Assignment 2

Problem 2:

a) This is an instance of a Constraint Satisfaction Problem. What is the set of variables, and what is the domain of possible values for each? How do the constrains look like?

There are 81 - M variables, one for each square n_{ij} that is not assigned by the definition of the problem.

For each variable, the domain is 1 - 9.

There are three contraints:

1. For each value in n_{1j} through n_{9j}, no number in the domain can be repeated.
2. For each value in n_{i1} through n_{i9}, no number in the domain can be repeated.
3. For each of the 9 different 3x3 boxes, no number in the domain can be repeated.

b) One way to approach the problem, is through an incremental formulation approach and apply backtracking search. Formalize this problem using an incremental formulation. What are the start state, successor function, goal test, and path cost function?

Start state: All variables are unassigned. Only the squares that belong to the set of M defined values are defined.

Successor function: Returns the Sudoku board with one newly assigned variable.

Goal test: Every variable is assigned a value within the given domain that satisfies all three contraints.

Path Cost: The number of variables defined by the agent, 81 - M.

Which heuristic for backtracking search would you expect to work better for this problem, the degree heuristic, or the minimum remaining values heuristic and why?

We know that the degree heuristic is better when there is a tie among the MRV heuristic. This is often the case when choosing the first variable to assign in an incremental formulation. However, because the Sudoku problem has M variables already filled, we are not guaranteed to have a tie among the MRV heuristic.

Because it is unlikely that all unassigned variables have the same MRV in a given state, the degree heursitic is not best. Although the degree heuristic reduces the branching factor and can make some significant decisions early that prevent a lot of backtracking later on, it fails to see which variables are most liekly to fail soon.

The degree heuristic in this case would not be helpful because every variable is involved in at most 3 constraints. MRV would be better there is a greater distribution among the possible MRV heuristic values for each variable (i.e. any unsigned variable in row 2 will have an MRV of at most 4, while any unassigned variable in row 7 will have an MRV of at most 8). Also, it is easier to sort them and pick the one with the smallest MRV.

What is the branching factor, solution depth, and maximum depth of the searchspace? What is the size of the state space?

Branching factor: (n!) * 9^n, where n is the number of nodes unassigned.

Solution depth: 81 - M

Maximum depth of searchspace: 81 - M. This is the same as the solution depth in this example because the depth of the search space corresponds to the number of variables assigned. Backtracking a tree with this characteristic ensures that the depth of the search space will not exceed the depth of the solution.

Size of the state space: $10^{(81 - M)}$ because each of the assignable variables has 10 possible options, either being assigned a value of 1-9 or not being assigned a value yet at all.

c) What, is the difference between “easy” and “hard” Sudoku problems? [Hint: There are heuristics which for easy problems will allow to quickly walk right to the solution with almost no backtracking.]

An easy Sudoku problem is one where there is little or no backtracking. Backtracking would occur when the MRV variable is chosen and we find a value variable with no legal values available.

Because the above chosen heuristic minimizes the number of searches by pruning, an "easy" Sudoku problem would be where the MRV heuristic continues to find variables with exactly 1 legal value available. This is the only scenario that guarantees no backtracking.

Therefore, a "hard" Sudoku problem would be one where the MRV heuristic continues to return a variable with a high number of legal values available, thus increasing the probability of assigning the wrong value to this variable, and increasing the number of backtracks required in the problem.

Another technique that might work well in solving the Sudoku game is local search. Please design a local search algorithm that is likely to solve Sudoku quickly, and write it in pseudocode. You may want to look at the WalkSAT algorithm for inspiration. Do you think it will work better or worse than the best incremental search algorithm on easy problems? On hard problems? Why?

```
function
    model <- a random assignment to all variables
    while true:
        if model satisfies constraints then return model
        unit <- a randomly selected unit that fails a constraint
        with probability 0.5:
            varToChange <- random variable in unit
        else
            varToReplace <- whichever variable participates in max number
of constraints
        Make value of varToChange set to least-constraining-value heuristic
```

I think that this algorithm will not work as well as the best incremental search algorithm on hard or easy problems because it relies heavily on changing a random value. While this does avoid local minimums and

ensures that it will eventually solve the problem, it does not guarantee for it to do so as quickly as the best incremental search algorithm might.