Vehicle Routing Problem using wisdom of Crowds with Genetic Algorithms

Lindie Chenou
[Computer Science & Engineering]
Speed School of Engineering
University of Louisville, USA
lcchen01@louisville.edu

1. Introduction (What did you do in this project and why?)

The Vehicle Routing Problem (VRP) could be a generalization of the well-known non-deterministic polynomial-time difficult issue, the Traveling Salesman Problem. The VRP has been a subject of logical distribution since George Dantzig and John Ramser distributed "The Truck Dispatch Problem" in 1959. The background is also that of supplying goods to consumers who have placed orders for those goods located at a central depot. The VRP's aim is to minimize the cost of the total path. In 1964, using an effective greedy method called the savings algorithm, Clarke and Wright built on Dantzig and Ramser's approach. VRP advance have been of interest to many areas of study, including scheduling, controls, and more, and have proved to be important for key industries such as agriculture, earth science, and transport.

The context of the problem is that there are a few numbers of depots each with a few numbers of vehicles that must convey products to a set of clients

with inclination to traveling a shorter remove. The issue has numerous applications counting way arranging and classification.

The issue of VRP consists of designing low cost distribution routes, subject to a variety of side constrains, across a collection of geographically dispersed customers. This topic has a central role in the management of distribution and is faced by tens of thousands of carriers worldwide on a daily basis. Wing to the variety of constrains faced in practice, the issue occurs in many ways. The VRP has drawn the attention of a significant portion of the operational research community for over 50 years. This is partly due to the economic significance of the problem, but also to the methodological difficulties it presents. For example, for thousands and even tens of thousands of vertices, the traveling salesman problem (TSP), which is a special case of VRP, can now be solved. The VRP, by comparison, is much harder to solve. In the relatively simply case, for example, where there are only power constraints, it is still difficult to solve instances with one or two hundred clients using precise algorithms. Most of the research effort in recent years shas shifted to the production of potent metaheuristics.

2. **Approach** (Describe algorithm you are using for this project)

```python
def crossoverPopulation(matingpool, eliteSize):
    def crossover(parent1, parent2):
        child = []
        childP1 = []
        childP2 = []

        geneA = int(random.random() * len(parent1))
        geneB = int(random.random() * len(parent1))

        startGene = min(geneA, geneB)
        endGene = max(geneA, geneB)

        for i in range(startGene, endGene):
            childP1.append(parent1[i])

        childP2 = [item for item in parent2 if item not in childP1]

        child = childP1 + childP2
        return child
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = crossover(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children
```

```
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0

    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness
```

```
def mutatePopulation(population, mutationRate):
    def mutation(individual, mutationRate):
        for swapped in range(len(individual)):
            if(random.random() < mutationRate):
                swapWith = int(random.random() * len(individual))

                city1 = individual[swapped]
                city2 = individual[swapWith]

                individual[swapped] = city2
                individual[swapWith] = city1
        return individual
    mutatedPop = []

    for ind in range(0, len(population)):
        mutatedInd = mutation(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop
```

I use the genetic algorithm that has been use for previous project, it used ordered crossover. In the ordered crossover, a irregular subset of the primary parent route is chosen and after that the leftover portion of the route is filled with cities from the moment parent withing the arrange in which they show up without copying and cities chosen form the primary parent. Also, in the use of the swap mutation which is when with a indicated low likelihood, two cities

will swap places in a course. The parameters were the same for each arrangement made. The number of elite solutions kept will be 1/5 of the number of cities within the dataset. So, in case you have got 100 cities at that point the first-class estimate will be 20, on the off chance that it's 200 at that point 40 and so on. The number of generations will be 500 and the swap mutation rate will be .001.

```
for k in range(3):
    bestRoute = geneticAlgorithmPlot(population=ResCity, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)
    bestRouteList = []
    sX = []
    sY = []
    IndexRoute = []
    for j in range(len(bestRoute)):
        bestRouteList.append((bestRoute[j].x, bestRoute[j].y))
        sX.append(bestRoute[j].x)
        sY.append(bestRoute[j].y)
        IndexRoute.append(key_list[val_list.index(bestRouteList[j])])
        sX.append(bestRoute[0].x)
        sY.append(bestRoute[0].y)
        plotPath(sX, sY)
        #print(IndexRoute)
        crowd.append(IndexRoute)
        #print(agg_matrix(crowd))
    agg = agg_matrix(crowd)
    Inv_Agg = np.zeros((nn+1,nn+1))


    for k in range(nn):
        for j in range(nn):
            Inv_Agg[k,j] = 1 - sc.betaincinv(2.8,3.2,agg[k, j]/(nn))

    r = range(nn+1)
    dist = {(i, j): Inv_Agg[i,j] for i in r for j in r}
    aggRoute = tsp.tsp(r, dist)[1]
    sortedaggRoute = [Intres[i] for i in aggRoute if Intres[i] != depot]
    #import pdb; pdb.set_trace()
    cost = 0
    for u in range(nn):
        cost += City.distance(ResCity[u],ResCity[u+1])
    cost += City.distance(ResCity[nn-1],ResCity[0])
        #import pdb; pdb.set_trace()
VRP_cost += cost
VRP_Route.append(sortedaggRoute)
#import pdb; pdb.set_trace()
print(VRP_Route,VRP_cost)
```

```
cities = pd.read_table("Random11.tsp",delimiter = ' ',names=['rows_id','lat','lon'])[7:]
cityList_np = cities.to_numpy().astype(float)[:,1:3]
m = cityList_np.shape[0]
cityList_dict = dict()
R_Dist = [[dist_xy(cities.iloc[i],cities.iloc[j])  for i in range(m)] for j in range(m)]
R_D = np.asarray(R_Dist)
depot = np.random.randint(m)
#depot = 1
nber_of_vehicles = 1
Repartition = compute_partition(R_D,nber_of_vehicles)
#import pdb; pdb.set_trace()
Agg = np.zeros((m,m))
```

To make the wisdom of crowds part of the program, the program was looped around 3 times (to limit the time) and inside those loops each edge from the coming about fittest arrangement was put away in a list. So, when the circle was wrapped up executing there would be a list that contained each edge form all the created arrangement. After this I made two set of settled for circles that compared each edge form each arrangement to each other edge in all the other arrangement. And I made a check variable that would keep track of on the off chance that it found the same edge inside another arrangement. At that point once one edge has wrapped up being compared to all others in the event that the check variable was break even with to half or more than half of the arrangement it would be included to a list so it can be included within the last arrangement. With importing matplotlib.pyplot, which is a plotting library for the python programming language which incorporates a universally useful GUI that will show the output produce. First, two rundowns that is utilized form past projects that held the x and y coordinates for every city and put away them in independent rundown. With that, two more list would be created to take the path generated by the greedy function. To use the indexes of the cities in the path in order to append the x and y coordinates in the proper order that corresponds to where the city is located in the generated path. So, they can be plotted in the correct request, at that point a capacity was made that would take

in a rundown of x directions and y facilitates it would then plot every one of the focuses in the request showed by the rundown and show them utilizing the GUI

To attempt to solve VRP multiple steps were taken to pursue this. First the cities where partition into multiple subset depending on the number of vehicles. Then the cost of each partition was computed. Then the partition with the minimum cost was selected. And last the cost for each subset was solve using WoC from the TSP for each vehicle.

```
def subsets_k(collection, k): yield from partition_k(collection, k, k)
def partition_k(collection, min, k):
  if len(collection) == 1:
    yield [ collection ]
    return

  first = collection[0]
  for smaller in partition_k(collection[1:], min - 1, k):
    if len(smaller) > k: continue
    # insert `first` in each of the subpartition's subsets
    if len(smaller) >= min:
      for n, subset in enumerate(smaller):
        yield smaller[:n] + [[ first ] + subset]  + smaller[n+1:]
    # put `first` in its own subset
    if len(smaller) < k: yield [ [ first ] ] + smaller

def compute_partition(RD,nber_vehicles):
    cost_list = []
    n = RD.shape[0]
    for nber, k_subsets in enumerate(subsets_k(list(range(n)), nber_vehicles), 1):
        cost = 0
        for u in k_subsets:
            for k in range(len(u)-1):
                cost += RD[k,k+1]
            cost += R_D[len(u)-1,depot]
        cost_list.append((cost,k_subsets))

    best_partition = min(cost_list, key=lambda yc: yc[0])

    return best_partition
```

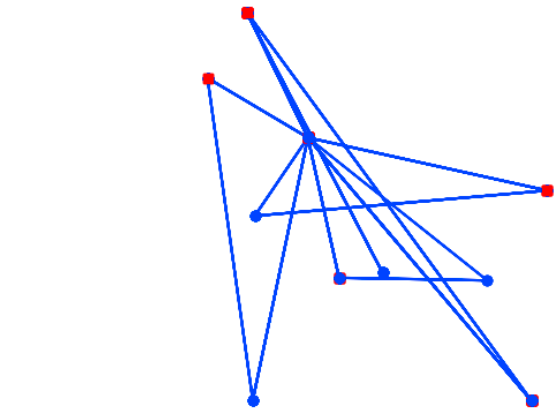
3. **Results** (How well did the algorithm perform?)

3.1 Data (Describe the data you used.)

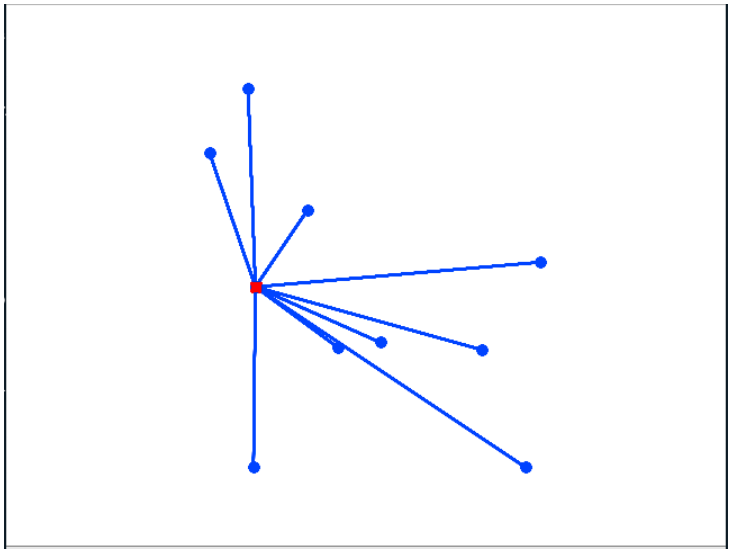
For the set of data for this project, I use the Random11.tsp file from project5.

Results (Numerical results and any figures or tables.)

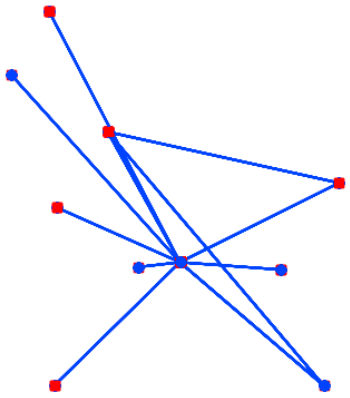
With 5 vehicles and random number for depo



```
Initial distance: 168.13987714211027
Final distance: 168.13987714211027
Initial distance: 168.13987714211027
Final distance: 168.13987714211027
Initial distance: 168.13987714211027
Final distance: 168.13987714211027
[[1, 0]] 168.13987714211027
Initial distance: 154.26376048468262
Final distance: 154.26376048468262
Initial distance: 154.26376048468262
Final distance: 154.26376048468262
Initial distance: 154.26376048468262
Final distance: 154.26376048468262
[[1, 0], [1, 2]] 287.5725612669204
Initial distance: 172.94886144210412
Final distance: 172.94886144210412
Initial distance: 172.94886144210412
Final distance: 172.94886144210412
Initial distance: 172.94886144210412
Final distance: 172.94886144210412
[[1, 0], [1, 2], [1, 4]] 344.44574328737855
Initial distance: 126.91759293524932
Final distance: 126.91759293524932
Initial distance: 126.91759293524932
Final distance: 126.91759293524932
Initial distance: 126.91759293524932
Final distance: 126.91759293524932
[[1, 0], [1, 2], [1, 4], [1, 6]] 456.14666886368866
Initial distance: 236.1863399231917
Final distance: 236.1863399231917
Initial distance: 236.1863399231917
Final distance: 236.1863399231917
Initial distance: 236.1863399231917
Final distance: 236.1863399231917
[[1, 0], [1, 2], [1, 4], [1, 6], [1, 8, 9]] 598.7453935202266
```

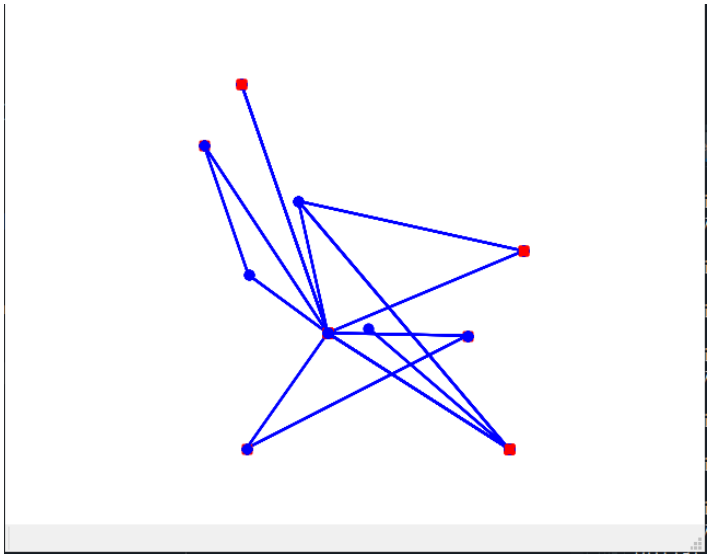


With 9 vehicle and random number for depo.



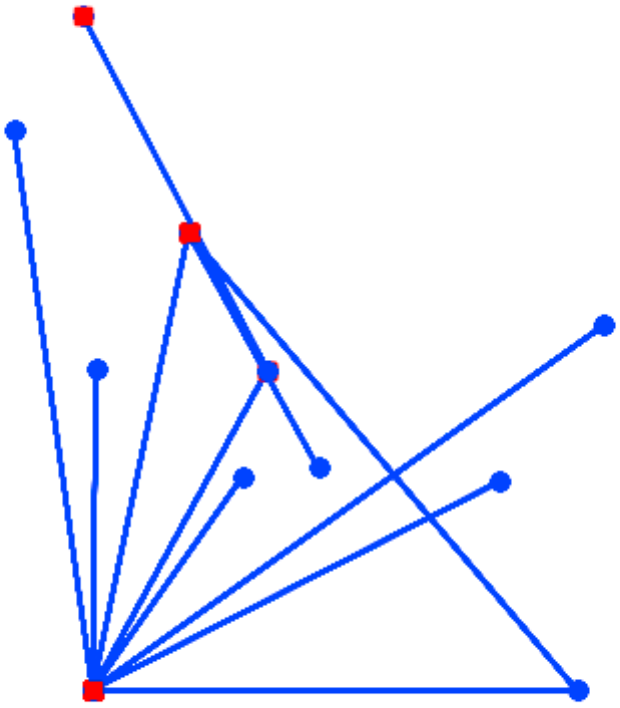
```
[[0], [2, 1], [3], [4], [5], [6], [7], [8], [1]] 469.94095694706135
```

Unique depo and 5 vehicles.



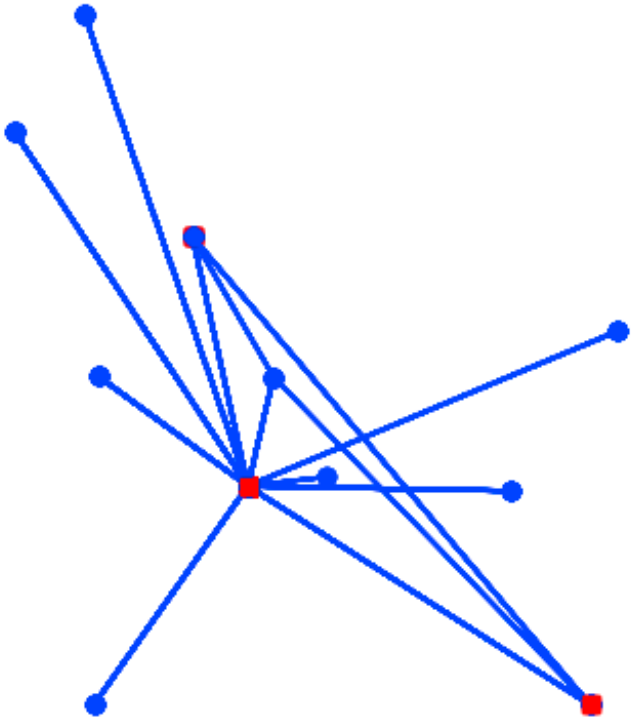
```
[[7], [7, 1], [7, 3], [7, 5], [7, 7], [7, 9]] 215.11113102462582
```

Depo = 10 with 3 vehicles



```
[[5, 1, 4, 6, 2, 3, 0, 7], [8], [1, 9]] 653.1702432589357
```

Depo = 10 with 2 vehicles



```
[[5, 1, 8, 6, 9, 0, 7, 3, 2, 4], [1]] 710.1373862819643
```

4. **Discussion** (Talk about the results you got and answer any specific questions mentioned in the assignment.)
   I consider Vehicle routing problem a big brother of the TSP problem. I use the example of how UPS operates the day to day with delivery drivers, on where they should go so, they don't overlap each other and minimize the cost. My program was able to use wisdom of crowds to attempt to solve Vehicle Routing problem. The only file that I was able to run successfully was the Random11.tsp

from previous project. The main problem that I have to with my program is the edges. I ran my program around 50 times and the edges for radom11.tsp was successful only 5. Passing the partition to the city list from the WoC was very difficult which cause a lot of errors in the results of my program. Depending on the amount of Vehicle was there the result will fluctuate. With only having one Vehicle that was the most successful output. But as the number of vehicles increase the result became worst for the file. I believe something went wrong in my implementation of VRP because I was able to implement a working code that will take the file and show the output without using WoC. In theory, with having more vehicle the algorithm is supposed to perfume better, but my program didn't accomplish that. I will continue to make improvements on the program for better result for the research paper and the presentation. The program does work but it's just isn't efficient.

5. **References** (If you used any sources in addition to lectures please include them here.)

https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35
https://www.sciencedirect.com/topics/engineering/genetic-algorithm
https://towardsdatascience.com/introduction-to-optimization-with-genetic-algorithm-2f5001d9964b