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)*

(a) 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.

(b) What are the difficulties in TSP?

In the theory of computational complexity, the TSP belongs to the class of NP-complete problems. Therefore, it is likely that the worst-case running time for any algorithm dealing with the TSP increases superpolynomially (but no more than exponentially) 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 for many optimization methods. Even though the problem is computationally difficult, many heuristics and exact algorithms 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.

(c) What are the applications of TSP in real life?

The TSP has several applications even in its original formulation, such as planning, logistics, and the manufacturing of microchips. Slightly modified, it appears as a sub-problem in some areas, including 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 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. In many applications, additional constraints such as limited resources or time windows may be imposed.

2. 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.

	新峨嵋	開寧	六福	鑫杭	新南	仁金	丹陽	稻江	榮金	建籬	威克	金蓮	明美	明水	吉安
新峨嵋	0	555	540	3172	4546	3848	3394	4771	6307	5363	4941	3603	10421	10216	5026
開寧	750	0	339	2612	3473	3288	2833	4211	6100	4803	4381	3280	9861	9656	4466
六福	775	256	0	2872	4246	3548	3094	4471	6007	5063	4641	3303	10121	9916	4727
鑫杭	3120	2480	2953	0	1395	1210	862	3018	4709	3610	2556	3525	7783	7578	3274
新南	4158	4186	4826	1709	0	1000	1348	4035	5334	4357	3908	4810	8800	8595	4290
仁金	3908	3268	3741	788	1011	0	427	3113	4943	3705	3212	3889	7879	7673	3369
丹陽	3481	2841	3314	361	1310	1056	0	3412	5242	4004	2917	4187	8177	7972	3668
稻江	6329	5412	6162	3908	4164	4266	4201	0	1830	2382	3377	1505	4769	4564	650
榮金	8027	7111	7861	5607	4557	4828	5900	2065	0	965	1970	2619	3314	3109	1640
建籬	5852	5228	5686	3522	3325	3596	4023	2010	965	0	1005	2564	4154	3949	1776
威克	5600	4987	5433	3217	3013	3284	3510	3787	2534	1432	0	4124	4723	4518	3335
金蓮	3624	3011	3457	5267	5522	5625	5560	2246	3196	2972	3966	0	5574	5369	2383
明美	10583	8664	10416	8163	8082	8353	8456	5149	3139	3958	4555	5474	0	238	4547
明水	13541	8671	13374	8170	8051	8322	8463	5156	3108	3927	4524	5481	275	0	4516
吉安	6606	5690	6440	4186	4287	4559	4479	278	1619	1777	2772	1314	5047	4842	0
永明	9639	7720	9472	7218	7474	7576	7511	4205	3854	4673	5270	4530	949	744	4460
天津	2987	2374	2820	1649	2460	2744	1872	2068	3981	2660	2018	1954	6422	6216	2324
六條通	4048	3435	3882	2352	2672	2957	2645	1935	3765	2527	2057	2037	7810	7605	2191
濟南	3980	3339	3813	876	453	738	1165	3279	3800	2824	2374	4054	8044	7839	3534
黎元	9006	8095	8839	5009	4140	4221	4648	7311	6987	6011	5562	7864	10539	10334	7076
復昌	9394	9064	9227	4559	3491	3771	4197	8557	8233	4339	6807	9110	8746	8541	8322
教育大學	8743	7833	8576	4219	3751	3431	3858	7048	6725	5749	5299	7602	10277	10072	6814
安松	4938	4325	4771	2178	1956	2215	2641	4179	3856	2231	2430	4733	7408	7203	3945
合維	6814	6218	6647	3812	2745	3024	3451	5994	5671	4355	4245	6548	9223	9018	5760
中廣	4498	3858	4331	1378	620	590	1017	3704	4269	3004	2701	4479	8469	8264	3959
福中	9707	9377	9540	8052	7526	6381	8412	8506	8497	7512	7063	9602	7356	8056	8586
嘉馥	9361	9032	9195	6162	5095	5374	5801	8339	7411	6331	6717	14676	9907	9702	8104
道生	7196	6866	7029	5541	5016	5300	5901	5995	4777	3697	4552	7091	7342	7137	6075
松聯	8509	8180	8343	4336	4134	4419	4845	6337	5428	4348	4588	6891	7836	7631	6103
松高	7823	7494	7656	6168	5643	5928	6528	6623	5718	5629	5179	7491	7012	6806	6702
光復	8608	8279	8441	4229	3162	3441	3868	6406	5478	4398	4657	6960	7833	7628	6171

	永明	天津	六條通	濟南	黎元	復昌	教育大學	安松	合維	中廣	福中	嘉馥	道生	松聯	松高	光復
新峨嵋	9475	3003	3692	3803	10352	7565	7172	4991	6998	5014	9617	9392	7953	7945	7511	7535
開寧	8915	2443	3132	3243	8084	7397	6099	4822	5887	3976	9449	9223	7784	7777	7343	7367
六福	9175	2704	3392	3503	10052	7265	6872	4691	6698	4714	9317	9092	7653	7645	7211	7235
鑫杭	6837	1871	1938	1165	6006	4327	4022	1944	3862	1898	7017	6792	5353	4222	4272	4290
新南	7854	3748	4437	879	4622	3322	2637	2131	2982	1200	6023	5136	5680	4408	4676	3246
仁金	6932	2772	2726	779	5622	3994	3637	1997	3478	1100	7116	6890	5451	4274	4325	3962
丹陽	7231	2233	2299	1077	5618	3990	3633	2306	3474	1399	7414	7189	5750	4583	4634	3955
稻江	3823	3571	2932	3934	9486	7681	7561	5380	7165	5391	9300	9496	7884	7657	7708	7647
榮金	3184	5270	4631	4052	8332	6528	6407	3360	6011	4238	11540	8343	6615	5453	5504	6494
建籬	4023	3179	2807	2819	7100	4702	5175	2426	4107	3006	7554	6434	4477	4519	4570	4587
威克	4593	2874	3312	2507	6788	3703	4863	1428	3109	2693	5552	5436	3570	3521	3572	3590
金蓮	4628	2561	3250	5293	9257	7452	7332	5151	6936	5162	14399	9267	9235	7428	7479	7418
明美	1117	7826	7187	7577	11857	8070	9932	5594	9536	7763	8475	9010	6626	6790	6955	8227
明水	1124	7833	7194	7545	11826	8280	9901	5563	9505	7732	8686	9220	6836	7000	7165	8437
吉安	4101	3849	2492	3782	8063	6258	6138	3957	5742	3968	11302	8073	7534	6234	6285	6224
永明	0	6881	6242	7244	12248	8259	10323	6309	9927	8154	8664	9199	6815	6979	7143	8416
天津	5476	0	770	2310	7317	5513	5392	2905	4996	3223	7676	7451	6012	6004	5570	5478
六條通	6864	1453	0	3075	7356	5551	5431	3485	5035	3261	7817	7592	6153	5527	5711	5517
濟南	7098	2731	2797	0	5023	3203	3079	1211	2686	321	6522	5017	4857	3488	3539	3168
黎元	10409	6255	6702	4100	0	2076	2011	3581	1621	3873	4752	3866	4615	3952	3475	2232
復昌	8093	5447	7948	3428	2083	0	1805	2656	1435	3180	3035	2149	2898	2235	1757	786
教育大學	10146	5992	6439	3310	1260	2093	0	3585	1966	3484	4770	3884	4633	3970	3493	2250
安松	7770	2807	3215	1872	4635	2718	2650	0	2201	1624	4643	4528	3569	2613	2664	2682
合維	9957	4700	5108	2682	1777	1628	810	2135	0	2434	4329	3442	4710	3424	3067	1636
中廣	7523	3363	4051	469	5242	4103	3257	1721	3196	0	6934	6709	5270	3998	4049	3638
福中	7608	7589	7997	7102	4645	2946	4631	4771	4261	5791	0	1007	2739	2690	2671	3103
嘉馥	9254	7244	7651	5031	3832	1939	3624	4412	3254	4784	886	0	2394	2331	2312	2096
道生	6689	5078	5486	4591	4685	2721	4406	3028	3915	4166	3135	2933	0	1441	1606	2878
松聯	7183	5147	5555	3761	3537	1573	3258	2401	2888	3673	2805	2552	2946	0	825	1730
松高	6358	5706	6113	5218	4305	2342	4027	2756	3657	5347	2790	2564	1140	675	0	2499
光復	8287	5117	5524	3098	2758	551	2480	2509	2600	2851	3252	2366	3045	1730	1781	0

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

3. A Start Initialization by Rejection Sampling. *(20 points)*

(a) 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

```python
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

```
✓ def find(target=5e8): ···

(12, 479001600)
```

It can only handle 12 distinct nodes in the path.

(b) Write a simple program to generate a sequence of random order from 01 to 30.

Python Code

```python
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

```
✓ def random_path(graph): ···

array([[26,  9, 19, 14,  7, 23,  5, 24, 13,  6, 15, 30, 27,  3, 20, 10,
4, 12, 25, 21, 11, 18, 22,  8, 16, 17,  1, 28,  2, 29]])
```

(c) 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

```python
def path_distance(graph, path):
  N = len(path)
  distance = distance_array[0, path[0]]

  for i in range(N-1):
```

```python
    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

```
✓ def path_distance(graph, path): ···

155033.408
```

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

(d) 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

```python
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

```
✓ class RejectionSampling(object): ···

array([[13., 16., 26., ..., 18., 22.,  7.],
       [26., 27.,  8., ..., 17.,  7.,  1.],
       [ 9., 16., 14., ..., 19., 13., 25.],
       ...,
       [18.,  4.,  6., ..., 13., 17.,  5.],
       [ 1., 15.,  3., ..., 30.,  6., 21.],
       [11., 27.,  7., ..., 18., 20.,  4.]])


✓ GoodPaths = RejectionSampler.good_path_gen() ···

(1000, 30)
```

4. 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.

```python
def path_distance(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
```

- **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.

(a) Genetic Algorithm (GA).

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

Pseudo-code:

```
'''
Pseudocode

START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP
'''
```

- **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
```

```python
        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):
```

```python
        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:

```
The best route: 新峨嵋 → 吉安 → 榮金 → 安松 → 松高 → 嘉馥 → 教育大學 → 復昌 → 道生 → 福中 → 開寧 → 鑫杭 → 明美 → 永明 → 天津 → 合維 → 光復 → 黎元 → 仁金 → 金蓬 → 六條通 → 威克 → 松聯 → 明水 → 建龍 → 稻江 → 六福 → 濟南 → 中廣 → 新南 → 丹陽 → 新峨嵋
The total distance: 110084
```

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

(b) Simulated Annealing (SA).

Similar to usual SA algorithms, Δ_E is defined as new path distance minus current path distance. If $\Delta_E < 0$, accept the new path. Otherwise, randomly accept the new path with probability $\left(1 - \frac{t}{\#iterations}\right)T$, where t is the current iteration number and T is temperature.

Pseudo-code:

```
'''
s = s0; e = E(s)                          #設定目前狀態為s0 其能量E(s0)
t = 0                                     #評估次數t
while t < tmax and e > emin               #若還有時間 評估次數t還不到tmax 且結果還不夠好 能量e不夠低 則：
    sn = neighbour(s)                       #隨機選取一鄰近狀態sn
    en = E(sn)                              #sn的能量為E (sn)
    if random() < P(e, en, temp(t/tmax))     #決定是否移至鄰近狀態sn
        s = sn; e = en                        #移至鄰近狀態sn
    t = t + 1                                #評估完成 次數k加一
return s                                  #回傳狀態s

'''
```

- **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:

```
The best route: 新峨嵋 → 丹陽 → 鑫杭 → 仁金 → 新南 → 濟
南 → 中廣 → 安松 → 合維 → 教育大學 → 黎元 → 光復 → 復昌 →
松聯 → 松高 → 嘉馥 → 福中 → 道生 → 建龍 → 吉安 → 稻江 → 金
蓬 → 永明 → 明水 → 明美 → 榮金 → 威克 → 天津 → 六條通 → 開
寧 → 六福 → 新峨嵋
The total distance: 51405
```

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

5. Summary and Discussion. *(10 points)*

(a) Summarize your comparison results on two methods.

Genetic Algorithm	Simulated Annealing
The best route: 新峨嵋 → 吉安 → 榮金 → 安松 → 松高 → 嘉馥 → 教育大學 → 復昌 → 道生 → 福中 → 開寧 → 鑫杭 → 明美 → 永明 → 天津 → 合維 → 光復 → 黎元 → 仁金 → 金蓬 → 六條通 → 威克 → 松聯 → 明水 → 建龍 → 稻江 → 六福 → 濟南 → 中廣 → 新南 → 丹陽 → 新峨嵋	The best route: 新峨嵋 → 丹陽 → 鑫杭 → 仁金 → 新南 → 濟南 → 中廣 → 安松 → 合維 → 教育大學 → 黎元 → 光復 → 復昌 → 松聯 → 松高 → 嘉馥 → 福中 → 道生 → 建龍 → 吉安 → 稻江 → 金蓬 → 永明 → 明水 → 明美 → 榮金 → 威克 → 天津 → 六條通 → 開寧 → 六福 → 新峨嵋
110,408 m	51,405 m

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

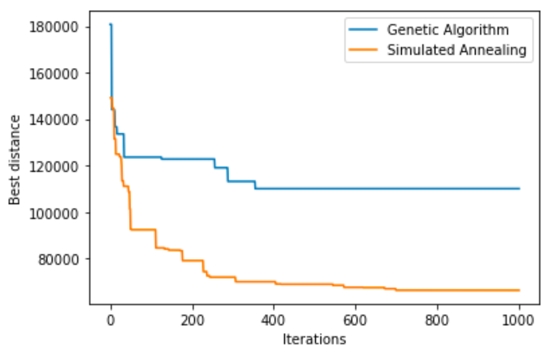

Figure 2. Best distances at every iteration by GA and SA.

(b) 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.

(c) 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 $\left(1-\frac{t}{\#iterations}\right)T$. If we try some schedulers like $\frac{T}{\log t}$ or $(1-\varepsilon)^t \times T$, 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:

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><pre><code>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</code></pre></td><td><pre><code>def getNeighbours(self, solution):
    a = np.random.choice(a = range(len(solution)),size = 2, replace=False)
    pos1 = a[0]
    pos2 = a[1]
    neighbor = solution.copy()
    neighbor[pos1] = solution[pos2]
    neighbor[pos2] = solution[pos1]
    return neighbor</code></pre></td></tr>
</table>

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.

(d) 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.

6. 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.