**FINAL PROJECT**

**COMPUTATIONAL METHODS FOR DATA SCIENCE**

**FALL SEMESTER 2022**

D11948002 資料科學博士班一年級 巫哲嘉

|  |
| --- |
| **Abstract**.  This project introduces different algorithms to solve the traveler salesman problem (TSP). TSP is introduced in section 1. Section 2 shows how I collect the data, i.e., distance matrix. The initialization method, Rejection Sampling, I employed to improve the computational efficiency of algorithms is introduced in section 3. Section 4 are implementations of two known metaheuristic methods, including Genetic Algorithm (GA) and Simulated Annealing (SA). Section 5 compares the results given by different algorithms. Moreover, the summary and discussion of the study will be given in this section as well. Section 6 is a bonus section to discuss whether Low-Rank approximation could have any positive influence on dealing with the distance matrix. |

1. **A Revision on Traveler Salesman Problem** *(10 points)*
2. **What is TSP?**

The process of this traveler salesman problem (TSP) is started by choosing a city in a given list, and next, find a tour that visits each of the cities exactly once and ends up by returning to the first city. The TSP asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? As a result, we have to apply optimization methods to decide the order of the cities for travelers to pass by. In this study, the cities are represented by the 7-11 convenience stores.

1. **What are the difficulties in TSP?**

In the [theory of computational complexity](https://en.wikipedia.org/wiki/Computational_complexity_theory), the TSP belongs to the class of [NP-complete](https://en.wikipedia.org/wiki/NP-completeness) problems. Therefore, it is likely that the [worst-case](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) [running time](https://en.wikipedia.org/wiki/Time_complexity) for any algorithm dealing with the TSP increases [superpolynomially](https://en.wikipedia.org/wiki/Time_complexity#Superpolynomial_time) (but no more than [exponentially](https://en.wikipedia.org/wiki/Exponential_time_hypothesis)) with the number of cities. The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a [benchmark](https://en.wikipedia.org/wiki/Benchmark_(computing)) for many optimization methods. Even though the problem is computationally difficult, many [heuristics](https://en.wikipedia.org/wiki/Heuristic) and [exact algorithms](https://en.wikipedia.org/wiki/Exact_algorithm) are known.

TSP is considered difficult because it aims to find the shortest route that includes a given set of locations and returns to the starting point. The problem is known to be NP-hard, which means that there is no known algorithm that can solve it quickly for all possible cases. As a result, a range of heuristic and approximate algorithms are proposed to find good solutions in practice.

1. **What are the applications of TSP in real life?**

The TSP has several applications even in its original formulation, such as [planning](https://en.wikipedia.org/wiki/Planning), [logistics](https://en.wikipedia.org/wiki/Logistics), and the manufacturing of [microchips](https://en.wikipedia.org/wiki/Integrated_circuit). Slightly modified, it appears as a sub-problem in some areas, including [DNA sequencing](https://en.wikipedia.org/wiki/DNA_sequencing). Among these applications, the concept of city represents, for instance, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, the trace of pheromone that ants left behind on the path, or a [similarity measure](https://en.wikipedia.org/wiki/Similarity_measure) between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources want to minimize the time spent moving the telescope between the sources; in such problems, the TSP can be embedded in an [optimization problem](https://en.wikipedia.org/wiki/Optimal_control). In many applications, additional constraints such as limited resources or time windows may be imposed.

1. **Data Collection.** *(20 points)*

The distance between two 7-11 was obtained from “Google Map” at 00:00 on Dec 8th. The distance is calculated by the route path for cars in the real world. Note that the unit of distance is meter.

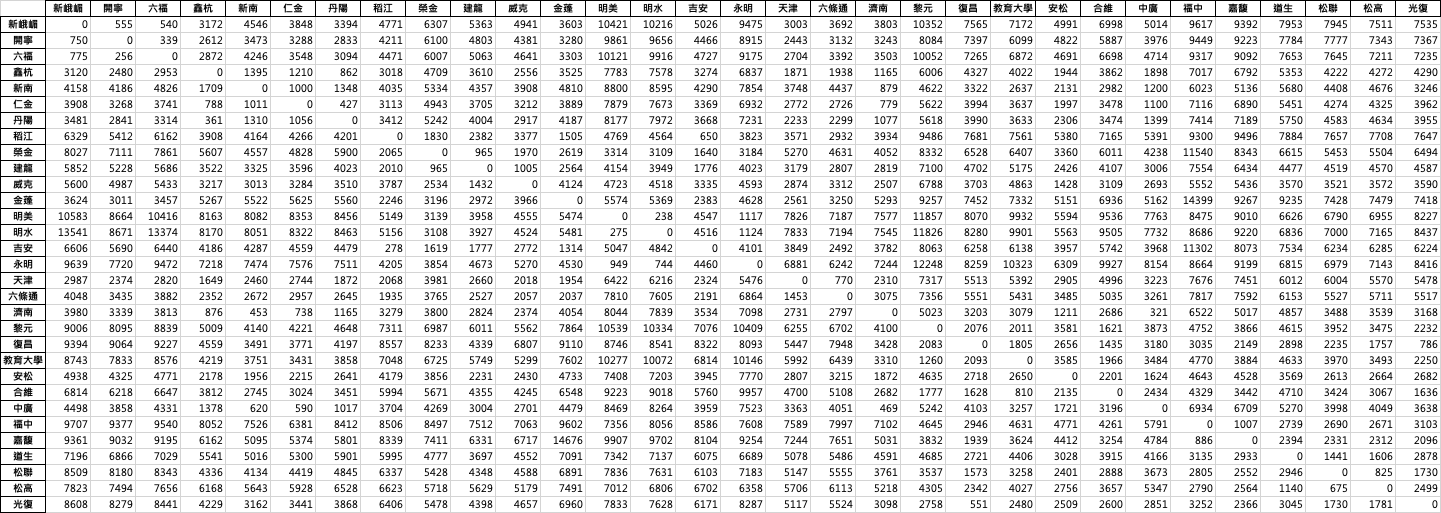


Figure 1. The distance (m) matrix for 31 7-11 stores.

1. **A Start Initialization by Rejection Sampling.** *(20 points)*
2. First, we need to know the size of our domain. Assume a computer can enumerate at most 0.5 billion sequences of random order. How many distinct nodes it can actually handle (not including the starting and ending nodes) in the path?

|  |
| --- |
| Python Code |
| def find(target=5e8):  x = 0  while math.factorial(x) <= target:  x = x + 1  if math.factorial(x) > target:  break  return x-1, math.factorial(x-1)  find() |
| Results |
|  |

It can only handle 12 distinct nodes in the path.

1. Write a simple program to generate a sequence of random order from 01 to 30.

|  |
| --- |
| Python Code |
| def random\_path(graph):  N = 30  path = []  cities\_No = list(range(len(graph)))  for i in range(1,N+1):  randval = random.randint(1, len(cities\_No)-1)  randomCity = cities\_No[randval]  path.append(randomCity)  cities\_No.remove(randomCity)  return path  path = random\_path(graph = distance\_array)  np.array(path) |
| Results |
|  |

1. Generate 1000 sequences and calculate their distances (NOTE: start from Node 00, through the path, and back to Node 00). Report their average distances and set it as the threshold.

|  |
| --- |
| Python Code |
| def path\_distance(graph, path):  N = len(path)  distance = distance\_array[0, path[0]]  for i in range(N-1):  distance = distance + graph[path[i], path[i+1]]    distance = distance + graph[path[N-1], 0]  return distance  dist\_rec = np.zeros(1000)  for i in range(1000):  path\_temp = random\_path(graph = distance\_array)  dist\_tmp = path\_distance(graph = distance\_array, path = path\_temp)  dist\_rec[i] = dist\_tmp  dist\_rec.mean() |
| Results |
|  |

The average distance of 1000 random samples is 155033.408 m which will be used as threshold in the following rejection sampling.

1. Initialization Step. Generate 1000 initial sequences with” good” distances by the spirit of rejection sampling, i.e., if the generated sequence has shorter distance than the threshold, accept it, otherwise, do something to decide whether to accept it.

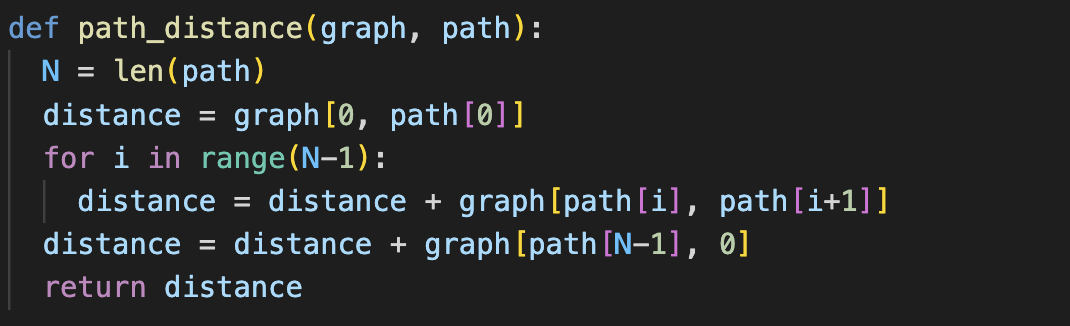
|  |
| --- |
| Python Code |
| class RejectionSampling(object):  def \_\_init\_\_(self, threshold, num\_samples):  self.threshold = threshold  self.num\_samples = num\_samples  def random\_path(self, graph):  N = 30  path = []  cities\_No = list(range(len(graph)))  for i in range(1,N+1):  randval = random.randint(1, len(cities\_No)-1)  randomCity = cities\_No[randval]  path.append(randomCity)  cities\_No.remove(randomCity)  return path  def path\_distance(self, graph, path):  N = len(path)  distance = graph[0, path[0]]  for i in range(N-1):  distance = distance + graph[path[i], path[i+1]]    distance = distance + graph[path[N-1], 0]  return distance  def good\_path\_gen(self):  init\_path = []  while len(init\_path) < self.num\_samples:  path\_tmp = self.random\_path(distance\_array)  dist\_tmp = self.path\_distance(distance\_array, path\_tmp)  is\_the\_same = [(path\_tmp == s).all() for s in init\_path]  if True not in is\_the\_same:  if dist\_tmp <= self.threshold:  init\_path.append(np.array(path\_tmp))  else:  prob = random.random()  if prob > 0.7:  init\_path.append(np.array(path\_tmp))  return init\_path  RejectionSampler = RejectionSampling(threshold=155033, num\_samples=1000)  GoodPaths = RejectionSampler.good\_path\_gen()  np.array(GoodPaths) |
| Results |
|  |

1. **An Implementation of Known Metaheuristic Methods.** *(40 points)*

In this section, please implement the following algorithms in TSP.

Some fundamental definition and ideas are summarized as follows:

* **Particle**: The particle is defined as a path that passes through 31 7-11 stores. Therefore, it is a vector with length 30 since the starting and ending point are fixed to be node 00, whose entries belong to {1, 2…30}, representing 30 7-11 stores. Furthermore, the entries of each vector are distinct.
* **Objective function**: The objective function for TSP is the function to calculate the path distance throughout all 31 7-11 and come back to the first 7-11.



* **Goal**: We want to find the path that can minimize the total distance.
* **Constraints**: All paths need to start from Node 00, through the path, and come back to Node 00. Travelers can pass by each 7-11 only once within a path.

1. **Genetic Algorithm (GA)**.

Genetic algorithm consists of three main parts: selection, crossover and mutation.

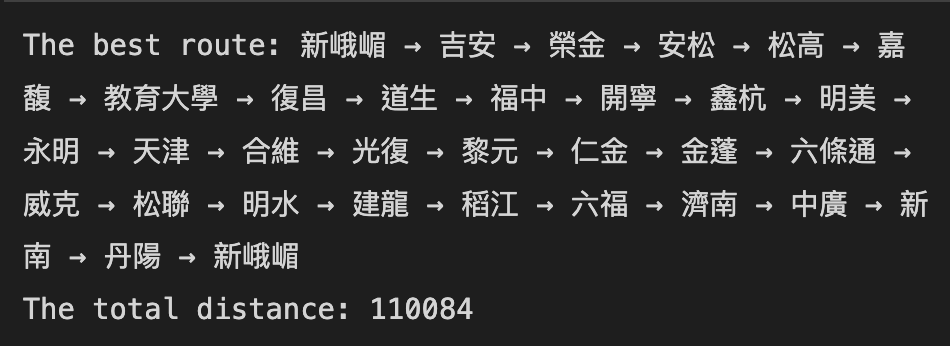
**Pseudo-code**:



* **Genetic Algorithm:** The whole algorithm has been organized as a class down below which mainly consists of roulette wheel selection, uniform crossover and mutation.

|  |
| --- |
| RejectionSampler = RejectionSampling(threshold=155033, num\_samples=1000)  GoodPaths = RejectionSampler.good\_path\_gen()  start\_time = time.time()  class Genetic\_algorithm(object):  def \_\_init\_\_(self, distance\_array, iteration, mutation\_prop):  self.distance\_array = distance\_array  self.pop\_list = None  self.iteration = iteration  self.mutation\_prop = mutation\_prop  self.population\_size = 1000  self.total\_GA = []    def get\_dist(self, seq):  seq0 = np.insert(seq, [0, len(seq)], [0, 0])  return sum([self.distance\_array[c1, c2] for c1, c2 in zip(seq0[:-1], seq0[1:])])    def roulette\_wheel\_selection(self):  list\_ = self.pop\_list.copy()  selection = []  for \_ in range(2):  fitness\_ls = [self.get\_dist(ind) for ind in list\_]  f\_sum = sum(fitness\_ls)  probability = [f/f\_sum for f in fitness\_ls]  p = np.random.random\_sample()  sum\_prob = 0  for i, prob in enumerate(probability):  sum\_prob += prob  if sum\_prob >= p:  target = list\_.pop(i)  selection.append(target)  break  return selection[0], selection[1]    def uniform\_crossover(self, gp\_1, gp\_2):  index = int(np.random.choice(len(gp\_1), 1))  new\_gp\_1, new\_gp\_2 = gp\_1[:index], gp\_2[:index]  ls\_1, ls\_2 = [], []  for g1, g2 in zip(gp\_1[index:], gp\_2[index:]):  ls\_1.append( (int(np.where(gp\_2==g1)[0]), g1) )  ls\_2.append( (int(np.where(gp\_1==g2)[0]), g2) )  ls\_1, ls\_2 = sorted(ls\_1), sorted(ls\_2)  ls\_1, ls\_2 = np.array([g for (i, g) in ls\_1]), np.array([g for (i, g) in ls\_2])  new\_gp\_1 = np.concatenate((new\_gp\_1, ls\_1))  new\_gp\_2 = np.concatenate((new\_gp\_2, ls\_2))  return new\_gp\_1, new\_gp\_2    def mutation(self, gp, point=5):  # mutation for 排列編碼，隨機選取 {point} 個點，將其向左一個位置做交換  index = np.random.choice(len(gp), 5, replace=False)  index = np.insert(index, [len(index)], [index[0]])  new\_gp = gp.copy()  for i, j in zip(index[:-1], index[1:]):  new\_gp[i] = gp[j]  return new\_gp  def select\_candidata(self, seq\_list):  # 產生 population list: 從範圍內挑選 {self.population\_size} 個  if self.population\_size < len(seq\_list):  candidata = np.random.choice(range(len(seq\_list)), self.population\_size)  self.pop\_list = [seq\_list[i] for i in candidata]  else:  self.pop\_list = seq\_list  self.best\_y = max([self.get\_dist(ind) for ind in self.pop\_list])  self.total\_GA = [self.best\_y]    def main\_program(self, seq\_list):  self.select\_candidata(seq\_list)  i = 0  while i < self.iteration:  (gp\_1, gp\_2) = self.roulette\_wheel\_selection()  new\_gp\_1, new\_gp\_2 = self.uniform\_crossover(gp\_1, gp\_2)  if np.random.random\_sample() < self.mutation\_prop:  new\_gp\_1 = self.mutation(new\_gp\_1)  new\_gp\_2 = self.mutation(new\_gp\_2)    candidate = [self.get\_dist(gp) for gp in [gp\_1, gp\_2, new\_gp\_1, new\_gp\_2]]  min\_value = [[k, v] for k, v in enumerate(candidate) if v==min(candidate)][0]    if min\_value[0] > 1:  replacement = []  for ind in self.pop\_list:  if (ind == gp\_1).all():  replacement.append(new\_gp\_1)  elif (ind == gp\_2).all():  replacement.append(new\_gp\_2)  else:  replacement.append(ind)  self.pop\_list = replacement    if min\_value[1] < self.best\_y:  if min\_value[0] == 2:  self.best\_x, self.best\_y = new\_gp\_1, min\_value[1]  elif min\_value[0] == 3:  self.best\_x, self.best\_y = new\_gp\_2, min\_value[1]  i += 1  self.total\_GA.append(self.best\_y)  GA = Genetic\_algorithm(distance\_array=distance\_array, iteration=1000, mutation\_prop=0.5)  GA.main\_program(seq\_list = GoodPaths)  end\_time = time.time()  print("--- %s secs ---" % (end\_time - start\_time))  print("The best path:", GA.best\_x, "\nThe total distance:", GA.best\_y) |

The best route generated by GA is shown below along with its corresponding total distance:

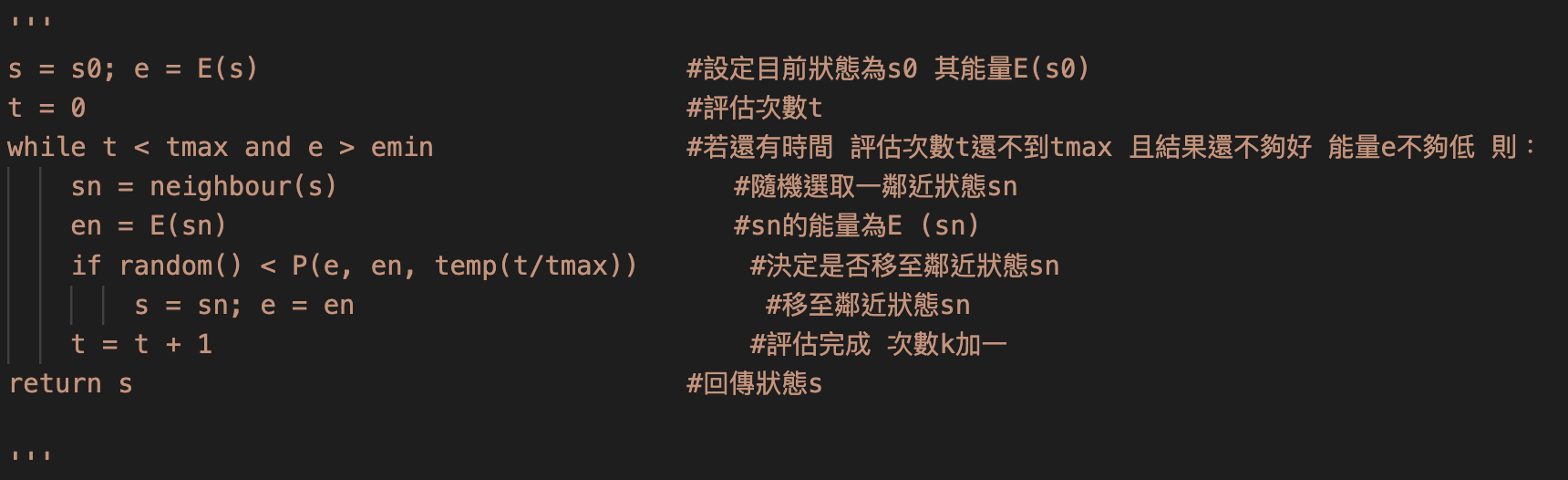


It took only 47 seconds to finish the GA computing process.

1. **Simulated Annealing (SA)**.

Similar to usual SA algorithms, is defined as new path distance minus current path distance. If , accept the new path. Otherwise, randomly accept the new path with probability , where is the current iteration number and T is temperature.

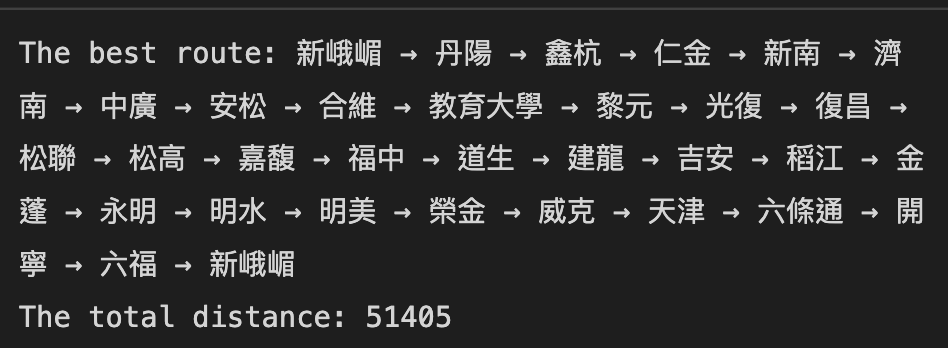
**Pseudo-code**:



* **Simulated Annealing:** The whole algorithm has been organized as a class down below which mainly consists of the random path generation, path distance calculation, how to get neighboring solutions, the temperature scheduler and the main program to execute simulated annealing.

|  |
| --- |
| start\_time = time.time()  RejectionSampler = RejectionSampling(threshold=155033, num\_samples=1000)  GoodPaths = RejectionSampler.good\_path\_gen()  class SimAnn(object):  def \_\_init\_\_(self, n\_iterations):  """  args:  n\_iteration (int): Number of iterations  """  self.n\_iterations = n\_iterations    def random\_path(self, graph):  N = 30  path = []  cities\_No = list(range(len(graph)))  for i in range(1,N+1):  randval = random.randint(1, len(cities\_No)-1)  randomCity = cities\_No[randval]  path.append(randomCity)  cities\_No.remove(randomCity)  return path  def path\_distance(self, graph, path):  N = len(path)  distance = graph[0, path[0]]  for i in range(N-1):  distance = distance + graph[path[i], path[i+1]]    distance = distance + graph[path[N-1], 0]  return distance  def getNeighbours(self, solution):  neighbours = []  for i in range(len(solution)):  for j in range(i + 1, len(solution)):  neighbour = solution.copy()  neighbour[i] = solution[j]  neighbour[j] = solution[i]  neighbours.append(neighbour)  return neighbours  def getTemp(self, t, temp):  out = (1 - t/(self.n\_iterations))\*temp  return out    def run(self, graph, init\_x, e=1e-30):  t = 0  T0 = 100  pcur\_x = np.array(init\_x)  pcur\_y = self.path\_distance(graph, path=pcur\_x)  pb\_x, pb\_y = pcur\_x, pcur\_y  x\_list = []  y\_list = []  while t < self.n\_iterations:  pcurx\_neighbors = self.getNeighbours(pcur\_x)  neighbor\_idx = random.randint(0,len(pcurx\_neighbors)-1)  pnew\_x = pcurx\_neighbors[neighbor\_idx]  pnew\_y = self.path\_distance(graph, path=pnew\_x)  dE = pnew\_y - pcur\_y  if dE <= 0:  pcur\_x, pcur\_y = pnew\_x, pnew\_y  if pcur\_y < pb\_y:  pb\_x, pb\_y = pcur\_x, pcur\_y  else:  T = self.getTemp(t, T0)  T0 = T  if np.random.random(1) < np.exp(-dE/(T+e)):  pcur\_x, pcur\_y = pnew\_x, pnew\_y  t = t + 1  x\_list.append(t)  y\_list.append(pb\_y)  return pb\_x, pb\_y, x\_list, y\_list  SimAnnealler = SimAnn(n\_iterations = 1000)  res\_x, res\_y, SA\_x, SA\_y = SimAnnealler.run(graph=distance\_array, init\_x=GoodPaths[0]) |

The best route generated by SA is shown below along with its corresponding total distance:



It took 380 seconds to finish the whole SA computing process.

1. **Summary and Discussion.** *(10 points)*
2. Summarize your comparison results on two methods.

|  |  |
| --- | --- |
| Genetic Algorithm | Simulated Annealing |
|  |  |
| 110,408 m | 51,405 m |

Table 1. The best path and distance provided by GA and SA

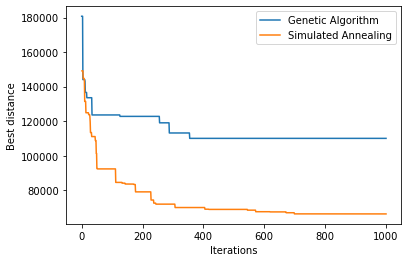


Figure 2. Best distances at every iteration by GA and SA.

1. State the advantages and disadvantages of two methods shown in this application to TSP.

Table 1 shows the best paths generated by GA and SA respectively. After 1000 iterations, the best path distance provided by SA is 51,405 m. On the other hand, the best path distance generated by GA is 110,408, which is more than twice as much as the best distance provided by SA. At this point, we can come to a short summary that SA outperforms GA in this travelling salesman problem. For further information during the training process, we can take a look at the figure 2.

From figure 2, we can see that SA performs much better than GA, which is more efficient in updating and finding the shorter paths. The main reason for why GA performs poorly is that GA uses too many random methods in the algorithm, such as crossover and mutation. Even though the random methods prevent GA from being trapped in a local optimum, it is not quite efficient in searching the global optimum. However, we find that the idea of SA is a little similar to hill climbing which makes it rather efficient in searching the local optimum. The clever part of SA is that it has a temperature cooling process which leverages a probability function to determine if it can accept a worse path at the moment. while this idea could slow down the process for SA to locate the global optimum, more importantly, it could keep SA from being trapped in a local optimum and increase the probability of finding the global optimum.

1. State at least one potential improvement on the best method to make the algorithm even better on this TSP application.

Among the two algorithms, SA and GA, the better one is SA. One thing we can try is that we can run SA several times and record the shortest path distance, for example, 50000 m. Then we alter the threshold of rejection sampling from original threshold around 153300 m to 50000 m, which I believe will increase the probability of obtaining better initial starting points. Since SA is sensitive to the choice of initial points, we could provide more nice choice of initial samples to have higher chance to get to global optimum.

Another way I would suggest is that we could change another temperature scheduler in the algorithm. For now, we are using . If we try some schedulers like or , perhaps we could achieve better performance.

Moreover, I noticed that the SA took around 370 seconds to complete the whole iterations which was way too much than I expected. Hence, I did a little modification on the getNeighbors function:

|  |  |
| --- | --- |
| Before | After |
|  |  |

Table 2. Modification on getNeighbors function

From the table above, I found that I wasted too much resources on finding permutations for neighboring solutions. In the end, I still randomly chose one of the possible solutions. So why not I just generate one possible neighboring solution and simplify the whole calculation process. After the modification, the time needed to complete the entire iterations dramatically decreased from 370 seconds to merely 35 seconds, which is nearly ten times faster than the old version. Besides, we can still get the shortest path whose distance is around 51,000 m which is similar to the solutions that the previous version could provide.

1. State the potential improvement of the initialization method, either an improvement from the current rejection sampling, or another method that replaces the rejection sampling.

Chances are that we unnecessarily need to do rejection sampling. Once we restrict our initial samples to a subset of total sample space, it is highly likely that we lead the algorithm into local optimum instead of global optimum. Therefore, I argue that we could just employ random sampling to start the initialization.

Nonetheless, if we still want to adopt rejection sampling, we could leverage the idea of temperature scheduler in SA to gradually adjust the probability of allowing paths whose total distance longer than the threshold into the candidate list of initial points.

1. **Low-Rank Approximation on Distance Matrix.** *(Bonus 20 points)*

In the first part of the course, we learn many matrix manipulation techniques that may help to reduce the matrix computations. Although there exist low-rank sparse decomposition methods for adjacency matrix (i.e., distance matrix in our case) of a graph, comment on why these methods cannot help in our problem? Or if you think it can help reducing the rank (i.e., low-rank approximation), demonstrate how it works by using the 7 × 7 sub-matrix (i.e. stores in Zhongzheng and Wanhua districts).

Firstly, I would argue that the Low-Rank Approximation cannot help in our problem. To begin with, when we are solving the TSP problems, we always need to refer to the original distance matrix, since our goal is trying to minimize the total distance of our paths. There is no point in simplifying the distance and obtain an approximated total distance of our paths which does not help us find the better solution. In addition, if we intentionally apply Low-Rank Approximation to TSP problem, it could probably take us more time to process the matrices multiplications and figure out the right dimensions.

Nevertheless, if one day we have to deal with an extremely large TSP problem with the distance matrix so large that we are not able to successfully process the entire distance matrix and keep the algorithm from operating appropriately. Then perhaps we could consider using some Low-Rank Approximation techniques. Furthermore, the reason that we cannot successfully process the entire distance matrix is the limited storage for memory from PC. From time to time, we could encounter the predicament that the data matrix consumes too much memory space that the algorithm could not work properly. Then we can consider factorizing the original distance matrix into a number of smaller low-rank matrices with LU factorization or Singular Value Decomposition, which allow us to carry out the optimization algorithms with less memory requirements. In that sense, we can say that Low-Rank Approximation is helpful in solving TSP problems.