For the first problem, we had to determine whether the sudoku was complete. In order to determine whether the sudoku was complete, we checked if every variable had been assigned. Whether or not the assigned variable was correct didn’t matter, so we just iterated through all the variables in the csp and checked if they were assigned. If any variable was unassigned, we returned false, otherwise, we returned true.

For the second problem, we had to determine whether a variable in the csp was consistent. In order to check whether or not it was consistent, we iterated through all constraints containing the variable to be checked. For each constraint, we checked if the value we wished to assign to the variable was already assigned to one of its constraints. If it was already assigned, we returned false, otherwise, we returned true.

For the third problem, we had to implement a simple backtracking algorithm. For this algorithm, we selected an unassigned variable and chose a value in its domain to assign to it. If this value was consistent, we would assign the value and recursively call backtrack until the csp was fully filled out. If no value in the domain is consistent, then a false is returned and the csp must rollback to a point in which consistent values can be assigned.

For the fourth problem, we had to implement the AC3 algorithm. For this algorithm, we put all arcs into a queue. After, we pop an arc from the queue and check if the domains of the two variables violate any constraints. If they do, we remove the violations from one variables domain and check if the variable still has anything in its domain. If it doesn’t, we return false. Otherwise, we append all of the variables neighbors to the queue and continue. If the queue makes it to the end, we return true.

For the fifth problem, we had to implement the variable and value ordering heuristics. For variable ordering heuristic, we create a list of all unassigned values, and order them from by domain size. We then go through the list from the beginning and compare domains, if there is an absolute smallest domain, we can just return that variable as the smallest. If there are ties in domain size, then we compare the amount of constraints each variable shares with other unassigned variables. For value ordering, make a dictionary to hold the variable domains. We then go through each constraint that the variable is involved in. Every time there is an overlap in domains, we increment the respective value in the dictionary. Afterwards, we sort the domains from lowest to highest based on the dictionaries values.

For the sixth problem, we just put all the code together to form faster Sudoku solver. Using the value ordering and variable ordering heuristics, we improved the basic backtracking algorithm from problem three, allowing us to choose which variable and values to check first more efficiently.

As seen in the graphs, as board size increases, the time taken to solve increases exponentially. Board difficulty seems to have less of an effect on the time taken to solve. While there does seem to be some correlation between board difficulty and time taken, there are also outliers such as very hard, which takes a significantly greater amount of time to solve compared to all other tests, and the impossible difficulty which actually takes about as much time as the hard Sudoku.

4x4 very easy 0.006 seconds

4x4 easy 0.005 seconds

6x6 very easy 0.087 seconds

6x6 easy 0.199 seconds

6x6 medium 0.115 seconds

6x6 hard 0.039 seconds

6x6 very hard 0.083 seconds

6x6 super hard 0.336 seconds

8x8 very easy .725 seconds

8x8 easy 3.345 seconds

8x8 medium 4.821 seconds

8x8 hard 5.903 seconds

8x8 very hard 22.58 seconds

8x8 super hard 9.904 seconds

8x8 impossible 6.505 seconds

9x9 very easy 46.01 seconds

9x9 easy 10.73 seconds

9x9 medium 91.97 seconds

9x9 hard failed

9x9 harder 63.8 seconds

10x10 very easy 261.08 seconds