Project 4

Phoenix Daak
[Computer Science and Engineering]
Speed School of Engineering
University of Louisville, USA
podaak01@louisville.edu

Introduction

The Traveling Salesman Problem (TSP) is used to demonstrate the shortest path within a set of nodes. In Project 4, the TSP was completed using a Genetic Algorithmic approach. This approach demonstrated how algorithms can replicate the characteristics of nature. The approach was optimal when it comes to large datasets. Instead of a searching algorithm that searches for the next best node, the genetic approach searched for the best path overall while creating new paths from existing paths. By generating paths using the most fit paths within the population, along with a combination of mutations and crossovers, it is shown that an optimal route within the TSP can be generated.

Approach

Initial Population:

Generating the initial population involved a series of random shuffles of a list of the total nodes. First, the population size was set. This variable was used in a loop that shuffled the list of the total nodes and appended to a population list until the list reached the assigned population size. The distance of each path within the population was calculated and added to a global dictionary that was later used to compare the offsprings. The function responsible for generating the initial population can be seen in Figure 1:

```
def generateInitialPopulation(): # generating initial population
    global populationSize
    global distance
    tmpNodes = [] # used to generate initial population
    population = [] # storing the set of paths

    # appending nodes to the tmpNodes
    for node in nodes:
        tmpNodes.append(node['node'])

    for i in range(populationSize): # for the range of the set population size
        random.shuffle(tmpNodes) # shuffle the list of nodes to form a random path
        if tmpNodes not in population: # if the shuffled path is not in the population yet, add it to the population
            path = tmpNodes.copy() # path is set to the shuffled path
            dist = calcPathDistance(path) # calculate distance of path
            distance[tuple(path)] = dist # store the path to the dictionary that contains all the distances of paths
            population.append(path)
    reproduce(population) # begin generating offsprings from population
```

Figure 1. Generating the initial population

Once the initial population was generated, the reproduction of the paths began where different offsprings were generated and replaced chromosomes within the current population.

Offspring Generation (Crossovers):

After the initial population was generated, the reproduction of the fittest chromosomes began. This process of choosing the best two parents to produce an offspring involved a series of steps where the population was searched through and the path with the least distance was chosen.

A measure was taken to ensure that the two parents were not identical paths. The program for choosing the most appropriate parents can be seen below:

```
# getting the most fit for parent1
for path in population:
    if len(parent1) == 0:
        parent1 = path.copy()
    else:
        if distance[tuple(parent1)] > distance[tuple(path)]:
            parent1 = path.copy()

# getting second most fit for parent2
for path in population:
    if path != parent1:
        if len(parent2) == 0:
            parent2 = path.copy()
        else:
            if distance[tuple(parent2)] > distance[tuple(path)]:
                parent2 = path.copy()
```

Figure 2. Choosing best-fit parents for offsprings

Following the selection of parents was the crossover where different "genes" of the parents were selected and combined to generate an offspring.

The crossover process combined slices of the first parent and slices of the second parent. First, an index of where the slice from the first parent should begin in the offspring was randomly gathered. Next, the index at which the slice started within the parent was also randomly chosen. Lastly, the size of the slice from parent one was also randomly generated. After gathering the slice from the first parent, nodes that made up the slice were removed from the second parent to avoid any duplication of nodes within the path. The generation of the offspring began, starting by adding nodes from the second parent until the index of where the slice from the first parent was reached. At this point, the slice was inserted into the offspring and the insertion of nodes from the second parent continued until there were no remaining nodes. A breakdown of the crossover from the two parents is as follows:

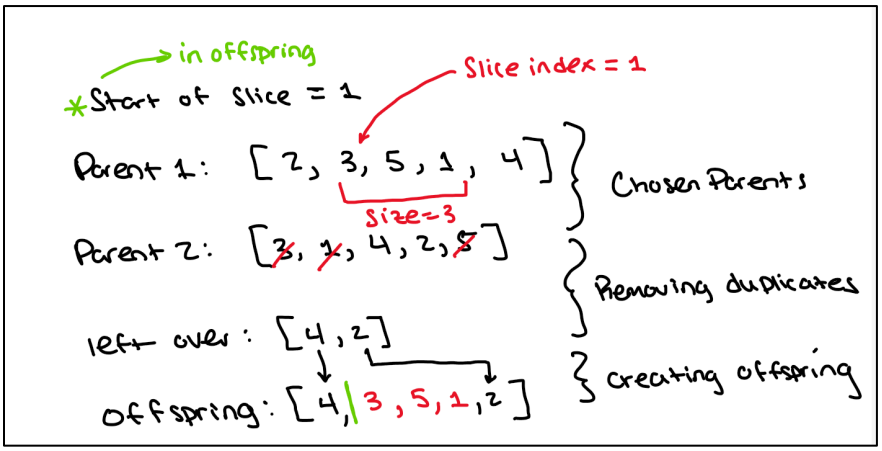

Figure 3. Breakdown of crossover

The code-equivalent to this breakdown may also be viewed below:

```
start = random.randint(1, numberOfCities//2) # getting index where slice should be stored

startOfSlice = random.randint(0, len(parent1)//2) # where the slice from parent1 will start
sizeOfSlice = random.randint(2, len(parent1)//2) + startOfSlice # size of slice from parent1

sliceOfParent1 = parent1[startOfSlice: sizeOfSlice] # getting random sample from Parent1

# handling duplicates
for node in sliceOfParent1:
    if node in parent2:
        parent2.remove(node)

leftOver = parent2 # what is left of parent2


# creating offspring
i = 0
while(i <= numberOfCities):
    if i == start:
        offspring.extend(sliceOfParent1)
        i = len(offspring) + 1
    else:
        offspring.append(leftOver[0])
        leftOver.remove(leftOver[0])
        i += 1
```

Figure 4. Performing crossover

Mutations also played a role in generating offsprings. This resulted in offsprings that did not resemble the characteristic of its parent being added into the population.

Mutations:

Introducing mutations within the offsprings happened rarely throughout the generations. The process of creating these mutations involved swapping two nodes within the offspring path. The chance of the mutation happening was kept at 0.01% but could be seen due to the large number of generations and reproductions that occurred. The process of creating a mutation within an offspring can be seen below:

```
mutation = random.randint(0, 1000) # 0.01%

if mutation == 0: # if mutation occurs, swaps two random nodes within the offspring
    nodeIndex1 = random.randint(0, len(offspring))
    nodeIndex2 = random.randint(0, len(offspring))
    offspring[nodeIndex1], offspring[nodeIndex2] = offspring[nodeIndex2] , offspring[nodeIndex1]
```

Figure 5. Generating mutations

Updating Population:

Following the crossover and generation of an offspring the population was updated. This involved comparisons of the new offspring and the least fit chromosome within the population. If the offspring was more fit than that of the existing path, the existing path was eliminated from the population and the offspring replaced the position.

The parents that were used to generate the new offspring were kept in a “couples” list preventing the same parents from creating more offsprings. This eliminated any possible

duplications of offsprings which would result in a population of the same paths. The comparison between the offspring and worst chromosome this shown below:

```
# if the offspring is not already in the population (no twins) then compare distance to worst chromosome
if offspring not in population and distOfOffspring < distance[tuple(worst)]:

    population = [path for path in population if path != worst and path != []] # removes the worst chromosome also addresses some formatting issue

    #implement some mutation
    mutation = random.randint(0, 1000) # 0.01%

    if mutation == 0: # if mutation occurs, swaps two random nodes within the offspring
        nodeIndex1 = random.randint(0, len(offspring))
        nodeIndex2 = random.randint(0, len(offspring))
        offspring[nodeIndex1], offspring[nodeIndex2] = offspring[nodeIndex2] , offspring[nodeIndex1]

    population.append(offspring) # add new offspring to the population

    # deleting worst and adding offspring to distane dictionary
    del distance[tuple(worst)]
    distance[tuple(offspring)] = distOfOffspring
```

Figure 6. Eliminating worst chromosome

Generations

The stopping criteria of the genetic algorithm was the number of generations that the reproduction ran for. Once the number of reproductions equated to the population count, a new generation was created. Assigning a generation number allowed for this evolution process described above to run at a set number of times. More generations resulted in better distanced paths. This happens because the two fittest chromosomes of the population are always used, and the worst chromosome is always removed when possible.

Results

Run Time:

The run time of the genetic algorithm was approximately 10 minutes. Modifying the population size and generations to larger numbers resulted in longer run times. For the purposes of this report, testing of the algorithm utilized a population size of 100 with 20,000 generations. These rates returned the least number of plateaus within the distance graph that will be seen later in the report.

Graph Results:

The results were returned in a graph GUI which updated every time a new-best path occurred. Below are the steps in which the algorithm generated the best path for a population size of 100, city count of 100, with 20,000 generations:

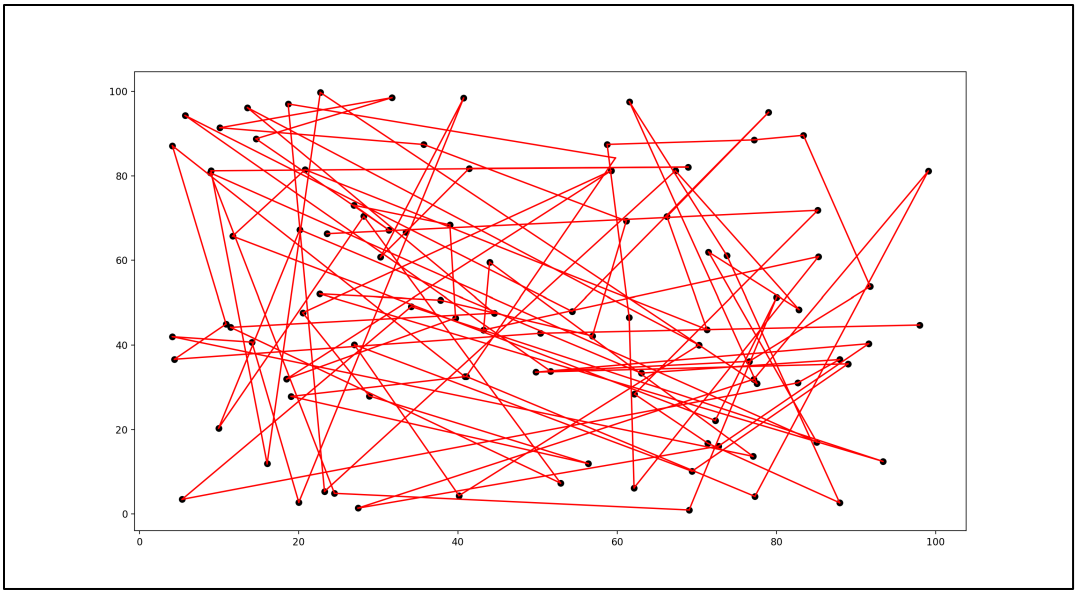

Figure 7. First generation of population

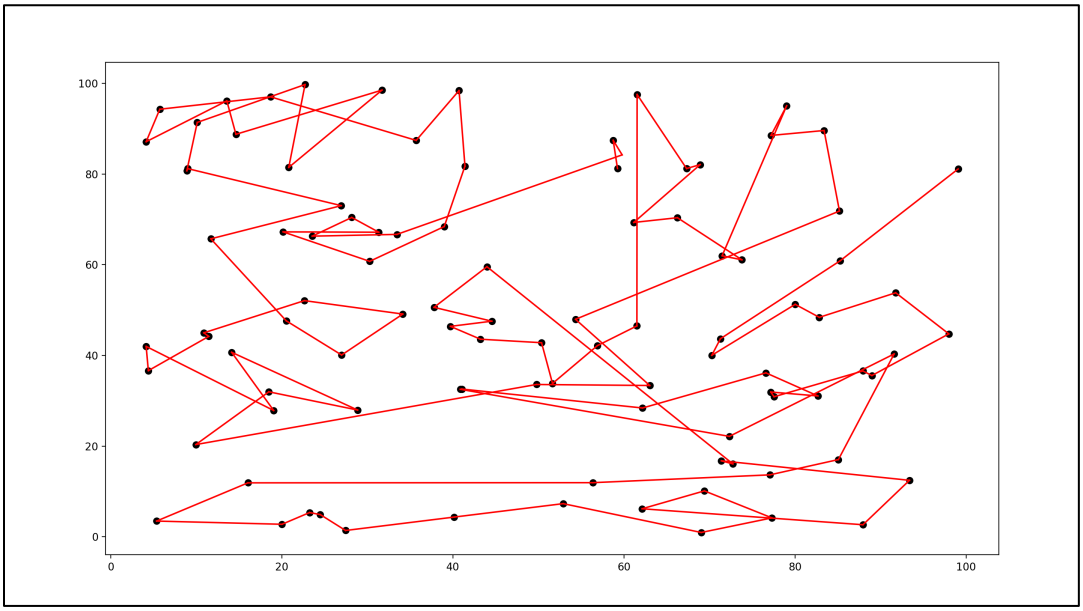

Figure 8. Approximately halfway of generation number

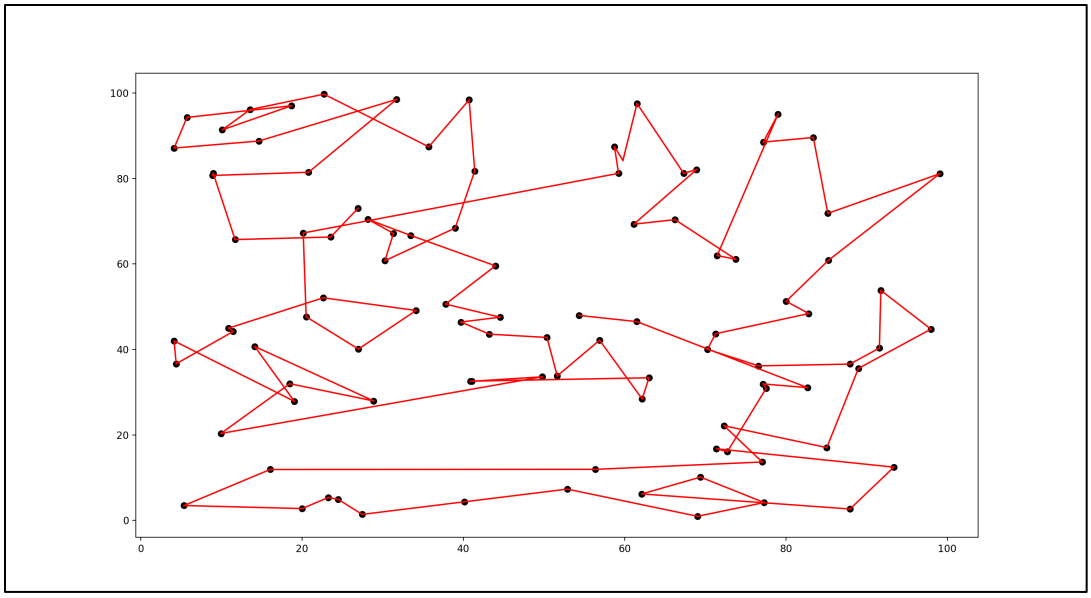

Figure 9. Last generation (20,000 generations)

A distance graph that plotted all the best path distances for each generation can be seen below:

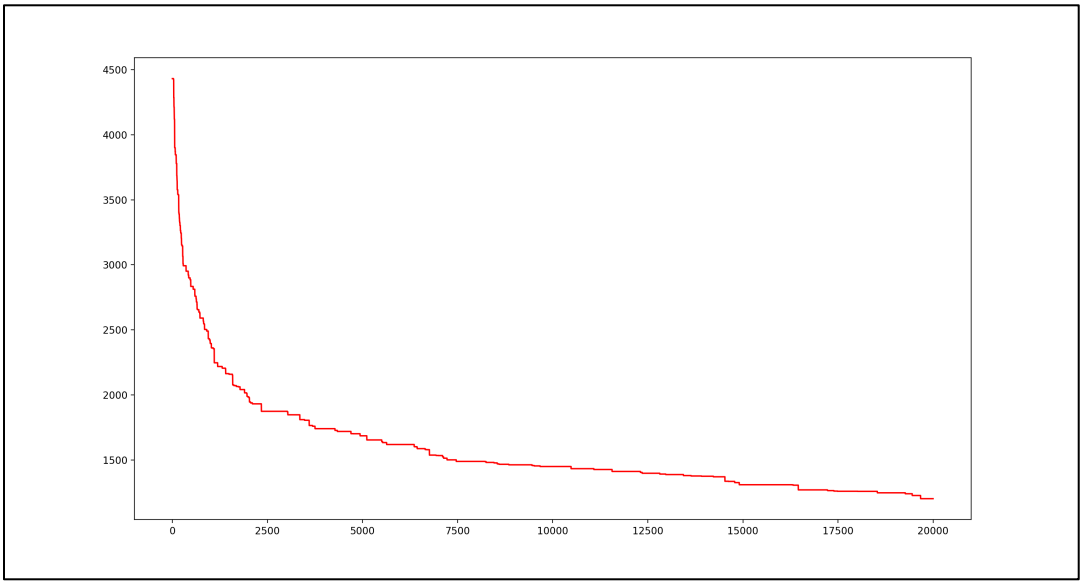

Figure 10. Distance graph for each generation

This shows that which each generation, the population grew more superior to its predecessor.

Best Solution:

The best solutions that were produced using population size of 100 with 20,000 generations resulted in a path with distance <1,500. The starting path in generation 0 often had a distance > 4,000. Below is a table that shows the starting path along with the ending path:

	Path	Distance
Starting Path	[77, 19, 83, 72, 1, 23, 45, 89, 27, 25, 5, 61, 65, 2, 87, 22, 6, 33, 31, 73, 49, 69, 53, 15, 93, 52, 28, 63, 3, 75, 10, 80, 99, 32, 34, 17, 78, 62, 85, 41, 95, 98, 42, 24, 86, 39, 55, 60, 9, 100, 8, 92, 91, 7, 20, 30, 66, 90, 37, 43, 97, 81, 74, 82, 11, 64, 79, 14, 13, 26, 94, 38, 36, 40, 18, 71, 35, 12, 70, 46, 88, 44, 67, 57, 76, 50, 48, 47, 29, 96, 4, 51, 68, 56, 54, 58, 59, 84, 16, 21]	4,431.309
Ending Path	[50, 77, 62, 5, 42, 78, 67, 57, 58, 80, 9, 44, 82, 27, 88, 22, 65, 48, 87, 76, 14, 2, 49, 12, 11, 47, 69, 16, 10, 70, 83, 94, 8, 54, 90, 13, 29, 51, 33, 56, 31, 84, 64, 59, 35, 18, 97, 4, 26, 96, 52, 100, 45, 55, 61, 46, 34, 23, 86, 32, 28, 63, 19, 91, 53, 41, 39, 17, 74, 75, 38, 66, 3, 21, 37, 24, 85, 73, 68, 25, 40, 6, 60, 98, 20, 81, 79, 95, 92, 43, 15, 1, 71, 72, 30, 89, 7, 36, 93, 99]	1,202.280

Figure 11. Starting and Ending Paths

The best solution that this genetic algorithm produced is that of the shortest distance a path can hold. By adjusting the population size and number of generations, the best solution may improve more. Fine tuning to address the best solution may be difficult but rewarding to an optimal route.

Statistics

Adjusting the parameters of the algorithm resulted in various results. Shown below is data collected from 3 different runs with 3 differing parameters.

Population Size	# of Generations	Min Distance	Max Distance	Average Distance	Average Run Time (sec)
25	1,000	2,437.65	2,602.09	2,508	53.67
50	10,000	1,593.84	1,754.34	1,604.54	200.32
100	20,000	1,200.56	1,413.92	1,225.39	600.43

Discussion

The biggest problem of implementing a genetic algorithm to solve TSP was the crossover operator. In the initial development of the algorithm, duplicates took over the population which resulted in the entire population generating offsprings of the same chromosome. Once a crossover that generated unique offsprings was developed the duplication problem ended.

One thing that would change if this project was completed again would be the crossover operator and elimination factors. These two factors play the most important role when it comes to generating a superior population.

I have learned various things while implementing a genetic algorithm. The first thing that I learned is that programming has no limitations. With this genetic algorithm, it is made to resemble that of natural reproduction to form an optimal solution. Diving deep into what goes into the crossover operator helped me learned how different crossover methods ultimately affect the entire population in the end. Another thing that I learned while implementing a genetic algorithm is how to think about the general outcome of a potential solution to a problem. With initially generating duplications within the population, it made me realize the importance of generalizing a problem so that the solution meets all the requirements. Because I was not looking down the line of different generations (100's – 1,000's) I did not see that it was the crossover method that led to these duplications. My overall impression of genetic algorithms is that they are very interesting in the way that they can be implemented however the programmer intends and also how it reflex natural characteristics.

References

CSE545 Artificial Intelligence: *Lecture 7 Beyond Classical Search*