Kendall Weistroffer
Due: 9.22.2016

Do question 4.3 (a) and (b), but do not implement anything. Instead, describe your problem representation for each algorithm and why it will work well.

Question 4.3:

A. For solving the TSP problem with a hill-climbing algorithm we have to define an objective function and an installation function to give us an idea of a random solution and a pretty good solution. The problem should be set up that the the path/cities are represented as a randomly ordered list, forming a tour. The installation function takes in nothing will return the total distance traveled amongst the cities traversed in a random order, to ensure that we do not get stuck at a random maximum (random reset). The objective function will take in some solution and will return the best solution we can find with the smallest distance (negated distance). In order to "climb the hill" or obtain a possibly better tour, we swap local neighboring cities/paths as a way to make sure that we do not create a invalid arrangement of cities/paths. This algorithm will work well because the algorithm is making small changes the local minimum/maximums to ensure that the shortest path is followed, uses a random reset set-up to ensure that we do not get stuck at a local min/max, and keeps connectedness by only swapping local neighbors. In comparison to A*, the Hill-Climbing Algorithm will provide a near-optimal solution instead of the optimality that A* guarantees. However in some cases the Hill-Climbing algorithm can solve certain problems quicker than A*.

B. For solving the TSP problem with a genetic algorithm, we first need to set up our crossover, mutation, and fitness functions, our path/cities should be represented in a list format, once again forming a tour. Since we are trying to find the shortest distance traveled, our fitness function would use the negation of the distance traveled, and our mutation would be to swap one or some of the cities (but not change the number of cities present). For our crossover function, we would select a subset of the cities/paths from each parent and add those subsets to the child. First we would select a subset from parent 1 and add that subset to the child. Then, we would fill in the remaining cities/paths from parent 2 in order, ensuing that there are no duplicate/missing paths/cities. This algorithm would work well because the "fittest" solutions are used to create sets of increasingly "fitter" solutions to the problem at hand. By setting up the tour in a list format and then changing the order that the tour appears based on orders that we know work well (choosing subsets of paths from each parent) with a slight change (mutation of swapping cities), until we reach a near-optimal solution. Comparing this algorithm with A*, we come across a similar case, were genetic algorithm provides a near-optimal solution but without the optimality that A* guarantees.