Programming Assignment # 3

Description of the problem and the algorithms used in the problems

- Problem 1: Implementing is_complete
  - I iterate through all of the csp variables and have an if statements that returns false if any one of the assignments are not assigned. After the iteration the program simply returns true.
- Problem 2: Implementing is_consistent
  - The list of constraints, a passed in variable must satisfy with its neighbors, are given to us with csp.constraints[variable]. Using this, we check if the passed in value is satisfiable within these constraints using is_satisfied() function for each of constraint's var2 (neighbors) values that have been assigned.
- Problem 3: Implementing backtracking
  - We simply followed the pseudo code that was given to us during lecture. If all the variables have been assigned, then we end the function and return true. Or else we first go through the ordered domain list of unassigned values and see if assigning the the domain in the domain list to the variable given from select_unassigned_variable(csp) does not violate any constraints. If it doesn't we call backtrack on the now altered csp or else we backtrack.
- Problem 4: Arc Consistency (ac3)
  - Again, we simply followed the pseudo code that was given to us during lecture. We have a queue of arcs in csp that consists of tuples (assignment3.Variable, assignment3.Variable). While the queue is not empty, we get pop off the first tuple in the queue and put the two variables in the revise function.
  - The revise function loops through the current variable's domain and checks if the constraints associated that value for the variable will not satisfy any values for one of its neighbor. If it doesn't, then it deletes the value from the variable's domain and signifies the variable's domain as having changed to the ac3 function.
- Problem 5:
  - select_unassigned_variable: For selecting an unassigned variable based on its minimum-remaining-value heuristic, we first obtained a list of variables not assigned, sorted them based on the number of values in their domain and to create a second list that contained the list of variables tied for least number of domain values. In this list, we assigned the number of unassigned constraints that each variable has to itself and sorted this second list based on the number. Then we returned the first variable in this list, as it is the one that is has the most number of constraints of other unassigned variables to which it is associated with.
  - order_domain_values: For a Least-Constraining-Value heuristic, we first looped through each value in the passed in variable's domain list. With each value, we assigned it to that value and looped through another loop through constraints, checking to see how many neighbors, that haven't had a value assigned, would have values be assignable after the variable sets its value. The number of assignable values are added up among the

variable's neighbors and the resulting list of variable's domain is sorted using this number.

- Problem 6:
  - We mostly just copied and pasted our solutions from other files into problem 6 and ran the combined program as a single one. There seemed to be a speed up for the initial 4x4 puzzles but seemed to be slower when dealing with larger puzzles compared to that of p3's solution. This seemed to be the case because our solution had a handful of for loops in various methods that p3 wouldn't have to go through. The implementations of these for loops may have counteracted the positive affect p6's solution should have had in time complexity.

Work Done:

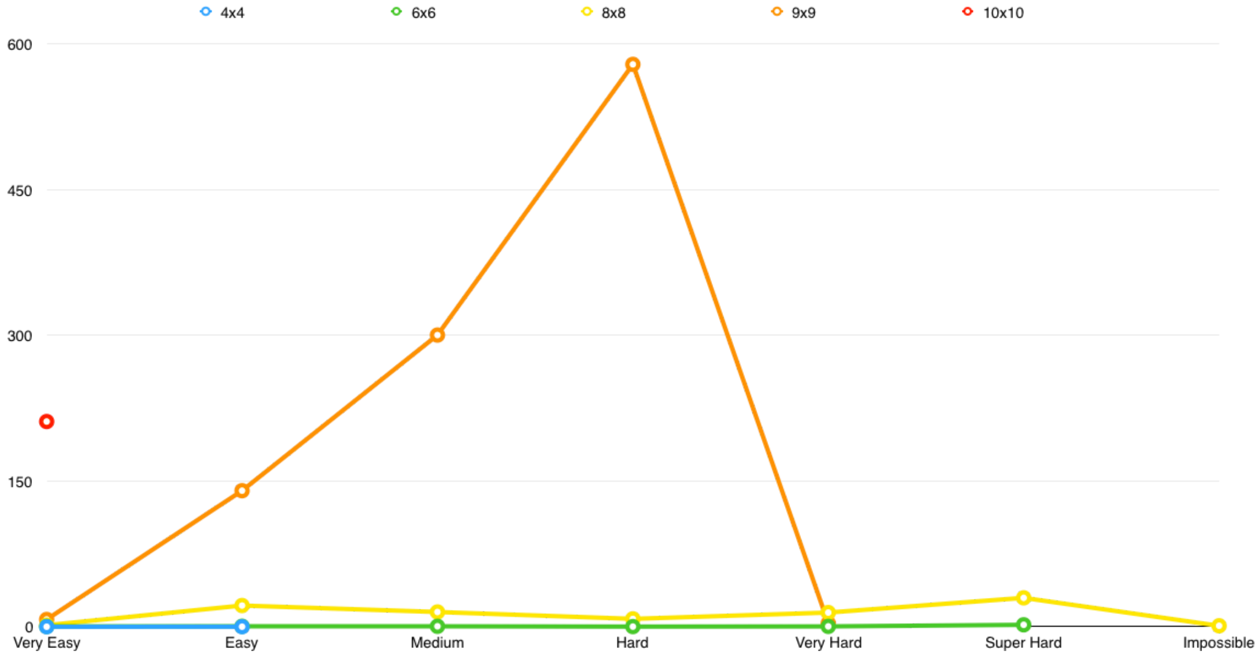
- Woo-Jong
  - Worked on p2, p3, p4's revise function, p5's order_domain_values function, and p6's code time complexity testing.
- Sarah Haroon
  - I worked on p1, p4, p5's order_domain_value, and I worked on improving the overall time efficiency. For the most part this was pair programming/analyzing.

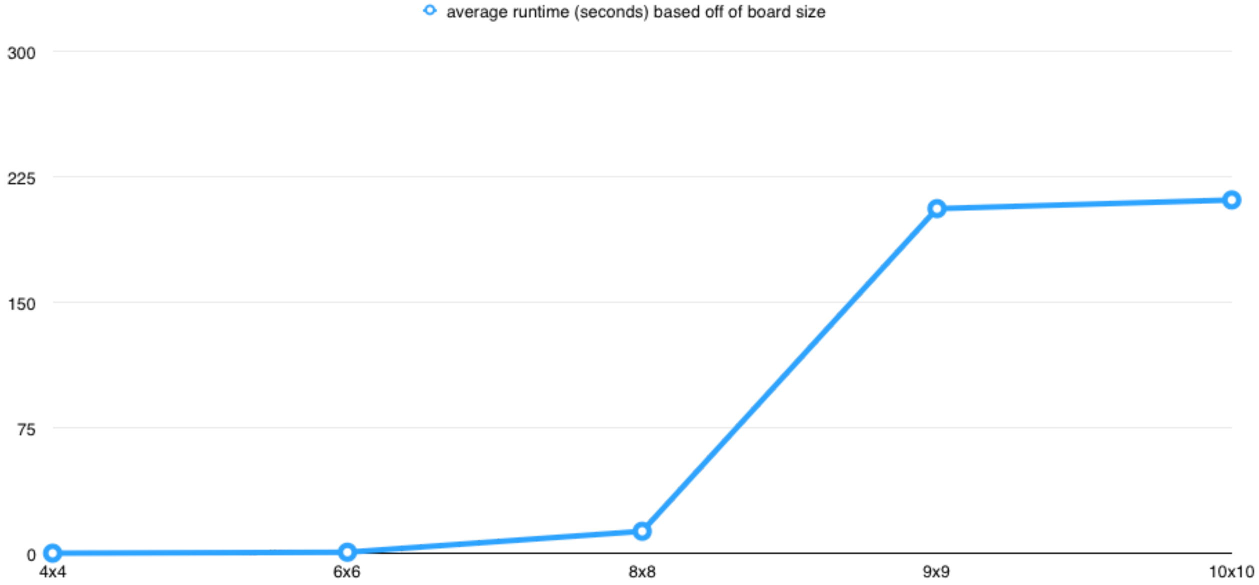
Analyzing Various Puzzle Runtime Data

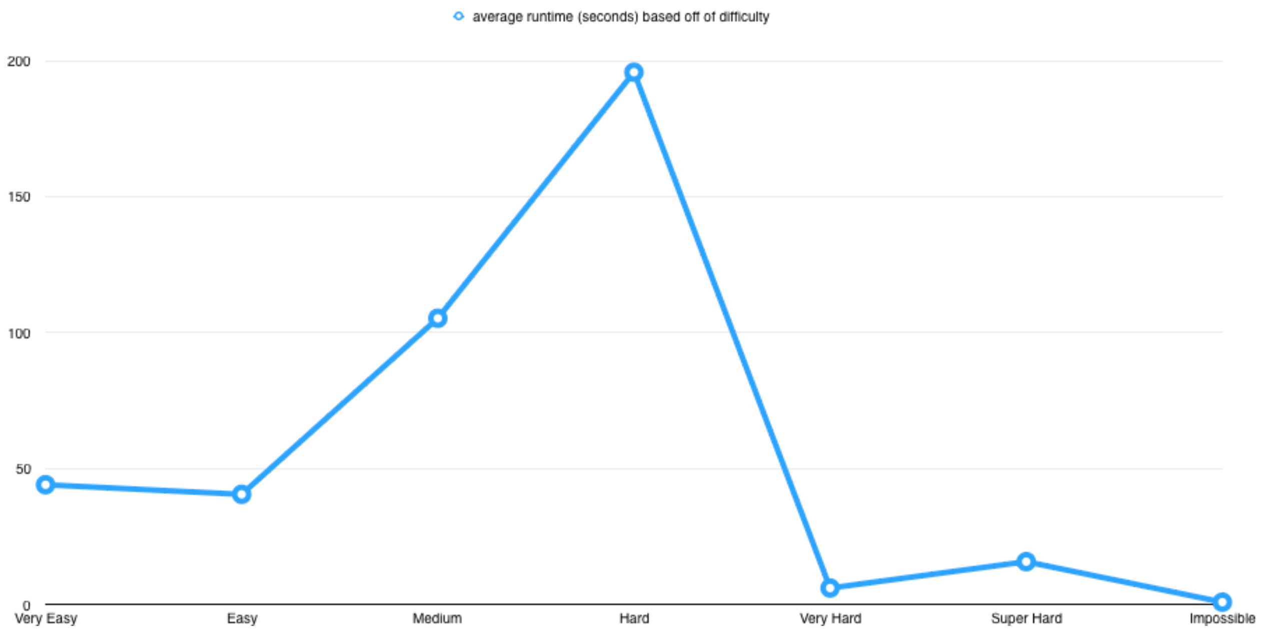
The running time (in seconds) based on difficulty and board size for p6 followed a similar pattern in which the time steadily increased by a small margin as board size and difficulty became higher. At a certain point, one of the puzzles in the higher difficulty and bigger board size created a big jump in the time it took to solve it, resulting in both graph's sudden jump after which it subsided back down to normal incremental levels again. This leads us to believe that there may have been certain instances where a particular set of numerical placements in the 9x9 sudoku puzzles made the program do more calculations than it normally would in that range of difficulty and size.

In general, starting from 9x9's, the time complexity was significantly higher than those of lower difficulty and size.

difficulty	4x4	6x6	8x8	9x9	10x10	average runtime (seconds) based off of difficulty
Very Easy	0.0140268	0.31280112	1.58310103	7.47107315	211.25813	44.12782642
Easy	0.0580329	0.46983289	21.8414058685	140.065894		40.608791414625
Medium		0.45681309	15.1850309372	300.278537989		105.3067940054
Hard		0.15003204	8.10266184	578.949390888		195.734028256
Very Hard		0.35738420	14.5984568	3.55530595		6.17038231666667
Super Hard		1.99984717	29.7338			15.866823585
Impossible			0.990803956985			0.990803956985
average runtime (seconds) based off of board size	0.03602985	0.624451751666667	13.1478943475264	206.0640403954	211.25813	58.4007785649538


4x4
6x6
8x8
9x9
10x10
600
450
300
150
0
Very Easy
Easy
Medium
Hard
Very Hard
Super Hard
Impossible


average runtime (seconds) based off of board size
300
225
150
75
0
4x4
6x6
8x8
9x9
10x10



board size	p3 (average time in seconds)	p6 (average time in seconds)
4x4	0.00484585762024	0.03602985
6x6	0.102180639902767	0.624451751666667
8x8	3.578026533125	13.1478943475264
9x9	3.94802904129	275.3121727976

Part three for some reason, according to this table did way better than part 6; however on certain cases of 8 by 8s, part 6 did significantly better than part 3 could ever do. As the difficulty level increased, the chances of part 3 doing worse than part 6 increases. When part 6 does better than part 3 on certain cases, the time complexity significantly decreases. One can see this in the following case:

For 8x8 impossible difficulty:

P3: 134.073457956 seconds

vs

P6: 0.990803956985 seconds

There were some noticeable difference when leaving certain heuristics but since our outputs seemed to differ various inputs without a very noticeable pattern, it was hard to tell whether the MLV or the LCV heuristic were the main influences in runtime. One of the hypothesis we could make was that the less constraints there were in certain cases with more domains, it would increase the time it takes to solve the puzzle in calculating the next unassigned variable compared to p3's unassigned variable function that just returned any next unassigned variable.