Name:
Huicheng Zhuang A99075768
Alvin Heng A11308554

CSE150 PA3 WRITEUP

1.Problem Description

Problem 1:

Just iterate through each of the variables and use is_assigned() to see if every variable has been assigned to a specific value.

Problem 2:

Iterate through every constraint that is related to the given variable assigned to the given value and use is_satisfied() to check if each of the constraints is satisfied.

Problem 3:

```
Procedure backtrack(csp)
Input:    csp = binary constraint satisfaction problem(sudoku)
Output:   if there is a solution to the given game.

If (csp is complete):
    return True

X := next unassigned variable
for all xi in X's domain:
    If (current assignment is consistent):
        back up current variable assignment
        X.assign(xi)
        if backtrack(csp):
            Return True
        roll back to previous backup point.
return False
```

Problem 4:

```
AC3(csp, arcs=None)
Input:      csp = binary constraint satisfaction problem
Output:     is there a solution

queue_arcs = queue(arcs if arcs is not None else csp.constraints.arcs())
while queue_arcs:
    var1, var2 = queue_arcs.pop()

    if revise(csp, var1, var2):
        if var1.domain is empty:
            return False
        for (var1, neighbor) in arcs in csp.constraints:
            if (neighbor != var2):
                queue_arcs.append((v, neighbor))
return True

Procedure revise(csp, xi, xj)
Input:      csp = binary constraint satisfaction problem(sudoku)
            xi = source node
            xj = neighbor node
Output:  if there are any revisions to the variable domains
revise(csp, xi, xj):

    revised = False

    for constraint in csp.constraints[xi, xj]:

    non_conflicts = []
    for di in xi.domain:
        non_conflicts = [dj for dj in xj.domain if constraint.is_satisfied(di, dj)]

        if everything conflicts and di in xi.domain:
            xi.domain.remove(di)
            revised = True
    return revised
```

Problem 5:

Select_unassigned_variable:

Iterate through all the variables. For each of them , compare the size of its domain with the current minimum size; if it is less than the current minimum, then set current minimum to that value and pick this variable.

If there is a tie, then compare the number of outgoing edges of that variable to the current maximum; if it is larger than the current maximum, then set maximum to that value and pick this variable.

Order_domain_values:

1. Pick out all constraints related to the input variable.
2. Put all of the variable’s neighbors into a list.
3. For each value in the variable’s domain, calculate how many times does it appear in all its neighbors domains. And this is going to be the number of choices that the current value assignment will rule out. (Since we are only handling not equal constraint relation)
4. Pair each value with its corresponding number of ruling out and add all pairs into a list , then sort it in ascending order.
5. Pull out the values in order and return

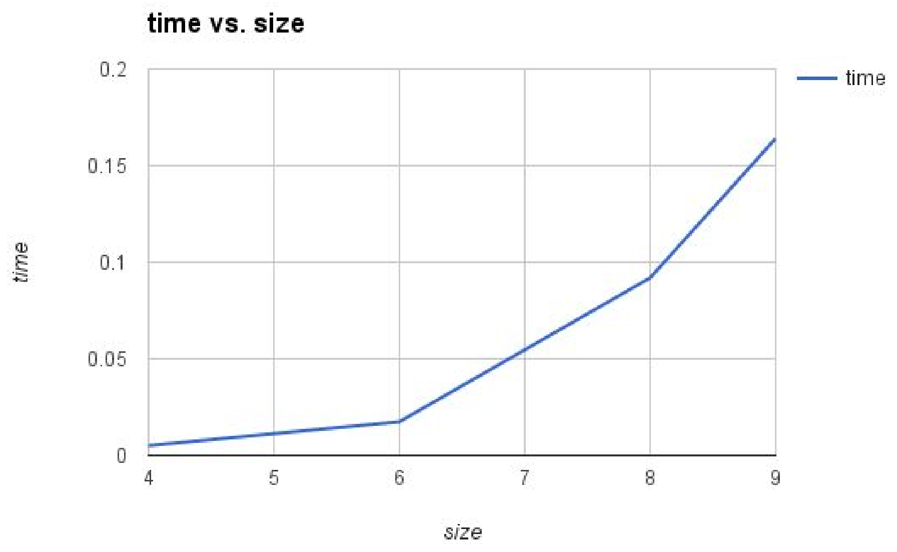
Problem 6:

```
Procedure backtrack(csp)
Input:    csp = binary constraint satisfaction problem(sudoku)
Output:   if there is a solution to the given game.

If (csp is complete):
    return True

X := next unassigned variable
for all xi in X’s domain:
    If (current assignment is consistent):
        back up current variable assignment
        X.assign(xi)
        if (inference(csp, var)):
            if backtrack(csp):
                return True
        roll back to previous backup point.
return False
```

2.Solver Test:

i) Time vs Size:

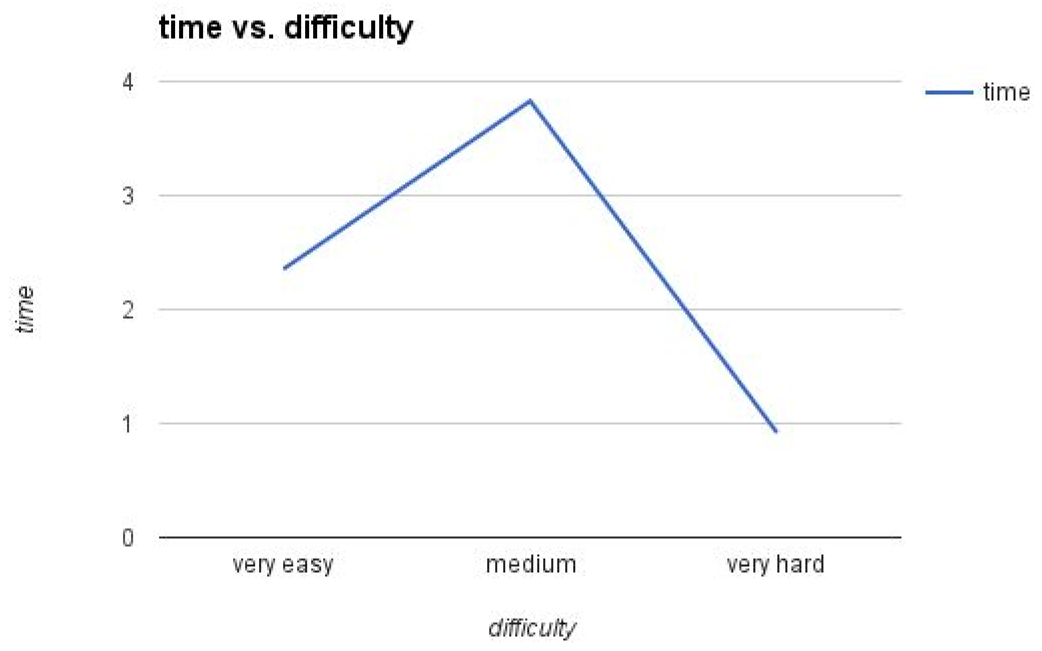
Size:	4*4	6*6	8*8	9*9
Time(s):	0.0051	0.0174	0.0919	0.1640



To test the time for solving different game board size, we pick different sizes of game boards with the same difficulty level. The time taken for solving a game increases when the size of the board gets bigger, which makes sense. And it looks like the bigger the board becomes, the faster the time taken to solve that game grows.

ii) Time vs Difficulty:

Difficulty:	Very Easy	Medium	Very Hard
Time(s):	2.3548	3.8288	0.9190



To test the time for solving games of different difficulty rates, we pick several different rating games with exactly the same board size. However, the result shows that the time actually taken to solve a game does not necessarily depends on how “difficult” that game is. And my guess is that since the difficulties are rated by human, sometimes a game seems hard to us is actually much easier to the computer.

3.Heuristic Analysis

With all 3 heuristics (p6), the algorithm took around 0.6 seconds

To test the individual heuristics we modified the backtracking algorithm in p6 to just use a single heuristic for each trial.
With just the AC-3 algorithm used for the inference step, the algorithm took around 0.38 seconds.
With just the variable selection heuristic, the algorithm took >10 minutes (didn't finish).
With just the value assignment heuristic, the algorithm took around 5.5 seconds.

While the AC-3 algorithm helped speed up the backtracking search, the variable selection and value assignment heuristics slowed down the search considerably and could probably be implemented more efficiently. The calculations for the heuristics took longer to do than it took to search without heuristics.

Without any heuristics (p3), the algorithm took over 4 seconds

4.Member Contribution:

Huicheng Zhuang:

I did p1, p2, a little bit of p3, did p4 first but failed to use AC-3 and did p5. From the huge improvement of p6 compared to p3, I learned that how important it is to use an efficient algorithm to solve a particular kind of problem.

Alvin Heng:

Implemented p3 and p6, redid p4 to use AC-3