Question 1

1. States: Represented by a list of cities with a search agent. In my program, these things are represented by their own unique classes. Cities maintain their own x and y coordinates while they are stored in an ArrayList. Agents keep track of the city they are visited, along with those that they have already visited and have yet to visit.

Initial State: Search agent starts on City A. Remaining cities remain unvisited.

Goal State: Search agent ends on City A. All cities are visited.

Successor Function: Generates all the remaining unvisited cities for the search agent. The g cost is the Euclidean distance between the search agent’s current city to any of these remaining cities.

1. My heuristic function generates the minimum spanning tree of the remaining nodes unvisited by the search agent. It does so by using an algorithm very similar to Kruskal’s algorithm. It iterates through all possible edges and stores vertices in disjointed sets to achieve a complexity of ~ 2 log V, where V is the number of cities in the problem.

This heuristic never overestimates the minimum remaining distance of the search agent. This is trivial as if the remaining path were cheaper, then you could construct a cheaper spanning tree using that path. Therefore, this function is admissible.

1. If the trend continues, extrapolating from the results we can see that the number of search nodes could be at least 100 billion. Such a large number of nodes would suggest a very long search time. If we divide the extrapolated results with the average number of search nodes for 16 cities and apply that ratio to the search time, then it would be 2.5 million times slower to solve the TSP for 36 cities on average.
2. The search ran out of memory on the 10th problem of the 11th folder.

Compared to the previous experiment, the expected result for 36 cities is ~IE+20 search nodes, which is a lot larger. So without the heuristic, the result is exponentially worse. We can already see that by the time we get to 9 or 10 cities, already more search nodes are needed than with 16 cities.

Question 2

1. <V, D, C>  
   V = {V00 … Vrc}, where V represents a Cell and rc represents the rows and columns in a 9x9 grid

D = {1 … 9}

C = {C1, C2, C3}, where

C1 = no 2 variables in a column can occupy the same domain  
Vic ≠ Vjc, i ≠ j, i ∈ D, j ∈ D

C2 = no 2 variables in a column can occupy the same domain  
Vri ≠ Vrj, i ≠ j, i ∈ D, j ∈ D

C3 = no 2 variables in a cell block can occupy the same domain  
SameBlock(Vij, Vkl), Vij ≠ Vkl, i ∈ D, j ∈ D, k ∈ D, l ∈ D

1. Implemented.

I found in general for versions a and b that the graph formed a bell curve that was skewed right. The more “balanced” the number of initial values were, the more time it took for the algorithms to solve the problem. I suppose this makes sense, because the less initial values there are, the more “freedom” there is to finding a solution. But having more initial values means less possibilities, which reduces the number of checks needed.

Version B has less variable assignments overall than version a. This makes sense, as forward checking allows us to remove failed searches earlier.

I did not fully implement version c due to a shortage of time. But I did implement the most constraining variable heuristic. Judging from what I have done it looks like version c is skewed left instead of right. It also seems to do better the more initial values there are. I think if I were to implement most and least constraining variable the performance from variable c would look much better.