudoku instance is the upper bound of this problem class so the difficulty would vary inside this boundary. It is quite significant.

4 Conclusion

During this practical, I gained more insights into how a good constraint solver would improve efficiency of solving problems. Besides, I understand how a good heuristic like smallest-domain first matters.