CS4402 P2 - Constraint Solver Implementation

160004864

1 Introduction

The aim of this practical was to design and implement a constraint solver for binary constraints. The solver must employ 2-way branching and it should implement two algorithms, namely the Forward Checking and the Maintaining Arc Consistency algorithms. Furthermore, the solver should support two variable ordering strategies - a static, ascending variable strategy where the order is specified in the files describing the constraint satisfaction problem (CSP) instance, and a dynamic, smallest-domain first strategy. In addition, the solver should support ascending value ordering. Lastly, we were required to carry out experiments to empirically analyse the performance of the solver on a number of CSP instances, and report three measures, the time taken for a solution to be found, the number of nodes in the search tree as well as the number of arc revisions.

2 Search

Since our solver is required to support domain pruning and a procedure to undo it, we may view our search space as a tree, in which a node is added whenever an assignment of a value to a variable or made or whenever a a value is deleted from a variables domain. For this reason, I have decided to use a Node data structure which encapsulates the current state of our search following either a variable assignment or a value deletion. This state includes a list which holds the variables that have yet to be assigned, as well as a map which holds the domain values for each of the variables following pruning. Having an instance of Node dedicated for each point of the search, allows us to keep track of the previous state, before any operation takes place, and revert back to it if these operations cause a domain wipeout.

2.1 Node Data Structure

The aforementioned Node data structure is implemented in the `Node.java` class. Since we need to maintain the state of our search in this data structure, every single instance of this class holds a list of the variables that have not been assigned yet at this stage of our search, in the `varList` attribute. Furthermore, we also need to keep track of the domain of each variable following pruning, which is held in the `variablesMap` map. In this map, the keys are the variables and the value for each key is a list which keeps track of the variable's supported domain values.

This data structure, implements several important methods that are used, regardless of which algorithm is used by the solver. These are described below:

- `completeAssignment`: This is the method that is used to check whether all the variables have been assigned a value. At this point, the solver reports the solution that has been found and the program is terminated.
- `selectVar`: This method is invoked when the static, **ascending variable ordering strategy** is followed. It simply returns the first variable that appears in `varList`.
- `selectVarDynamic`: This method is invoked when the dynamic **smallest-domain first** heuristic is followed. It iterates `varList` and it returns the variable with the smallest domain.
- `selectVal`: This method receives the variable selected as a parameter and it returns the next supported value in its domain. The order of the values is kept sorted, ensuring that an ascending value ordering is kept.
- `assign`: This method is invoked whenever a value is assigned to a variable by the solver. It updates the variable's domain to reflect the assignment (so that is contains only the newly assigned value) and it also removes it from `varList`.
- `deleteValue`: This method is used whenever we need to delete an unsupported value from a variable's domain. It does so by simply accessing the current domain for that variable and deleting the value from it.
- `satisfies`: This is a utility function used by the `revise` method, described below. It takes two variables with two values (one for each variable) and it checks whether this potential assignment is satisfied by the CSP instance. It does so by first iterating all the constraints and finding the binary constraint between these two variables. Once the constraint is found, its tuples, which hold the valid assignments, are iterated and if a matching tuple is found, the method returns true. Otherwise it will return false. Since constraints in our `BinaryCSP` instance are expressed as

`c(i, j)` where $i < j$, special care is taken for ensuring that if we call this method for `arc(j, i)`, we are still identifying the corresponding constraint and their valid value assignments.

- `revise`: This method is responsible for revising an arc, for example, `arc`(x_i, x_j). Essentially, this method prunes the unsupported values in the x_i variable. It does so by iterating all of the available domain values of the x_i variable and it checks whether there exists a value in the domain of x_j that satisfies the constraint. If a value is unsupported, it will be removed (pruned) from the domain of x_i. Furthermore, if this procedure causes the domain of x_i to be empty, this method will throw a `ReviseException`, which allows the solver to detect a domain wipeout.

Note: Given the way we have used nodes to encapsulate the state of the search, there is no need to implement an `unassign` or a `restoreValue` procedure. This is due to the fact that by reverting to the previous node when an arc inconsistency is detected, we are implicitly unassigning or restoring a value from that variable as well as undoing any pruning as a result of arc revision.

3 Binary Branching

Before delving into the implementation details of the two algorithms, I will be discussing binary branching and its implementation in our solver. In binary branching, if we find that a partial assignment, for example assigning $x = v$ (assign value v to variable x) leads to an arc inconsistency, we can remove value v from the domain of variable x which essentially splits our search space into two. This has several advantages over a d-way branching approach, as dead ends are detected sooner, making the algorithm faster as it saves it from a lot of unnecessary work.

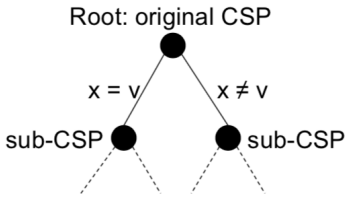

Figure 1: Graphical representation of binary branching. Taken from[1]

In this project binary branching is carried out through the `assign` and `deleteValue` operations of instanced of the Node class. A partial assignment

of a value to a variable with the `assign` operation, corresponds to branching left with $x = v$. If the algorithm detects that this partial assignment leads to a dead end, we can use the `deleteValue` operation to remove the value v from x (corresponding to $x \neq v$) and thus it is removed from further consideration.

4 Forward Checking

The basic idea of this algorithm is that after an assignment to a variable x_i, we revise arcs from every future variable, which is a variable that has yet to be assigned, to x_i once. This process enforces local arc consistency between these arcs[2]. The implementation of this algorithm can be found in the `ForwardChecking.java` file which defines the `ForwardChecking` class, a class with a number of attributes including a 'utility' attribute acting as a flag, indicating which variable ordering technique is to be used, the `currentNode` attribute which is used to hold the node currently under consideration and the `csp` attribute which holds the instance of our binary CSP.

The first `currentNode` will be the node representing the original CSP, before any assignments or deletions have been made, and it serves as our starting base. To create this node, the following operations are used:

- `initVarList`: This creates a list containing the variables of the CSP, that will be used as the `varList`, which is the list holding the unassigned variables, of the root node. As a result this will initially hold all the variables of the CSP.
- `initVariablesMap`: This method initialises the `variablesMap` attribute of the root node, which as mentioned earlier, maps each variable to a list of its domain values. Initially, the domain of each variable will be equal to the variable's domain as described in the input file.

Once the root node is created, we are ready to proceed with the actual forward checking algorithm which is carried out by the composed of the following operations:

- `forwardChecking`: This is the main recursive procedure. It receives an instance of node as a parameter which, initially, will be the root node. The first thing the method does is to check whether there has been a complete assignment, meaning that a solution has been found. If so, it will print the solution and exit and if not it will proceed to select a variable and a value and branch left.
- `branchFCLeft`: Branching can be thought of as the process of assigning the value to the variable chosen and carrying out the necessary arc re-

visions. This method receives an instance of the `Node` class as well as the variable and the value to be assigned as parameters. The state of the node parameter is copied and from it, a new node is created. Following that, the new node assigns the variable and the arcs from the future variables to that variable are revised. This is carried out by the `reviseFutureVars` method of the `Node` class, which simply calls `revise` for every `arc(futureVar, var)`. If local arc consistency is achieved, the node with the newly assigned variable and the revised arcs will be set as the current node and `forwardChecking` will be invoked with it as a parameter, as it represents the sub-CSP problem resulting from the assignment. If not, we will undo pruning and implicitly unassign the variable by setting the current node to the node instance passed as a parameter to the method.

- `branchFCRight`: This method is called whenever an arc inconsistency is detected in `branchFCLeft`, and it carries out the process of deleting the selected value from the variable's domain. It is similar to `branchFCLeft` in that it creates a copy of the node passed as a parameter (the current node) with the difference that instead of assigning a value to a variable, it deletes this value from the variable's domain. After doing so, it calls `reviseFutureArcs` and if local consistency is achieved the newly created node is set as the current node and `forwardChecking` is called with it as a parameter. Otherwise, the pruning is undone by setting the current node as the node that was passed as a parameter to this method. This also implicitly restores the value we deleted.

Note: This implementation is very similar to the one outlined in the lecture slides in pseudocode[3], as I closely followed it and used it as an archetype.

5 Maintaining Arc Consistency

The basic idea of this algorithm is that we start by making the problem globally arc consistent and we re-establish global consistency after every single assignment. The main benefit of this algorithm over forward checking is that it spots domain wipeouts earlier and it is thus more efficient.

At the core of this algorithm has to lie a procedure that enforces arc consistency, that is, a procedure which following an arc revision, it ensures that the rest of the arcs are consistent. AC1 is the simplest method to do so and it achieves arc consistency by iterating and revising all of the arcs until there are no changes. However this is inefficient as it will revises arcs even if it is not necessary to do so. AC3 is a more efficient method to enforce arc consistency as it makes use of the fact that support is bi-directional, meaning that if a value is supported on `arc(`x_1`,` x_2`)`, then is it supporting some value on `arc(`x_2`,` x_1`)`,

and likewise, if a value is not supported on `arc(`x_1`, `x_2`)` then it is supporting no values on `arc(`x_2`, `x_1`)`. For that reason, it would be unecessary to revise `arc(`x_2`, `x_1`)` after revising `arc(`x_1`, `x_2`)`[5]. Because of this advantage, we have implemented AC3 in this practical. The operations that carry out this procedure can be found in the `Node.java` file and they are as follows:

- `initialiseArcQueue`: This method initialises the queue utilised by AC3, which contains the arcs and the order in which they will be revised. This queue will initially contain all the arcs incident on the variable assigned. Note that this only includes arcs between the newly variable assigned and other unassigned variables. For example, take the 4Queens problem and imagine we have four variables, v_0, v_1, v_2, v_3. The initial AC3 queue when we assign a value to v_0 will contain the arcs `arc(`v_1`, `v_0`)`, `arc(`v_2`, `v_0`)` and `arc(`v_3`, `v_0`)`. When a value is assigned to v_1, however, initial queue will contain the arcs `arc(`v_2`, `v_1`)` and `arc(`v_3`, `v_1`)` since v_0 has already been assigned a value.
- `getSubsequentArc`: This method adds arcs to the AC3 queue that could possibly become inconsistent after an arc revision is performed. For example, following from the example above, when `arc(`v_1`, `v_0`)` is revised, the arcs `arc(`v_2`, `v_1`)` and `arc(`v_3`, `v_1`)` are added to the queue. Special care is taken so that arcs that are already present in the queue are not added once again.
- `ac3`: This is the method that carries out the AC3 procedure. After the queue is initialised it enters a loop in which the front-most arc in the queue is removed and it is revised, by invoking the `revise` method described earlier. If `revise` returns true, it will add the appropriate arcs to the queue. The loop will terminate either when the queue is empty, meaning that the CSP is arc consistent, or when a `ReviseException` is thrown due to a domain wipeout being detected.

Note: In order to represent arcs, we have created an `ArcPair` class, which can be found in the `ArcPair.java` file. It is a simple class with two attributes, one for each var present in the arc.

Now that we have explained how AC3 is implemented, we will go over the implementation of the maintanining arc consistency algorithm (MAC), which embeds AC3 (and can be thus referred to as MAC3). Its implementation can be found in `MaintaintArcConsistency.java`. Some aspects of its implementation are very similar to our implementation of the forward checking algorithm as they have the same attributes and they both make use of the node data structure to encapsulate different parts of the search. Upon its instantiation, the root node is initialised by using the `initVarList` and `initVariablesMap` methods, and then we enter the main, recursive procedure described below:

- `mac3`: The method will first create a new instance by copying the contents of the node that was passed as a parameter. Following that, a variable and a value will be selected for its assignment and it checks whether complete assignment has been reached, that is whether a solution has been found. In the case that it has, it will print the solution. Otherwise it will call AC3, and if global arc consistency hold, the method will recursively call itself with the newly created node as its, which will also be set as the current node. If an arc inconsistency is detected, the pruning will be undone by assigning the current node to the node passed as a parameter to the method (essentially the parent node). After that, yet another node will be created by copying the contents of the parent node, however in this case we will delete the value selected from the variable's domain. Following that, we'll check whether global consistency holds by calling AC3 and if it holds, we will assign this newly created node as the current node, and we will recursively call the method with this as the parameter. If an arc inconsistency is indeed detected, we will undo unpruning by assigning the current node attribute to the parent node.

Note: Our implementation follows closely the pseudo-code as given in the lecture notes[6].

6 Empirical Evaluation

Along with the specification, we were also given CSP instances as well as instance generators to create new ones. In order to empirically test the different combinations of algorithms and variable ordering strategies, I have designed a set of experiments which run each combination with all of the instances and the textual output, which contains the solution, the time to find it, the nodes in the search tree and number of arc revisions performed, is in files that can be found in the `results` directory.

The CSP instances are divided into three problem categories namely, the nQueens problem[7], Langford's number problem[8] and Sudoku puzzles[9]. We will be discussing the results of the experiments for each problem category, going over each of the instances solved.

Note: I have supplied a script, `empirical.sh` which automatically sets up and carries out the aforementioned experiments.

6.1 nQueens

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	1	13	18
FC	Dynamic	1	13	18
MAC3	Static	2	12	24
MAC3	Dynamic	2	12	24

Table 1: Results for the 4Queens problem.

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	6	47	96
FC	Dynamic	6	47	96
MAC3	Static	10	25	134
MAC3	Dynamic	8	25	134

Table 2: Results for the 6Queens problem.

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	17	153	366
FC	Dynamic	20	135	317
MAC3	Static	26	65	533
MAC3	Dynamic	25	68	525

Table 3: Results for the 8Queens problem.

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	19	151	415
FC	Dynamic	10	55	164
MAC3	Static	41	60	578
MAC3	Dynamic	38	36	358

Table 4: Results for the 10Queens problem.

In addition to the supplied instances, I have created additional instances of the nQueens problems, for which the results are reported below:

Algorithm	**Variable Ordering**	**Time(ms)**	**Nodes**	**Arc Revisions**
FC	Static	104	2645	8735
FC	Dynamic	41	229	820
MAC3	Static	199	839	11414
MAC3	Dynamic	89	101	1602

Table 5: Results for the 14Queens problem.

Algorithm	**Variable Ordering**	**Time(ms)**	**Nodes**	**Arc Revisions**
FC	Static	6219	263869	1082961
FC	Dynamic	46	265	1099
MAC3	Static	13704	73499	1354392
MAC3	Dynamic	256	137	2995

Table 6: Results for the 20Queens problem.

6.2 Langford's Number Problem

Algorithm	**Variable Ordering**	**Time(ms)**	**Nodes**	**Arc Revisions**
FC	Static	4	19	41
FC	Dynamic	4	19	41
MAC3	Static	7	16	85
MAC3	Dynamic	6	16	85

Table 7: Results for Langford's L(2,3) problem.

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	11	57	142
FC	Dynamic	14	57	141
MAC3	Static	17	29	225
MAC3	Dynamic	16	29	212

Table 8: Results for Langford's L(2,4) problem.

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	4424	7046	41709
FC	Dynamic	1381	3296	19981
MAC3	Static	702	181	11180
MAC3	Dynamic	647	58	4126

Table 9: Results for Langford's L(3,9) problem.

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	31728	30205	197318
FC	Dynamic	8421	13323	87804
MAC3	Static	2079	544	40180
MAC3	Dynamic	1650	232	16206

Table 10: Results for Langford's (3,10) problem.

6.3 Sudoku

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	319	296	8783
FC	Dynamic	104	82	3240
MAC3	Static	343	451	9404
MAC3	Dynamic	342	163	5647

Table 11: Results for the Simonis Sudoku problem.

Algorithm	Variable Ordering	Time(ms)	Nodes	Arc Revisions
FC	Static	101636	218712	5395849
FC	Dynamic	4703	20164	385830
MAC3	Static	96409	253990	5070164
MAC3	Dynamic	3818	11152	304831

Table 12: Results for the Finnish Sudoku problem.

6.4 Analysis

By looking at the results for CSP instances that describe smaller problems, such as all the nQueens instances, apart from 14Queen and 20Queen, and Langford's L(2,3) as well as Langford's L(2,4) in tables tables 1, 2, 3, 4, 7 and 8 respectively, it is difficult to identify any patterns emerging, and thus we can't make definite conclusions with regards to the different configurations of the solver. The similarity in results is due to the small number of variables and the small size of the search space. As a result, the differences brought by the use of different algorithms and different variable ordering strategies are not evident. As a result, in order to draw any useful conclusions we will be looking at the results for the other CSP instances.

The first conclusion that we can draw is that the dynamic variable ordering strategy is superior that the static one as in every single case it reduces the time required to find a solution, reduces the number of nodes used in the search tree and it also reduces the number of arc revisions performed, all by a significant amount. This strategy was immensely successful, regardless of the algorithm used. This is an expected result as the smallest-domain first heuristic chooses the variable that is most difficult to find a value for, allowing the solver to 'fail-fast' and move on to other partial assignments that are likely to succeed[10].

In order to see the differences between the two algorithms, we will discuss the results over the individual problem categories. Note that unless we make a distinction, we will be comparing FC with MAC3 when they use same variable ordering strategy. When we look at the nQueens problem, specifically 14Queens and 20Queens, the number of nodes in the search tree of the MAC3 algorithm are significantly fewer when compare to the nodes in the search tree used by FC. On the other hand, the number of arc revisions done by MAC3 are more which may also be the reason why it takes a longer time to find a solution. When it comes to the Langford's number problem, specifically L(3,9) and L(3,10),

we see once again that the number of nodes used by MAC3 are significantly fewer than the nodes used by FC. The number of arc revisions carried out by MAC3 is also smaller for the FC counterpart. A noteworthy observation is that for all Langford's number problem instances, independently of which variable ordering strategy is used by FC, MAC3 finds a solution in a shorter amount of time, using fewer nodes and requiring fewer arc revisions. This goes against the pattern that appears in the other two problem categories where the FC algorithms employing a dynamic variable ordering strategy is superior to the MAC3 algorithm employing a static variable ordering strategy. Lastly, we will look into the results for the two Sudoku problems. The results for the Simonis Sudoku instance in table 11, we can see that, contrary to the previous observations, the number of nodes needed by MAC3 are more than the number of nodes needed by FC. At the same time, MAC3 requires a higher number of arc revisions and also a longer amount of time to find a solution. We can thus conclude that for this problem, the FC algorithm yields better results. In the case of the Finnish Sudoku problem, as listed in 12, when a static variable ordering is employed MAC3 requires a larger number of nodes but fewer arc revisions when compared to FC. On the other hand, when a dynamic variable ordering strategy is employed, the MAC3 algorithm requires fewer nodes, fewer arc revisions and also requires a shorter amount of time to find a solution. Overall we can see that MAC3 usually requires a fewer nodes to find a solution at the cost of a higher number of arc revisions. This makes sense as for each change in a variable's domain we check for global consistency, which means revising not only the arcs between future variables incident on the variable changed, but also the arcs that could become inconsistent as a result of these subsequent revisions. The smaller number of nodes required by MAC3 also makes sense as it spots dead ends quicker than FC which only checks for local arc consistency.

In order to study the relationship between the time taken to find a solution with the number of nodes required and the number of arc revisions carried out, I've calculated the following correlation matrix:

	Time	**Nodes**	**Arc Revisions**
Time	1		
Nodes	0.79	1	
Arc Revisions	0.97	0.86	1

Table 13: Correlation matrix between the three metrics.

As we can see, we can see that there exists a very high correlation between the number of arc revisions carried out and the time required to find a solution.

We can thus determine that the number of arc revisions required is the best predictor as to how long the solver will find a solution and if we were to optimise the solver, we could achieve a significant performance boost if we make the revision procedure faster.

7 Conclusion

Overall, I'm very pleased with the result of this practical as I've managed to implement all the requirements set out in the specification. I have successfully implemented a solver, that supports two binary constraint problem algorithms and it also supports two variable selection heuristics, namely a static ascending variable strategy, and a dynamic smallest-domain first variable ordering strategy. Furthermore, I have successfully set up and conducted a number of experiments as part of an empirical evaluation process, which revealed certain traits of the solver while using different configurations, with regards to the time taken to find a solution, the number of nodes used in search and the number of arc revisions performed. If I had more time, I would first extend the solver to support more heuristics, such as random variable and value ordering heuristics and perhaps extend the solver to support constraints such as equality constraints. The practical was very successful in achieving its learning goals as I have gained experience in implementing a CSP solver that utilises two popular and widely used algorithms, with a number of heuristics.

References

[1] (2020) Constraint Programming CS4402, Week 6, Lecture 1: Basic Solution Procedures

[2] (2020) Constraint Programming CS4402, Week 6, Lecture 2: Combining Search and Propagation, Enforcing Arc Consistency.

[3] (2020) Constraint Programming CS4402, Week 7, Lecture 1: 2-Way Branching, Heuristics I, MAC: Maintaining Arc Consistency, Page 12.

[4] (2020) Constraint Programming CS4402, Week 7, Lecture 1: 2-Way Branching, Heuristics I, MAC: Maintaining Arc Consistency, Page 50.

[5] (2020) Constraint Programming CS4402, Week 6, Lecture 2: Combining Search and Propagation, Enforcing Arc Consistency, Page 65

[6] (2020) Constraint Programming CS4402, Week 7, Lecture 1: 2-Way Branching, Heuristics I, MAC: Maintaining Arc Consistency, Page 56.

[7] 054: N-Queens. (2020). Retrieved 3 May 2020, from http://www.csplib.org/Problems/prob054/

[8] 024: Langford's number problem. (2020). Retrieved 3 May 2020, from http://www.csplib.org/Problems/prob024/

[9] Sudoku. (2020). Retrieved 3 May 2020, from https://en.wikipedia.org/wiki/Sudoku

[10] Constraint Guide - Value&Variable Ordering. (2020). Retrieved 3 May 2020, from https://ktiml.mff.cuni.cz/ bartak/constraints/ordering.html