Report for HW2

姚博瀚 10/12/2023

Section 1: 如何實現P1(Brute Force 暴力法)

為了測試寫得對不對，首先我要有十個城市彼此間的距離（Part1）；再根據$distance_matrix$生成全部的路徑（Part2）；最後迭代所有路徑，找出擁有最短距離的那段路徑與它的總距離（Part3）。

Part 1: Given a distance matrix

Part1為隨機產生十個城市的二維座標，再計算出彼此間距離，放進一個nested list($distance\_matrix$)。$distance_matrix[i][j]$則為城市i到城市j的距離，此讀取距離的方式會被使用於Part3 $BF(distance_matrix)$裡的$calculate_total_distance(path, distance_matrix)$。

Listing 1: Given a distance matrix

```python
import math
import random

# Example:
# For reproducibility
random.seed(10)
# Generate 10 random cities in 2D space
num_cities = 10
cities_2d = [(random.uniform(0, 100), random.uniform(0, 100)) for _ in ↩
    range(num_cities)]
print("Location of cities: ", cities_2d)

# distance of two cities
def euclidean_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

distance_matrix = []
for city1 in cities_2d:
    distance_row = []
    for city2 in cities_2d:
        distance = euclidean_distance(city1, city2)
        distance_row.append(distance)
    distance_matrix.append(distance_row)
print("distance_martrix: ", distance_matrix)
```

Part 2: Generate all permutations of possible paths

使用遞迴來產生所有的路徑，一樣把每個路徑存在一個nested list。註1：這裡我直接排列$[0, 1, 2, ..., n-1]$ 當作排列n個城市的index，n為總城市數；註2：排完的路徑並不包含回最初城市，最後在BF function 裡計算總距離的地方會再加上回初始城市的距離；註3：此$generate_all_permutations(num_cities)$會包含在Part 3的$BF(distance_matrix)$裡。

Listing 2: $generate\_all\_permutations(num\_cities)$

```python
# Generate all permutations of cities
def generate_all_permutations(num_cities):
    # Initialize the list to store all permutations
    all_permutations = []

    # Helper function for recursive permutation generation
    def permute(current_path):
        # Base case: If the current path is of length num_cities, add it ↩
           to the list of permutations
        if len(current_path) == num_cities:
            all_permutations.append(current_path.copy())
            # print(f"Add {current_path} into all_permutaions")
            return
        # Recursive case: Try adding each city to the current path
        for city in range(num_cities):
            if city not in current_path:
                current_path.append(city)
                # print(f"Add {city} into current_path")
                permute(current_path)
                current_path.pop()
                # print(f"after removing:{current_path}")

    # Start the recursive permutation generation with an empty path
    permute([])
    # Return the list of all permutations
    return all_permutations

# Example:
num_cities = 3
all_permutations = generate_all_permutations(num_cities)
print("All permutations of cities: ", all_permutations)
```

Part 3: Brute Force function (BF) to iterate all of the possible paths

在迭代所有的路徑之前，必須先有產生所有路徑、計算某一路徑的總距離這兩個部分的程式，$generate_all_permutations(num_cities)$和$calculate_total_distance(path, distance_matrix)$分別為對應以上兩者的函數。每當迭代其中一個路徑則使用$calculate_total_distance(path, distance_matrix)$來計算出總距離，如小於目前最小總距離（不包含現在正迭代的路徑）則更新min_distance和min_distance_path為現正迭代的路徑。

Listing 3: $BF(distance\_matrix)$

```python
import math
import random
def BF(distance_matrix):
    # Generate all permutations of cities
    def generate_all_permutations(num_cities):
        # Initialize the list to store all permutations
        all_permutations = []

        # Helper function for recursive permutation generation
        def permute(current_path):
            # Base case: If the current path is of length num_cities, add ↩
               it to the list of permutations
            if len(current_path) == num_cities:
                all_permutations.append(current_path.copy())
                return
            # Recursive case: Try adding each city to the current path
            for city in range(num_cities):
                if city not in current_path:
                    current_path.append(city)
                    permute(current_path)
                    current_path.pop()

        # Start the recursive permutation generation with an empty path
        permute([])
        # Return the list of all permutations
        return all_permutations

    # Calculate total distance for a given path
    def calculate_total_distance(path, distance_matrix):
        total_distance = 0
        for i in range(len(path) - 1):
            total_distance += distance_matrix[path[i]][path[i + 1]]

        # Add the distance of returning to the starting city
        total_distance += distance_matrix[path[-1]][path[0]]
        return total_distance

    # Amount of cities
    num_cities = len(distance_matrix)
    # Generate all possible paths
    all_paths = generate_all_permutations(num_cities)

    # Initialize the two outputs
    min_distance = float('inf')
    min_distance_path = None

    # Iterate all paths to find the minimun distance
    for path in all_paths:
        total_distance = calculate_total_distance(path, distance_matrix)
        if total_distance < min_distance:
            min_distance = total_distance
            min_distance_path = path

    # Adjust min_distance_path to begin from 0 and end at 0
    current_tour = min_distance_path
    index_0 = min_distance_path.index(0)
    min_distance_path_part1 = list(min_distance_path[i] for i in range(↩
       index_0, num_cities))
    min_distance_path_part2 = list(min_distance_path[j] for j in range(0, ↩
       index_0))
    min_distance_path = min_distance_path_part1 + min_distance_path_part2
    min_distance_path.append(min_distance_path[0])

    return min_distance_path, min_distance
```

Section 2: 如何實現P2(Simulated Annealing 模擬退火法)

應用於P2的模擬退火法依照停止搜尋新路徑的條件，可以分成以下兩種：

1. 設定迭代數的最大值（$max\_interations$）

2. 設定未找到更短距離的路徑之連續迭代數的最大值（$stagnition\_limit$）

而在實作過程中，我想到如果把此兩種停止搜尋新路徑的方式結合在一起會不會更好？因此Section 2將只講述最大迭代數與未找到更佳解的連續最大迭代數結合在一起的程式，單獨使用以上兩種方法的程式請參考My HW2 code in Colab裡的Version of using stagnition_limit與Version of using max_interations。我採用的模擬退火法與暴力法其實有個相同之處，那便是都有個For loop來迭代路徑，但不同之處在於模擬退火法迭代的路徑，是目前最佳路徑再隨機交換兩個城市所形成的新路徑。

而模擬退火法的最大特色是即使新路徑的總距離並沒比目前最佳解更短，也會以$acceptance_probability(current_distance, new_distance, temperature)$產生的機率接受此新路徑為最佳解。此機率為一個波茲曼分佈的機率函數之變體：$p = e^{-\frac{|Distance\_current - Distance\_new|}{k \cdot Temperature}}, k = 1$。

註：此$acceptance_probability(current_distance, new_distance, temperature)$最後包含於$SA(distance_matrix)$函數內。

以下為產生此機率的Code：

Listing 4: $acceptance\_probability$

```python
def acceptance_probability(current_distance, new_distance, temperature):
        p = math.exp(-abs(current_distance - new_distance) / temperature)
        return min(1, p)
```

另外在$SA(distance_matrix)$函數內還包含以下兩個函數：

1. 計算該路徑總距離的函數（$total\_distance(tour, distance\_matrix)$）
2. 產生初始路徑的函數（$generate\_initial\_solution(num\_cities)$）

註：$generate_initial_solution(num_cities)$其實有點多餘，可直接加進$SA(distacn_matrix)$。

如以下所示：

Listing 5: Necessary functions for SA function

```python
def total_distance(tour, distance_matrix):
        distance = 0
        num_cities = len(tour)
        # path = []
        for i in range(num_cities):
            if i != (num_cities -1):
              # path.append([tour[i], tour[i+1]])
              distance += distance_matrix[tour[i]][tour[i+1]]
            else:
              # path.append([tour[i], tour[0]])
              distance += distance_matrix[tour[i]][tour[0]]

        return distance

def generate_initial_solution(num_cities):
    # return random.sample(range(num_cities), num_cities)
    return list(range(num_cities))
```

以下為$SA(distacn_matrix)$完整的Code，需注意的是在搜尋新路徑的迴圈之中加入$stagnation_limit$以控制如果已經找不太到更佳解，即停止整個迴圈不再迭代新的路徑。但$SA(distance_matrix, initial_temperature, cooling_rate, max_iterations, stagnation_limit)$仍須給定最大迭代次數（$max\_iterations$）。註：因我並沒有先固定頭尾兩個城市為0再進行迭代，所以最後有再把整個路徑調整成0開始0結束。

Listing 6: :$SAfunction$

```python
import math
import random

def SA(distance_matrix, initial_temperature, cooling_rate, max_iterations,
     stagnation_limit):

    ### Necessary functions for SA function
    def total_distance(tour, distance_matrix):
        distance = 0
        num_cities = len(tour)
        # path = []
        for i in range(num_cities):
            if i != (num_cities -1):
              # path.append([tour[i], tour[i+1]])
              distance += distance_matrix[tour[i]][tour[i+1]]
            else:
              # path.append([tour[i], tour[0]])
              distance += distance_matrix[tour[i]][tour[0]]

        return distance

    def acceptance_probability(current_distance, new_distance, temperature
       ):
        p = math.exp(-abs(current_distance - new_distance) / temperature)
        return min(1, p)

    def generate_initial_solution(num_cities):
        # return random.sample(range(num_cities), num_cities)
        return list(range(num_cities))
    ###

    num_cities = len(distance_matrix)
    current_tour = generate_initial_solution(num_cities)
    current_distance = total_distance(current_tour, distance_matrix)

    no_improvement_count = 0  # Track consecutive iterations without
       improvement

    for iteration in range(max_iterations):
        temperature = initial_temperature / (1 + cooling_rate * iteration)
        current_distance = total_distance(current_tour, distance_matrix)

        new_tour = current_tour.copy()
        # randomly select 2 cities to exchange for creating a new tour
            path
        index1, index2 = random.sample(range(num_cities), 2)
        new_tour[index1], new_tour[index2] = new_tour[index2], new_tour[index1]
        new_distance = total_distance(new_tour, distance_matrix)

        # Case of finding a path with shorter distance
        if new_distance < current_distance:
            current_tour = new_tour
            current_distance = new_distance
            no_improvement_count = 0  # Reset no_improvement_count if finding a path with shorter distance

        # Case of not finding a better path but accepting the new path under a probability
        else:
          acceptance_prob = acceptance_probability(current_distance, new_distance, temperature)
          if acceptance_prob > random.random():
              current_tour = new_tour
              current_distance = new_distance
              no_improvement_count = 0  # Reset no_improvement_count if accepting a new path
          # Consider the number of iterations without improvement and quit the for loop to find a new path if exceed a limit
          else:
              no_improvement_count += 1  # Track consecutive iterations without improvement
              if no_improvement_count > stagnation_limit:
                break

    # Adjust min_distance_path to begin from 0 and end at 0
    min_distance = current_distance
    min_distance_path = current_tour
    index_0 = min_distance_path.index(0)
    min_distance_path_part1 = list(min_distance_path[i] for i in range(index_0, num_cities))
    min_distance_path_part2 = list(min_distance_path[j] for j in range(0, index_0))
    min_distance_path = min_distance_path_part1 + min_distance_path_part2
    min_distance_path.append(min_distance_path[0])

    return min_distance_path, min_distance
```

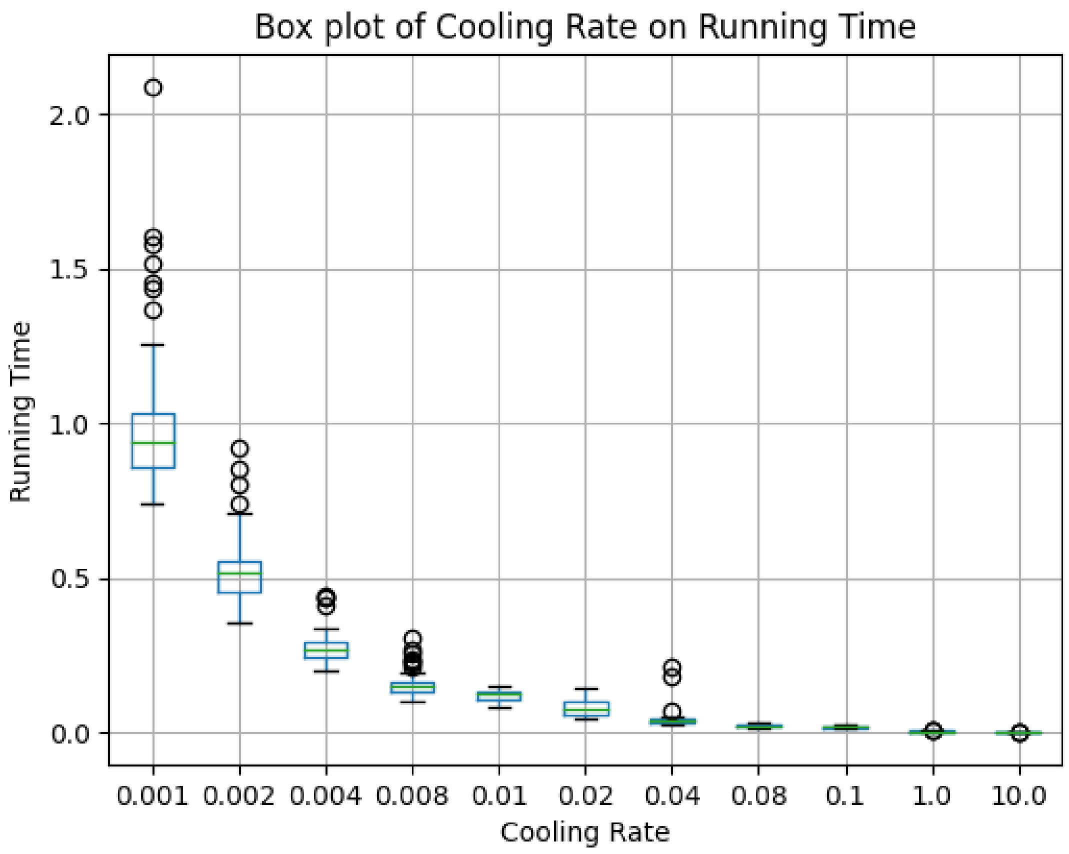
Section 3: P2實作過程中觀察到什麼？

實作P2時發現依照不同停止迭代方法，在跑程式上有不同的效率。接著在SA function的參數設置上（$distance\_matrix, initial\_temperature, cooling\_rate, max\_iterations, stagnation\_limit$）如果城市數量大於10以上很多$initial_temperature$可能需要相較10個城市時再調高。

因為我主要想看$cooling_rate$對跑完程式所需的影響（其他參數皆固定），我把$cooling_rate$當做一個Treatment，每一個level皆跑30次，紀錄下每次的所需時間再繪圖以下的Box plot：

Listing 7: Fixed conditions

```
initial_temperature = 1000.0
max_iterations = 1000000
stagnation_limit = num_cities * 20
```



另外我跑了One-way ANOVA去檢定$cooling_rate$對於running time的平均數是否有Treatment effect。如下圖所示拒絕虛無假設，所以有Treatment effect。

Listing 8: ANOVA table

```
                      sum_sq     df           F        PR(>F)
Q("Cooling Rate")  48.244823   10.0  580.823823  3.724373e-281
Residual            4.477082  539.0         NaN            NaN
```

接著我也想兩兩一對檢定彼此的running time平均數是否有顯著差異，以下為Tukey's method：

Listing 9: Multiple Comparison of Means

```
Multiple Comparison of Means - Tukey HSD, FWER=0.05
=====================================================
group1 group2 meandiff p-adj   lower   upper  reject
-----------------------------------------------------
 0.001  0.002  -0.5372    0.0 -0.6255 -0.4488   True
 0.001  0.004  -0.7664    0.0 -0.8547  -0.678   True
 0.001  0.008  -0.9089    0.0 -0.9973 -0.8206   True
 0.001   0.01  -0.9063    0.0 -0.9947  -0.818   True
 0.001   0.02  -0.9853    0.0 -1.0737  -0.897   True
 0.001   0.04  -1.0108    0.0 -1.0991 -0.9224   True
 0.001   0.08  -1.0266    0.0  -1.115 -0.9383   True
 0.001    0.1  -1.0296    0.0  -1.118 -0.9412   True
 0.001    1.0   -1.044    0.0 -1.1323 -0.9556   True
 0.001   10.0  -1.0457    0.0  -1.134 -0.9573   True
 0.002  0.004  -0.2292    0.0 -0.3176 -0.1409   True
 0.002  0.008  -0.3718    0.0 -0.4601 -0.2834   True
 0.002   0.01  -0.3692    0.0 -0.4575 -0.2808   True
 0.002   0.02  -0.4482    0.0 -0.5365 -0.3598   True
 0.002   0.04  -0.4736    0.0  -0.562 -0.3852   True
 0.002   0.08  -0.4894    0.0 -0.5778 -0.4011   True
 0.002    0.1  -0.4924    0.0 -0.5808 -0.4041   True
 0.002    1.0  -0.5068    0.0 -0.5952 -0.4185   True
 0.002   10.0  -0.5085    0.0 -0.5969 -0.4202   True
 0.004  0.008  -0.1425    0.0 -0.2309 -0.0542   True
 0.004   0.01  -0.1399    0.0 -0.2283 -0.0516   True
 0.004   0.02   -0.219    0.0 -0.3073 -0.1306   True
 0.004   0.04  -0.2444    0.0 -0.3327  -0.156   True
 0.004   0.08  -0.2602    0.0 -0.3486 -0.1719   True
 0.004    0.1  -0.2632    0.0 -0.3516 -0.1749   True
 0.004    1.0  -0.2776    0.0 -0.3659 -0.1892   True
 0.004   10.0  -0.2793    0.0 -0.3676 -0.1909   True
 0.008   0.01   0.0026    1.0 -0.0858   0.091  False
 0.008   0.02  -0.0764 0.1612 -0.1648  0.0119  False
 0.008   0.04  -0.1018 0.0099 -0.1902 -0.0135   True
 0.008   0.08  -0.1177 0.0011  -0.206 -0.0293   True
 0.008    0.1  -0.1207 0.0007  -0.209 -0.0323   True
 0.008    1.0   -0.135 0.0001 -0.2234 -0.0467   True
 0.008   10.0  -0.1368    0.0 -0.2251 -0.0484   True
  0.01   0.02   -0.079 0.1278 -0.1674  0.0093  False
  0.01   0.04  -0.1044  0.007 -0.1928 -0.0161   True
  0.01   0.08  -0.1203 0.0007 -0.2086 -0.0319   True
  0.01    0.1  -0.1233 0.0004 -0.2116 -0.0349   True
  0.01    1.0  -0.1376    0.0  -0.226 -0.0493   True
  0.01   10.0  -0.1394    0.0 -0.2277  -0.051   True
  0.02   0.04  -0.0254 0.9976 -0.1138  0.0629  False
  0.02   0.08  -0.0413 0.9143 -0.1296  0.0471  False
  0.02    0.1  -0.0443 0.8711 -0.1326  0.0441  False
  0.02    1.0  -0.0586 0.5411  -0.147  0.0297  False
  0.02   10.0  -0.0603 0.4962 -0.1487   0.028  False
  0.04   0.08  -0.0158    1.0 -0.1042  0.0725  False
  0.04    0.1  -0.0188 0.9998 -0.1072  0.0695  False
  0.04    1.0  -0.0332   0.98 -0.1216  0.0551  False
  0.04   10.0  -0.0349 0.9713 -0.1233  0.0534  False
  0.08    0.1   -0.003    1.0 -0.0913  0.0854  False
  0.08    1.0  -0.0174 0.9999 -0.1057   0.071  False
  0.08   10.0  -0.0191 0.9998 -0.1074  0.0693  False
   0.1    1.0  -0.0144    1.0 -0.1027   0.074  False
   0.1   10.0  -0.0161    1.0 -0.1044  0.0723  False
   1.0   10.0  -0.0017    1.0 -0.0901  0.0866  False
----------------------------------------------------
```

註：Reject = True，代表拒絕虛無假設，Group1和Group2的running time平均數有顯著差異。

Section 4: 比較兩者優缺點和實作的心得

暴力法的優點當然是能找到絕對的最佳解，但缺點就是如果僅僅城市數量多個1個，便需要多計算11倍的路徑量（假設原城市總數為10），並且如果此城市的座標並不是二維的話，在計算距離會再更複雜。而模擬退火法則適用於大量的城市數，在僅僅十個城市比暴力法快上十幾秒可能呈現不太出現優勢是什麼，但如果有上百個或更多的城市，則可在可接受的時間內執行完並給出可能的最佳解。

註：My HW2 code in Colab