Nicholas Allaire, A10639753
Son Tang, A11370127
Marcel Aguiar, A10904034

Programming Assignment #3 Write up

1. Description of the problem and algorithms used in the problems

Problem 1: We loop through all the variables in the given csp, and check if that variable *is_assigned()*. If we come across an unassigned variable, we return false. Otherwise, true. No algorithms were used.

Problem 2: We loop through all the constraints associated with the given variable. Then we check each variable to see that it is assigned and the constraint is satisfied. If not, we return false. No algorithms were used.

Problem 3: We implemented a basic backtracking algorithm to solve different sized sudoku boards. It is basic because variable ordering, value ordering and the inference heuristics are not implemented at this point. The backtracking algorithm starts off by checking to see if there is a solution by calling the method implemented in problem 1, if there is a solution it returns true and the algorithm is done. If no solution is found, we select the next unassigned variable and loop through all the values in its domain. At the start of the loop, we log the current state of assignments so later in the algorithm we can revert back to a state if we don't find a solution. Next we check to see if the variable is consistent and if so, assign the domain value to the variable and recursively call the backtrack method. If the variable is not consistent, we rollback to the saved state we logged earlier in the method. If no solution is ever found, we return false.

Problem 4: In problem 4, we implemented the AC-3 algorithm. Our solution also allows for running the Maintaining Arc Consistency algorithm, depending on if the variable *arc* that is passed in is not *None*. The algorithm loops through each arc in the queue and checks for consistency in the *revise()* method. If not consistent, it removes *x* from the domain. If removed, it then creates tuples for each constraint for that variable and adds them to the arc. If the revise check succeeds, then *ac3* returns true.

Problem 5: In problem 5 we implemented the variable and value ordering heuristics, previously mentioned in problem 3. The variable heuristic utilizes the minimum value remaining (MVR) heuristic and uses the degree heuristic to break ties. The MVR loops through all of the unassigned variables and determines the variable with the smallest domain. If there is a tie between two variables and their domain size, we chose the variable with the largest number of constraints on other unassigned variables. If there is also a tie here, we chose randomly between these variables. The value ordering heuristic utilizes the least constraining value (LCV) heuristic. The LCV returns an

ordered list of the domains values of the variable passed in. The ordering is determined by the number of neighboring variables in the constraint graph for that variable. When finished, the ordering needs to be sorted, so a simple sort is used.

Problem 6: In problem 6, we use the implementations done in problems 4(AC3) and 5(MVR + degree and LCV) and add those to problem 3(backtracking) to make backtracking more efficient. Other conditions were added to each of the algorithms to check for certain cases to make the program run more efficiently. Our optimized backtracking ended up running at 24.931 seconds at it's fastest and averaged around 25.326 seconds.

2. Graphs

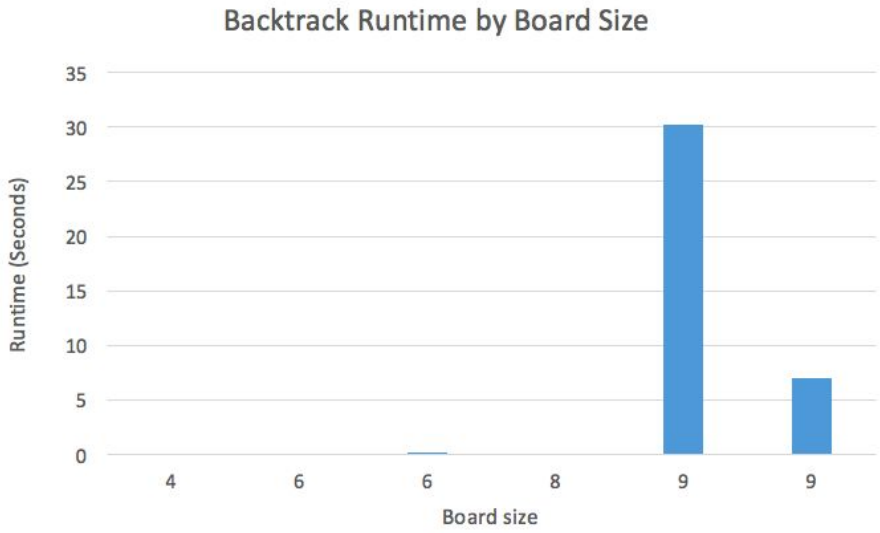

Figure 1. Backtrack Runtime by Board Size. The runtimes of the backtracking algorithm modified by heuristics on various board sizes are shown. The board size indicate an n x n board where n is the board size shown.

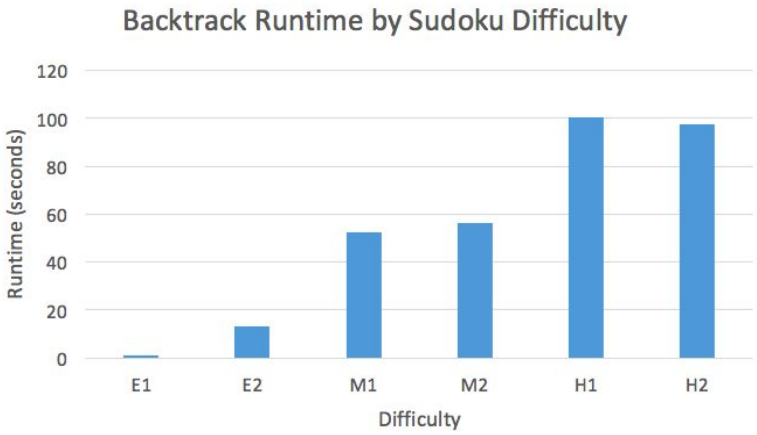

Figure 2. Backtrack Runtime by Difficulty. The runtimes of the backtracking algorithm modified by heuristics on various difficulties are shown. The difficulty is determined by how many numbers are missing from the board. E1 and E2 indicate easy puzzles, M1 and M2 indicate medium puzzles, and H1 and H2 indicate hard puzzles.

The solution time both increases when the size of the board increases and when the difficulty increases. The solution time increases greatly when the board size reaches 9x9. However, the run time still largely depends on the difficulty of the Sudoku problem. This can be seen in the first graph as the runtime of an 8x8 problem is lower than that of the 6x6 problem. The second graph shows how the runtime increases depending on the difficulty of the problem (E1 and E2 are easy problems, M1 and M2 are medium problems, and H1 and H2 are hard problems).

3. Analyze the effect of each heuristic

Runtime with basic backtracking (in secs)	4.67757892609
+ inference	1.08236598969
+ variable ordering	0.978546857834
+ value ordering	0.855565071106

Adding inference reduced the run time of the basic backtracking algorithm the most, nearly cutting the example runtime by ~75%. The inference makes the most impact on runtime because it is the only one that reduces the size of the domain. The other heuristics lowered the runtime by allowing the algorithm to fail more quickly, so that it can find the solution more quickly. However, they do not reduce the size of the domain, or space that the algorithm has to search through, so their effects on the runtime are not as significant as that of the inference.

4. Paragraph from each author stating their contribution and what they learned

Nick: I helped program and debug problem 2, problem 3, worked on the revise function in problem 4, worked on the select_unassigned_variable function, and helped try to optimize problem 6. I learned that sudoku is very easily solvable using these techniques, if optimized. I was a little confused about how the whole backtracking algorithm worked, especially the AC3 part of it. But now after coding it, I have a better understanding of how and why it helps improve the backtracking algorithm. I also learned that adding simple checks to break out of loops early can drastically optimize a CSP, in our case adding an else statement cut down p6 from 11 min to ~50 seconds.

Son: Worked to help implement problems 1, 2 and 3. I also helped to debug problems 4 and 5, and spent a lot of time to optimize problem 6 as much as I could. I helped with the writeup by finding sudoku problems and creating the graphs. I learned first hand how much time it takes for the computer to solve constraint problems with large numbers of possible states. I also learned the importance of optimizing code to cut down on possible states whenever possible as it drastically reduces the time it takes to find a solution for large problems.

Marcel: I helped program problems 1 and 2. At first I had trouble understanding problems 3-5, but was caught up on how we implemented the solution for them. I helped try to optimize problem 6 and brainstormed ways to speed up the *backtrack()* and *revise()* functions. I also contributed to the writeup. The biggest learning points from this assignment were the backtracking and ac3 algorithms. Although ac3 seemed complex when first introduced, the assignment helped understand how it finds the solution to a csp problem, and how MAC relates to it.