Natural Computing - Homework #1

Dylan Shaffer

February 4, 2019

Problem 1

1.1 Statement

Compare the effectiveness of the text's hill climbing (two) and simulated annealing algorithms to find the max of

$$f(x) = 2^{-2((x-0.1)/0.9)^2}(\sin(5\pi x))^6 \quad \text{with} \quad x \in [0, 1].$$

Use a real valued representation. Include a plot of the function with the location of the max and a plot of the estimate as a function of the iteration number. How sensitive are the algorithms to initial values?

1.2 Method

The methods used for this problem are the Iterative Hill Climbing and the Simulated Annealing algorithms from the text. Hill Climbing will randomly initialize a starting point and then move in the positive X direction until it reaches a next step that will be a decrease. The max value found will then be returned and then the whole process is repeated until a certain number of iterations are done. The maximum found through Hill Climbing will not always be a global maximum, however given enough iterations it will find the global maximum eventually. Simulated Annealing is initialized at zero and then will move forward from there, the process is similar to hill climbing with the addition of some random chance to make a step forward even when it makes negative progress. At every step the next step's value is compared to the current value, if the next step is greater then the program will accept the next step, however if the next step's value is lower the program will generate a random chance if the next step should be accepted. This process allows the program to escape being stuck in a local maximum and work towards a global maximum. I used a determined amount for the iterations of hill climbing and epochs for each hill climb calculation. If I set these numbers relatively high the process will normally find the correct solution. Along the way for Hill Climbing I am using a step size of 0.001 which allows for a wide search to pin point down the max.

```
def iterative_hill_climbing(iterations, epochs):
    itr = 1
    best = 0

    while (itr < iterations):
        current = hill_climbing(epochs)
        if (evaluate(current) > evaluate(best)):
            best = current

        itr += 1

    return best


def hill_climbing(epochs):
    current = random.random()
    itr = 1
    improvement_made = True

    while (itr < epochs and improvement_made):
        x_prime = current + 0.0001

        if (evaluate(x_prime) > evaluate(current)):
            current = x_prime
            improvement_made = True
        else:
            improvement_made = False

        itr += 1

    return current
```

The Hill Climbing Algorithm from the text:

$k = 0$
Initialize x
Eval(x)
while $k < kmax$ and improvement is made **do**
  $x' = x +$ perturbation
  if (Eval(x') ¿ Eval(x) **then**
    $x = x'$
  end if
  $k++$
end while

The hill climbing that was implemented starts with a best solution equal to zero and then starts from a random point along the x-axis. This point is used as the starting point for hill climbing and it will climb to the next local maximum. This process is repeated a certain number of times to allow for a

wider variety of start points. The overall best solution is kept and displayed at the end of the program.

Simulated Annealing Algorithm code

```
def simulated_annealing(init_val):
    temperature = 800
    cooling_factor = 0.001
    x = init_val
    current_energy = evaluate(init_val)
    i = 1

    while temperature > 0.01:
        x_prime = x + 0.001
        new_energy = evaluate(x_prime)

        if (new_energy >= current_energy):
            x = x_prime
            current_energy = new_energy
        elif (random.random() >
                math.exp((current_energy - new_energy) / temperature)):
            x = x_prime
            current_energy = new_energy

        temperature = temperature - i * cooling_factor
        i += 1

    return x
```

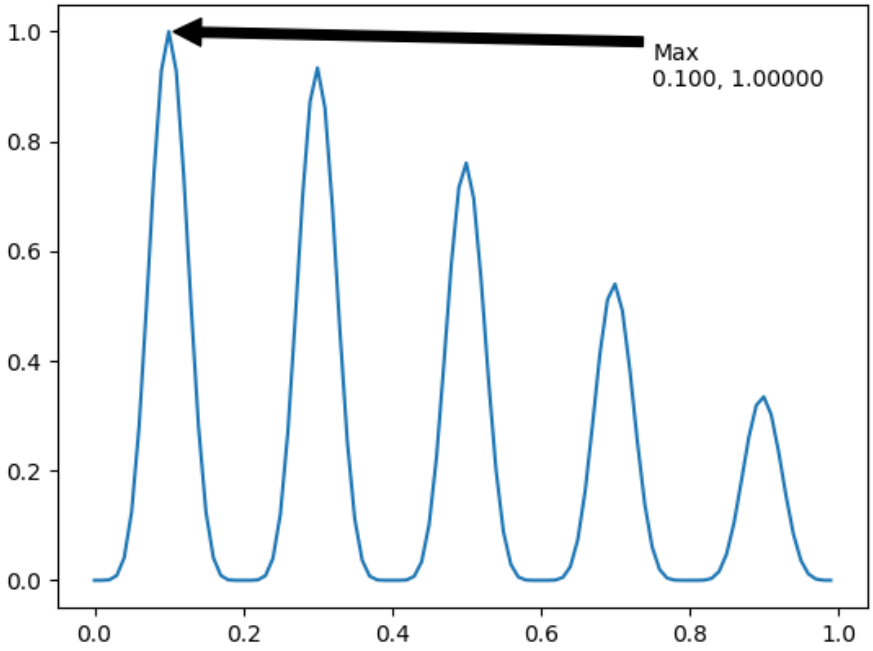
The Simulated Annealing algorithm from the text:

$k = 0$
Initialize T
Initialize x
Eval(x)
while $k < kmax$ and not done **do**
  $x' = x +$ perturbation
  Eval(x')
  if $rand(0, 1) < e^{(Eval(x)-Eval(x'))/T}$ **then**
    $x = x'$
  end if
  $k++$
  Update T
end while

The simulated annealing that is implemented will start at zero and move forward with a probabilistic chance of going forward even when the next step leads to a lower value. The method will continue to run until the temperature is cooled to 0.001. The best solution is kept throughout the method and is displayed at the end of the program.

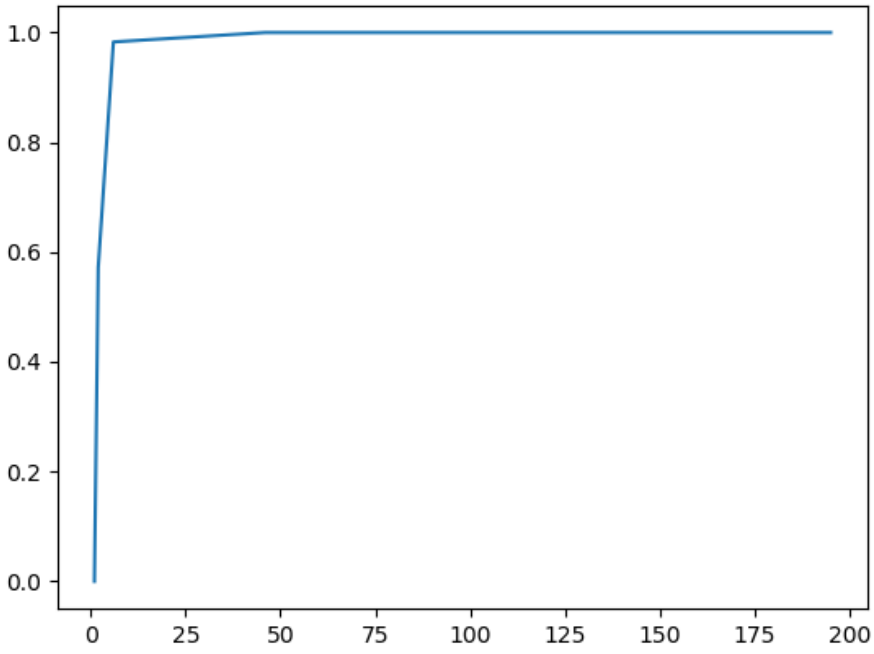
1.3 Results

Both algorithms were able to find the global maximum most of the time. Hill climbing will sometimes find a local maximum and not the global maximum but with enough generations it will more than likely be able to discover the max.

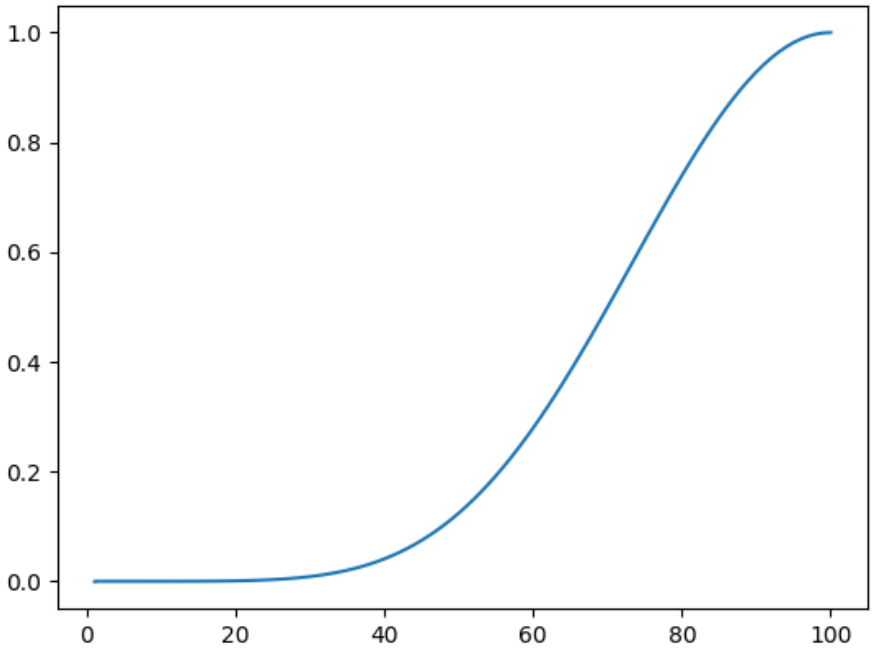


The global maximum that both algorithms were able to find.

I believe both of these algorithms are effective for finding the maximum value of a given function. Simulated Annealing definitely has an advantage in that it requires less code, it is good at getting itself out of local min/max and working towards a global. Hill Climbing was able to find the maximum value of the function but only if I allowed it enough generations / iterations. With a low amount of iterations hill climbing is not very effective because it depends on applying to a wide variety of starting points in order to properly scan the function. Simulated Annealing is also dependent upon initial values to an extent. The beginning temperature needs to be sufficiently large and the cooling factor must be relatively small otherwise the algorithm might not have enough time to find the maximum. As long as simulated annealing is given the correct parameters to last long enough, it is superior algorithm in this case.



Plot of estimated max as a function of iteration number for Hill Climbing.



Plot of estimated max as a function of iteration number for Simulated Annealing

Problem 2

2.1 Statement

In lecture we addressed the Traveling Salesman Problem using Simulated Annealing. To speed up convergence and increase the odds of finding the global extremal, it makes sense to try an evolutionary algorithm. The mutation operator can be adapted from the SA algorithm. Skip recombination in this problem. Write an evolutionary algorithm to solve the TSP as generated in the sample program. Compare deterministic and stochastic selection operators.

2.2 Experiments

For this problem I used the Evolutionary Algorithm from the text which involves initializing a population, evaluating the population, applying a selection process to reduce the population size and the mutate the remaining individual solutions to fill the population back to its maximum size. This process is repeated over and over gradually moving towards an optimal solution.

```
def evolutionary_search(cities, cities_number):
    population = initialize_population(cities_number)
    no_improvement_count = 0
    continue_search = True
    i = 0

    population = evalutate_population(population)
    population = natural_selection(population)
    current_best = population[0]

    while continue_search:
        population = variate(population, cities_number)
        population = evalutate_population(population)
        population = natural_selection(population)

        new_solution = population[0]

        if new_solution < current_best:
            current_best = new_solution
            no_improvement_count = 0
        else:
            no_improvement_count += 1

        i += 1

        if no_improvement_count > 500:
            continue_search = False

    return convert_sequence_to_solution(current_best, cities_number, cities)
```

Evolutionary Algorithm implemented in code.

$t = 0$
Initialize P
Evaluate P
while not terminated **do**
  Q = variation(P)
  Evaluate(Q)
  P = select(Q, P)
  $t = t + 1$
end while

Evolutionary Algorithm from the text.

The implementation for this problem only involves mutation during the variation stage and does not involve any recombination. For this problem the population consists of a list of arrays. The arrays are length N where N = Number_Of_Cities. These arrays consist of random integers ranging from 0 - 1000. These numbers are used to translate a sequence of random numbers into a tour of the cities using stack encoding. Specifically for an individual in the population, it will go through each random number in the sequence modding the number by N - i + 1, where i is equal to the iteration, the resulting integer is the index to use to index into the list of cities, this indexed city is used as the first city in the tour and is removed from the list and the method goes onto the next sequence building the tour until every city has been hit.

```
def convert_sequence_to_solution(sequence, cities_number, cities):
    temp_cities = cities
    new_solution = np.zeros(shape=(cities_number, 2)).astype(int)
    i = 0

    while i < cities_number:
        index = sequence[i]
        if ((cities_number - (i + 1)) <= 0:
            index = 0
        else:
            index = index % (cities_number - (i + 1))

        new_solution[i] = temp_cities[index]
        temp_cities = np.delete(temp_cities, index, 0)
        i += 1

    return new_solution
```

The conversion function used to convert a sequence of random numbers into a tour of cities.

All of the sequences are evaluated on their length of the tour and sorted from shortest to longest. Natural selection will kill off all the solutions except for the top ten shortest tours. I found that keeping a constant amount each time makes for a higher quality population. Keeping a random amount could result in potentially losing the current best solution. The population is refilled by mutating the remaining solutions. The mutation used within this program involves switching the position of two

random numbers in a sequence and the result is a new member of the population. All the mutations are done with the top ten solutions from the previous iteration. The selection of which of the top ten solutions is used for mutation is random, each iteration of mutation adds a new solution to the population and chooses a random solution from the previous top ten.

```
def mutate(population, cities_number):
    new_pop = population
    i = 0

    while i < cities_number - 10:
        index = int(np.random.randint(0, 9))
        mutant = population[index]
        mutant = swap(mutant)

        new_pop = np.insert(new_pop, len(new_pop) - 1, mutant, 0)
        i += 1

    return new_pop
```

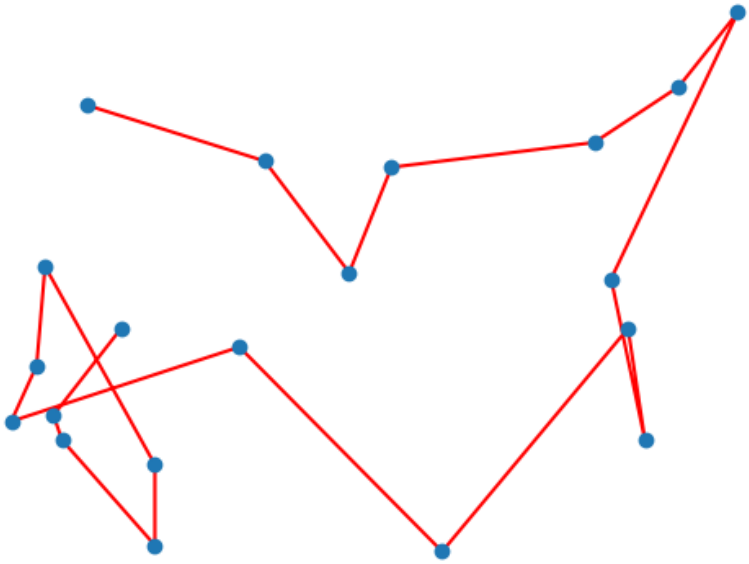
The mutation function used in the program. The swap function used will grab two random indexes and switch those positions within the sequence.

In order to guarantee that the evolution process will eventually end I implemented a counter which will increase by one every evolutionary iteration that there is no progress made (same best solution as last iteration) and will be reset to zero every time that improvement is made. I set a threshold of 5,000 on this counter so if there is no improvement for five thousand iteration the process will stop.

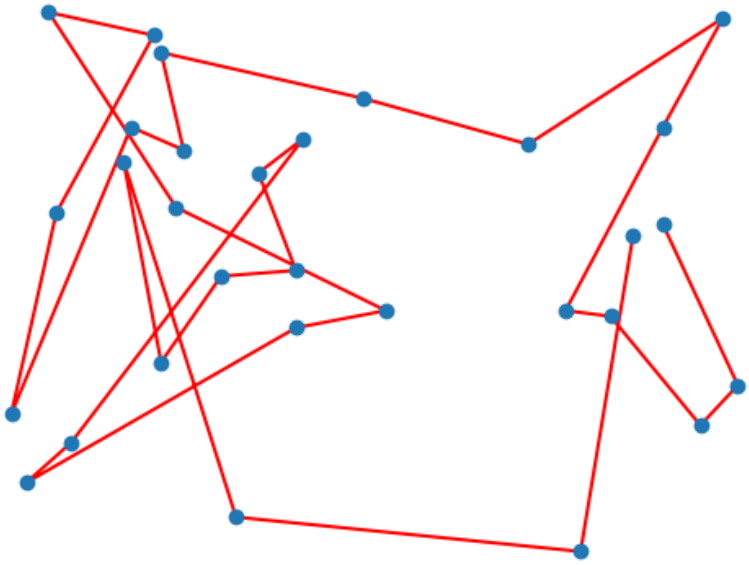
2.3 Results

After going through this experiment I found that an evolutionary algorithm will take a while to process, but will eventually converge to an acceptable solution. Sometimes the solution generated is not that great though and overall the generated solutions are not optimal but are better than a random solution. With thirty cities the algorithm would run slowly on my machine and I would have to wait a while for the best solution to look decent. I think that this slow speed is due to my machine being lower end and the algorithm requiring a lot of computing power. I specifically used ranked selection during natural selection, but I experimented with roulette selection but this method further increased the computing overhead and significantly slowed down the process so I decided to only use ranked selection. Even when the roulette selection was implemented it did not work as well as ranked. I also experimented with variations of mutation functions. Other mutations that I attempted involved sending each sequence through a random multiply/mod equation, cutting the sequence at a random index and reconnecting at opposite ends and also inverting a sub-list of the sequence. Overall these other mutations seemed to make the solution converge to a local minimum and not be able to recover and in the end swapping two random numbers works the most consistently. Using a deterministic amount for the selection turned out to be good because that way the population is always built from the fittest solutions, when using a random selection it seemed inconsistent at building better solutions. Overall I do not believe an evolutionary algorithm with stack encoding is the correct way to solve the travelling sales man problem.

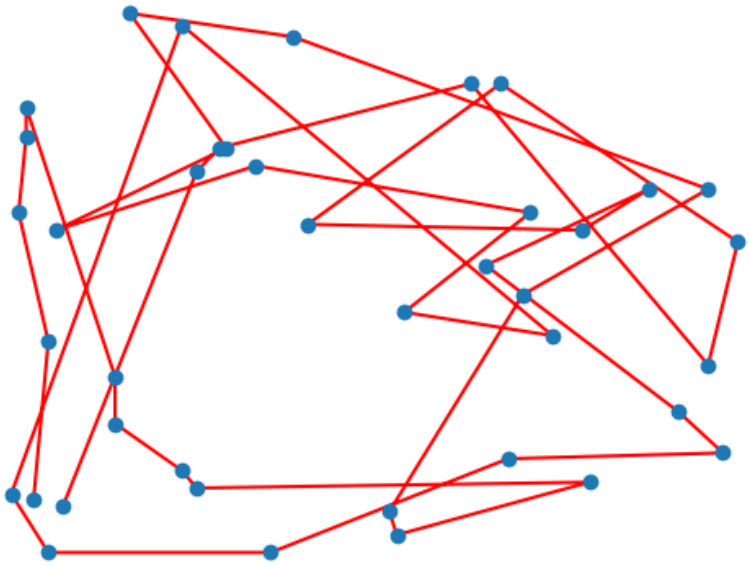
The following are results generated from the algorithm at varying number of cities involved.



Tour of 20 cities generated from the algorithm.



Tour of 30 cities generated from the algorithm.



Tour of 40 cities generated from the algorithm.

Overall the algorithm will find a decent solution that looks alright, but it starts to really struggle after 35 or more cities. This will not find the optimal solution but it finds a solution that would be good enough in most cases.

Problem 3

3.1 Statement

In problem 2, we implemented EA code to solve the Traveling Salesman Problem. In this problem, implement recombination (crossover) in your EA. For this problem you will need to use an encoding that prevents crossover that creates an invalid candidate. As before, compare deterministic and stochastic selection operators.

3.2 Experiments

This experiment build upon problem 2 and uses the same code for the most part minus some minor modifications to the variation process. The same algorithm is used as before where the population is a list of arrays of random numbers which are used to generate a city tour. So the overall evolution process is unchanged and performs and behaves mostly the same. The change that was added to variation is now mutation and recombination are both implemented adding a wider variation of solutions.

The recombination that is implemented will grab the reduced population after it has been through the selection process, then grab two random solutions (parents) split each at a random index and put opposite ends of both back together to create a new solution (child) from the two original ones. This process adds more variation into the population to attempt to widen the search space.

```
def recombination(population, cities_number):
    new_pop = population
    i = 0

    while i < cities_number:
        index1 = int(np.random.randint(0, 15))
        index2 = int(np.random.randint(0, 15))
        split = int(np.random.randint(1, len(new_pop[0])))

        parent1 = new_pop[index1]
        parent2 = new_pop[index2]

        gene1 = parent1[:split]
        gene2 = parent2[split:]

        child = np.append(gene1, gene2, 0)
        new_pop = np.insert(new_pop, len(new_pop) - 1, child, 0)
        i += 1

    return new_pop
```

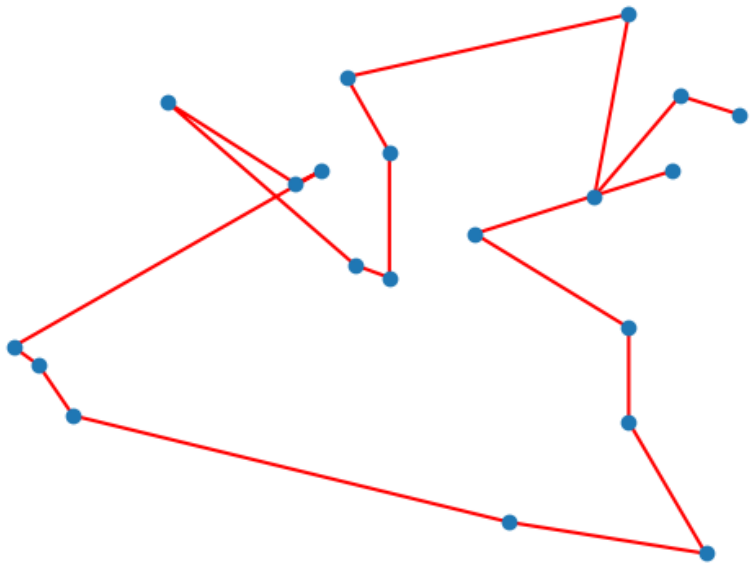
Recombination function used in the program.

3.3 Results

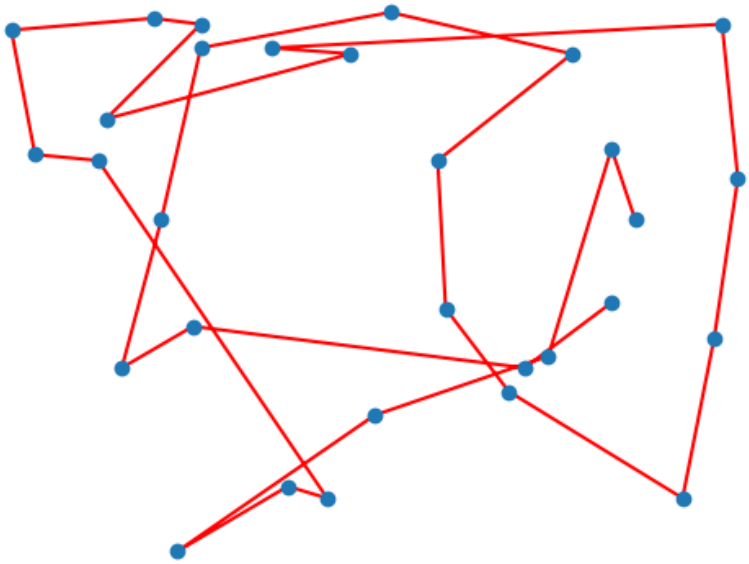
The results from this experiment show that adding recombination into the evolutionary algorithm can widen the search space of the problem by generating a wider variation of solutions. The results generated are still not optimal but they are still decent solutions and there might be slight improvement from having mutations alone. Choosing random solutions for recombination was more effective than using predetermined indexes because it allows for even more variation. The same goes for randomly selecting the split index because this means not every child is 50/50 from their parents and one of the parents will likely influence the child solution more than the other parent.

The solutions generated from after adding recombination seem similar to the tours generated before with mutation alone. As before though the algorithm starts to struggle to find a decent solution past 35 cities. I do not think that adding recombination into the variation function created any advantage over mutation alone. I think that recombination could be helpful in other problems but for stack encoding travelling sales man problem it is not helpful.

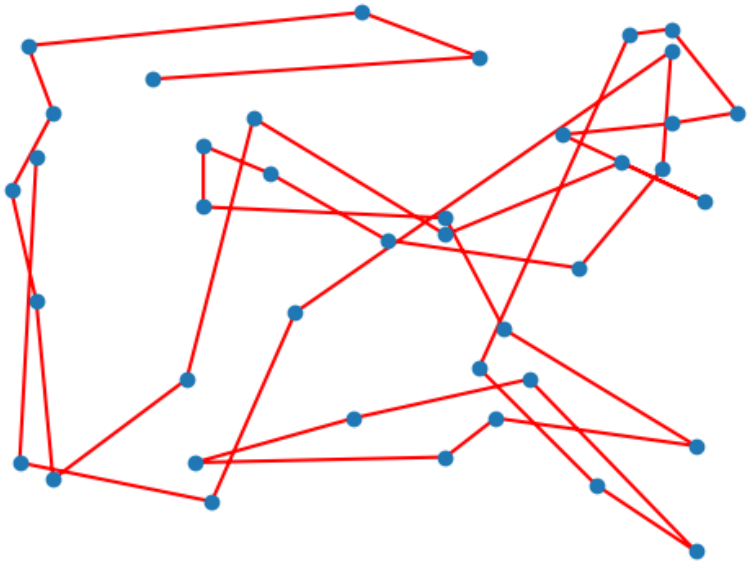
The following are results generated from the algorithm at varying number of cities involved.



Tour of 20 cities generated from the algorithm.



Tour of 30 cities generated from the algorithm.



Tour of 40 cities generated from the algorithm.

A. I. (Additional Information)

If you are interested in checking my code and work to see that I actually implemented the algorithms stated within this document I have provided a git repository containing all of my code files. For this homework I used python 3 along with numpy and scipy packages. If you have anaconda distribution then it should work perfectly.

Git Repository
https://github.com/dylan-shaffer/csc449-hw1