Connor Johnson

Assignment 1

1. h(P) = $distance(P, G)$, P is the current state, G is the goal state, distance id the straight line distance from P to G
2. This heuristic is good because it represents the time it would take to bike the straight line distance from the current node. Because of the fact that the distance function is the shortest possible distance to the goal, this function does not overestimate the solution so it is admissible. It is not consistent.
3.

Arad, Bucharest

algorithm: astar_graph | path: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'] | cost: 30.03 | nodes: 4

algorithm: astar_tree | path: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'] | cost: 30.03 | nodes: 4

algorithm: id_astar | path: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'] | cost: 30.03 | nodes: 10

algorithm: id_dfs | path: ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'] | cost: 30.03 | nodes: 22

Bucharest, Zerind

algorithm: astar_graph | path: ['Bucharest', 'Fagaras', 'Sibiu', 'Arad', 'Zerind'] | cost: 35.04 | nodes: 5

algorithm: astar_tree | path: ['Bucharest', 'Fagaras', 'Sibiu', 'Arad', 'Zerind'] | cost: 35.04 | nodes: 5

algorithm: id_astar | path: ['Bucharest', 'Fagaras', 'Sibiu', 'Arad', 'Zerind'] | cost: 35.04 | nodes: 19

algorithm: id_dfs | path: ['Bucharest', 'Fagaras', 'Sibiu', 'Arad', 'Zerind'] | cost: 35.04 | nodes: 511987

Craiova, Fagaras

algorithm: astar_graph | path: ['Craiova', 'Rimnicu', 'Sibiu', 'Fagaras'] | cost: 21.48 | nodes: 5

algorithm: id_astar | path: ['Craiova', 'Rimnicu', 'Sibiu', 'Fagaras'] | cost: 21.48 | nodes: 8

algorithm: id_dfs | path: ['Craiova', 'Rimnicu', 'Sibiu', 'Fagaras'] | cost: 21.48 | nodes: 14

The astar_tree algorithm caused an infinite loop

4.

a. I was surprised by how high the number of nodes if_dfs increased when the length of the path increased to 5. Also, astar_tree caused an infinite loop in the last test, This is possible because the heuristic algorithm I used is not consistent which can cause the algorithm to get caught in a loop of nodes.
b. The astar_graph is works best. It found the correct path by expanding the minimal number of nodes for each test.
c. Id_dfs is not guaranteed to be optimal because it doesn't consider the path length in its search. DFS finds the path that travels through the fewest nodes, not the shortest path cost.

Part B:

1. The variables are the patients $\{X_1, X_2, \ldots, X_n\}$
2. The domains are the N nurses that care for the patients $\{N_1, N_2, \ldots, N_n\}$
3. $\text{Max}(\text{Workload}(N)) \leq W$, $\max(\text{Workload}(N))) - \min(\text{Workload}(N))) \leq D$
4. All variables are neighbors of each other since any combination of nurses can cause one of the constraints to no longer be true
5.

```
Nurses.workload[] // array that keeps track of the workloads of all the nurses

Assign(var, val, assignment){

    If (var,val) is in assignment{

            Return failure

    }

    Assignment.add(var,val)    //Adds the var val pair to assignment

    Nurse[val] += var.workload   //adds the workload to that nurses total workload

    Return assignment

}


Unssign(var, val, assignment){

    If (var,val) is not in assignment{

            Return failure

    }
```

```
    Assignment.remove(var,val)    //removes the var val pair from assignment
    Nurse[val] -= var.workload   //subtracts the workload to that nurses total workload
    Return assignment
}
```

6.

```
checkConstraints(var1, val1, var2, val2){
        if nurses.workload[val1] + var1.workload > W or
                        nurses.workload[val2].workload + var2.workload > W
                return false   //returns false if either nurse has a workload that now
                                exceeds W
        if nurses.workload[val1] + var1.workload – min(nurses.workoad) > D
                return false   // returns false if nurses have a workload difference greater
                                than D
        if nurses.workload[val2] + var2.workload – min(nurses.workoad) > D
                return false   // returns false if nurses have a workload difference greater
                                than D

        return true   // return true if all constraints are passed
```

7. This method would be used for the backtracking algorithm. It would be called to check if a node follows the constraints. If it doesn't that branch would be cutoff from the tree. It could also be used in forward checking to keep track of possible domains for unassigned variables. checkConstraints would be called on unassigned variables to find what are the possible domains that can be assigned.
8.