CS4402

Practical 2 - Constraint Solver Implementation

190004947

November 30, 2021

University of
St Andrews

Contents

1 Introduction

For this practical I have implemented two different constraint solvers. The algorithms I have implemented are Forward Checking and Maintaining Arc Consistency. The latter of which enforces arc consistency at each level of recursion to discover dead ends in the search tree early to avoid thrashing through assignments unnecessarily.

Unfortunately, after many hours of tinkering, I was not able to build out fully functioning solvers. While Forward Checking does not solve any of the provided problem instances, my implementation of Maintaining Arc Consistency returns solutions for 4Queens, 8Queens, and SimonisSudoku. I am genuinely perplexed at why it is able to solve 4Queens and 8Queens while failing at 6Queens. I think I have mismanaged book-keeping of pruning events and thus the UndoPruning procedure does not act as intended resulting in invalid search trees. In my attempt to fix my implementation, I actually built two versions of it, writing the second from scratch. I encountered similar problems which I assumed was coming from one (or more) of my utility functions which is the only part of the program that remained consistent between the two versions. I tried troubleshooting until past the deadline as I was convinced that I would find somewhere a minor issue that was causing an incorrectly branching search or invalidating variable domains.

2 Design Decisions

For both Forward Checking and Maintaining Arc Consistency, I began by creating a BinaryC-SPReader to read in the constraint problem which is then used to parse the problem values. I have set up four initialisation functions to initialise the constraints, domains, assignments, and varList which stores all of the unassigned variables to be passed through the various solving methods. Each of these are stored as ArrayLists consisting of different objects. The constraints are stored as BinaryConstraints which is a provided Java class. To store the domains and assignments I built out the DomainMapping and AssignmentMapping classes which both store an integer to represent the variable being mapped and then the domain is stored an ArrayList of integers whlie the assignment is stored simply as another integer. Both of these classes contain multiple utility functions to get values and manipulate them as necessary.

Again for both implementations, I have had to keep track of pruned values in order to correctly undo them during backtracking using the UndoPruning procedure. This is achieved using three global variables, latestPrune, prunesFromAC, and prunesToUndo. The first of these is a DomainMapping which stores the variable whose domain has been pruned and a list of the pruned values. It is updated every time the Revise method makes an edit to a domain. In FC when ReviseFutureArcs is called or in MAC when AC3 is called, after domain values are pruned this DomainMapping is added to an ArrayList stored as prunesFromAC; this variable name represents the fact that, in MAC, it is storing the prunes from running the enforce arc consistency function AC3. Note that in FC this variable stores all the prunes that were made during ReviseFutureArcs. At the end of the AC3/ReviseFutureArcs function, the prunes that were made during that method call are added to the prunesToUndo list. This variable is supposed to behave like a stack and was initially implemented as a stack but during my troubleshooting I tried an ArrayList implementation which also didn't work as planned and so that is how it stands in my submission. The idea is that it contains a list of lists, each inner list being the set of values that were pruned. When UndoPruning is called, the most recently added set of prunes are to be returned to the domains they were taken from. This is in line with the pseudocode provided in the lectures which defines UndoPruning as undoing the pruning completed in either AC3 or ReviseFutureArcs. This returns the state of the domains to how they were before the search branching through which we have now backtracked.

3 Forward Checking

My implementation of Forward Checking follows exactly the pseudocode provided in the Week 9 lectures. It starts with the target case of a complete assignment which, if met, will print out all of the assignments and return. Initially, instead of returning, I ran System.exit() to close the Java Virtual Machine session and jump right out of the program. This is because when a solution is found multiple recursion levels deep, the solution is printed out every time the program comes out of a recursive call. After this target case, a SelectVar method is called to choose the next variable to be assigned in the search. There are two SelectVar methods for each of the variable ordering strategies; smallest-domain first and ascending order. While ascending order simply chooses the variables in the order they were initialised, smallest-domain first chooses the variable with the smallest domain and hence the least number of possible prunes.

Once a variable to be assigned has been chosen and a value from its domain has been chose for the first assignment in the search, both BranchFCLeft and BranchFCRight are called to search the two sides of the binary search tree. BranchFCLeft assigns the variable selected to the value selected before calling ReviseFutureArcs which will search through the incident arcs of all future (unassigned) variables pruning values from their domains as fit. If this function returns true, this means that no domains were wiped, there are still possible solutions down that branch, and we make a recursive call to ForwardChecking with the new varList that doesnt contain the assigned variable. In the case that ReviseFutureArcs wipes a domain, this means that there are no assignments for that variable that will satisfy all the constraints and we should no longer search down the current branch. The recursive call would then be skipped and UndoPruning is called to take all variable domains to the state they were in before the ReviseFutureArcs call. Once the domain pruning has been undone, the selected variable is unassigned and it is added back to the varList.

When the left branch finishes and the variable from its root is unassigned, we then move into the right branch with BranchFCRight. This method will prune the selected value from the domain of the selected variable to check if it results in a domain wipe. If there is no domain wipe, we might need to continue down the branch and ReviseFutureArcs is called to determine whether or not possible solutions still exist down that branch. If so, like in the left branch, we make a recursive call to Forward Checking with the updated varList. When ReviseFutureArcs reports that there are no possible solutions in the current branch, the latest pruning is undone and the value taken earlier is added back to the domain it came from.

In theory. this is a complete search algorithm and it should always find a solution if one exists. However, I could not get my implementation of Forward Checking functional and it doesn't return solutions for any of the provided problem instances.

4 Maintaining Arc Consistency

In Maintaining Arc Consistency, a similar logic is followed except we use the AC3 algorithm to propagate arc consistency throughout the search which helps us find dead ends earlier by making inferred domain prunings. As in the FC algorithm we have two methods for selecting the next variable to be assigned which is set as a command line argument; 0 represents smallest-domain first and 1 represents ascending order. Once a variable and value from its domain has been selected, we have our target case again in which all variable have been assigned. In this case, we print out all the assignments using the PrintAssignments method and return from the MAC3 function. If a complete assignment has not yet been found we run AC3 to check all the constraints against the value that has just been assigned, propagating the results as domains are changed. When we assign a variable, we assume that it holds the only value in its domain. In the case that AC3 reports no inconsistencies (domain wipeouts) we make a recursive call to MAC3 which will choose the next assignment and follow this process again until either a complete assignment (solution) is found or a domain is wiped out in AC3 telling the solver that no solution exists down the current branch. In the case that AC3 reports a domain wipeout, we move past the recursive MAC3 call and undo the pruning that was done during the last AC3 call using the stack structure discussed earlier.

Once the domains have returned to their previous state, we unassign the previously assigned variable as we have discovered that it cannot lead to a solution. We then prune the checked value from the domain of the variable to check if it leads to a domain wipeout which would require the search to jump back up a level. If the domain was not left empty, we check arc consistency again and propagate the change in domain. If all domains are non-empty after this call to AC3, we make a recursive call to MAC3 with the varList containing the last checked variable again. In the implementation, the variable is never explicitly taken out and put back into the varList, instead, I have embedded this process in the Assign and Unassign methods. If the last AC3 call reported that no solutions exist down the current branch, we again call UndoPruning to return the domains to the state they were in before we entered an impossible branch. After this is complete (and if the value pruning lead to a domain wipeout), we add the value back to the domain it was taken from and exit the current level of recursion to try a new variable assignment. This process will repeat until the target case is met and the completed assignments are printed out.

5 Empirical Evaluation

Since my implementation did not work as intended, I am unable to compare and contrast the performance of Forward Checking vs Maintaining Arc Consistency. It would have also been really interesting to see how the variable ordering strategies impacted the solving of different problem types and sizes.

Before I realised that I wasnt going to be able to see problem solutions, I set up a class called SolutionMeasures of which an instance is returned after the FC and MAC algorithms are run. The SolutionMeasures objects store the solver type used to create it (FC or MAC), the problem that was solved, the ordering strategy that was used, the final solution to the problem, the time taken to find the solution, the number of search tree nodes used, and finally the total number of arc revisions made during the search. This object stores all the details about a solve which I intended on using to collect data and build graphs in Python to display how each of these measures were affected by the solver type and the ordering strategy for each different instance type and size.

I have also written a Compare.java class which runs every provided problem instance on both solvers with both variable selecting heuristics. The main method of this class runs all of these solves and collects all of the returned SolutionMeasures objects which could then be used to reason about the impact of each of these variables. It also prints out the results using the SolutionMeasures.toString method which produces an easy to read string. Interestingly, my MAC implementation did work for three of the problem instances: 4Queens, 8Queens, and SimonisSudoku. The solutions and sentences produced by the aforementioned toString method are:

- 4Queens:
  - {1, 3, 0, 2}
  - Using MAC with the ascending ordering strategy, the ./instances/4Queens.csp problem was solved in 135 milliseconds using 8 search tree nodes and making 18 arc revisions.

- 8Queens:
  - {2, 6, 1, 7, 5, 3, 0, 4}
  - Using MAC with the ascending ordering strategy, the ./instances/8Queens.csp problem was solved in 146 milliseconds using 28 search tree nodes and making 164 arc revisions.
- SimonisSudoku:
  - {7, 2, 6, 4, 9, 3, 8, 1, 5, 3, 1, 5, 7, 2, 8, 9, 4, 6, 4, 8, 9, 6, 5, 1, 2, 3, 7, 8, 5, 2, 1, 4, 7, 6, 9, 3, 6, 7, 3, 9, 8, 5, 1, 2, 4, 9, 4, 1, 3, 6, 2, 7, 5, 8, 1, 9, 4, 8, 3, 6, 5, 7, 2, 5, 6, 7, 2, 1, 4, 3, 8, 9, 2, 3, 8, 5, 7, 9, 4, 6, 1}
  - Using MAC with the ascending ordering strategy, the ./instances/SimonisSudoku.csp problem was solved in 1196 milliseconds using 161 search tree nodes and making 786 arc revisions.

6 Conclusion

To conclude, I hope I have demonstrated how my implementation was supposed to run as I was disappointed by its lack of functionality. I wish I had more time to work on it as it would have been incredibly exciting to see the correct assignments pop up for each of the problem instances.

One this I would change if I did this assignment again is the way I stored domains and assignments. I would use a hash table to map variables to domains and assignments. This would've made acquiring specific domains and assignments much easier as in my current implementation I have to iterate through them all until I find the one with the variable I am searching for. A hash table would both make my job easier and also increase the speed of the solver.