HOMEWORK 2
COMPUTATIONAL METHODS FOR DATA SCIENCE
FALL SEMESTER 2022

D11948002 資料科學博士班一年級 巫哲嘉

1. Profit Maximization Problem for Fertilizer Company *(15 points)*

Fertilizer	Production Cost	Selling Price	Percentage Composition			
			Nitrates	Phosphates	Potash	Chalk
A	100	350	5	10	5	80
B	150	550	5	15	10	70
C	200	450	10	20	10	60
D	250	700	15	5	15	65

先假設四種肥料 A, B, C, D 的生產量為 X_A, X_B, X_C, X_D，且四種原料進口噸數為 X_{Nit}, X_{Pho}, X_{Pot}, X_{Cha}。

各原料與肥料間的成分比例可整理成以下等式：

$$
\begin{aligned}
X_{\text{Nit}} &= 0.05X_A + 0.05X_B + 0.10X_C + 0.15X_D \\
X_{\text{Pho}} &= 0.10X_A + 0.15X_B + 0.20X_C + 0.05X_D \\
X_{\text{Pot}} &= 0.05X_A + 0.10X_B + 0.10X_C + 0.15X_D \\
X_{\text{Cha}} &= 0.80X_A + 0.70X_B + 0.60X_C + 0.65X_D
\end{aligned}
$$

原料進口成本：

$$
\begin{aligned}
\text{COST}_{\text{Composition}} &= 1500X_{\text{Nit}} + 500X_{\text{Pho}} + 1000X_{\text{Pot}} + 100X_{\text{Cha}} \\
&= 255X_A + 320X_B + 410X_C + 465X_D
\end{aligned}
$$

生產成本：

$$
\text{COST}_{\text{production}} = 100X_A + 150X_B + 200X_C + 250X_D
$$

營業收入：

$$
\text{Sales} = 350X_A + 550X_B + 450X_C + 700X_D
$$

目標函數：

$$
\max\left(\text{Sales} - \text{COST}_{\text{production}} - \text{COST}_{\text{composition}}\right) = \max(-5X_A + 80X_B - 160X_C - 15X_D)
$$

限制式：

$$
\begin{aligned}
&X_A \geq 5000 \\
&X_B \geq 0 \\
&X_C \geq 0 \\
&X_D \geq 4000
\end{aligned}
$$

$$
\begin{cases} X_{\text{Nit}} < 1000 \\ X_{\text{Pho}} < 2000 \\ X_{\text{Pot}} < 1500 \end{cases} \Rightarrow \begin{cases} 0.05X_A + 0.05X_B + 0.10X_C + 0.15X_D < 1000 \\ 0.10X_A + 0.15X_B + 0.20X_C + 0.05X_D < 2000 \\ 0.05X_A + 0.10X_B + 0.10X_C + 0.15X_D < 1500 \end{cases}
$$

目標函數 $\max(-5X_A + 80X_B - 160X_C - 15X_D)$可以發現僅有肥料 B 的係數為正，換句話說當肥料 B 的數量越多，獲利就越大，因此我們將肥料 A、C 與 D 在滿足限制式的情況下設定最小值，然後試圖將肥料 B 的產量最大化。因此設定 $X_A = 5000,\ X_D = 4000,\ X_C = 0$ 後，代入限制式：

$$\begin{cases} 250 + 0.05X_B + 600 = 0.05X_B + 850 < 1000 \\ 500 + 0.15X_B + 200 = 0.15X_B + 700 < 2000 \\ 250 + 0.10X_B + 600 = 0.10X_B + 850 < 1500 \end{cases} \Rightarrow \begin{cases} 0.05X_B < 150 \\ 0.15X_B < 1300 \\ 0.10X_B < 650 \end{cases} \Rightarrow \begin{cases} X_B < 3000 \\ X_B < 8666.66 \\ X_B < 6500 \end{cases}$$

在進口條件的限制下 x_B 最多為 3000。

因此目標函數的最大值為 $-5 * 5000 + 80 * 3000 - 160 * 0 - 15 * 4000 = 155000$

2. Knapsack Problem via GA. *(35 points)*

Item	Type	Weight	Survival Points
Shadow Daggers	Knife	3.3	7
Huntsman Knife	Knife	3.4	8
Gut Knife	Knife	6.0	13
228 Compact Handgun	Pistols	26.1	29
Night Hawk	Pistols	37.6	48
Desert Eagle Magnum	Pistols	62.5	99
Ingram MAC-10 SMG	Primary	100.2	177
Leone YG1265 Auto Shotgun	Primary	141.1	213
M4A1 Carbine	Primary	119.2	202
AK-47 Rifle	Primary	122.4	210
Krieg 550 Sniper Rifles	Primary	247.6	380
M249 Machine Gun	Primary	352.0	485
Gas Mask	Equipment	24.2	9
Night-Vision Goggle	Equipment	32.1	12
Tactical Shield	Equipment	42.5	15

(a) Write down the statement of the optimization problem from the given information above. State clearly the objective function and the constraints

Constraints

- 限制式：

  (1) 重量上限為 529：

  $\text{Weight} \leq 529$

  (2) 至少要各帶一件 knife、pistol、equipment：

  $x_1 + x_2 + x_3 > 0$

  $x_4 + x_5 + x_6 > 0$

  $x_{13} + x_{14} + x_{15} > 0$

Objective Function

假設 15 種武器分別為 $x_1, x_2, \ldots, x_{15}$，以 0, 1 表示是否拿取該武器/裝備，並假設額外獲得點數為 bonus。

則可設定攜帶總重量為：

$$\text{Weight} = 3.3x_1 + 3.4x_2 + 6x_3 + 26.1x_4 + 37.6x_5 + 62.5x_6 + 100.2x_7 + 141.1x_8 + 119.2x_9 + 122.4x_{10} + 247.6x_{11} + 352.0x_{12} + 24.2x_{13} + 32.1x_{14} + 42.5x_{15}$$

總生存點數為：

$$\text{SP} = (7x_1 + 8x_2 + 13x_3 + 29x_4 + 48x_5 + 99x_6 + 177x_7 + 231x_8 + 202x_9 + 210x_{10} + 380x_{11} + 485x_{12} + 9x_{13} + 12x_{14} + 15x_{15}) + \text{bonus}$$

- 根據題目額外獲得點數 bonus 條件：
  (1) 當 $x_1 * x_6 = 1$，bonus + 5
  (2) 當 $x_4 * (x_9 + x_{10}) > 0$ 時，bonus + 15
  (3) 當 $(x_8 + x_{11}) * (x_6 * x_{15}) > 0$ 時，bonus + 25
  (4) 當 $x_{13} * x_{14} * x_{15} = 1$，bonus + 70

- 題目目標為使生存點數最大化：max (SP)

(b) Calculate the maximum number of possible combinations of inventory bags.

計算所有的可能性，扣掉不符合限制的及重量小計超過 529 的組合。

程式碼

```
getbinary = lambda x, n: format(x, 'b').zfill(n)

weight_list = [3.3, 3.4, 6.0, 26.1, 37.6, 62.5, 100.2, 141.1, 119.2, 122.4, 247.6, 352,
24.2, 32.1, 42.5]
points_list = [7, 8, 13, 29, 48, 99, 177, 213, 202, 210, 380, 485, 9, 12, 15]
all_set = 2**15
possible_set = []
impossible_set = []
for i in range(all_set):
    w = 0
    p = 0
    two_all = getbinary(i, 15)
    if int(two_all[0]) + int(two_all[1]) + int(two_all[2]) == 0:
        impossible_set.append(two_all)
        continue
    elif int(two_all[3]) + int(two_all[4]) + int(two_all[5]) == 0:
        impossible_set.append(two_all)
        continue
    elif int(two_all[12]) + int(two_all[13]) + int(two_all[14]) == 0:
        impossible_set.append(two_all)
        continue
    else:
        for j in range(15):
```

```
            if two_all[j] == '1':
                w += weight_list[j]
                p += points_list[j]
        if w > 529:
            impossible_set.append(two_all)
        else:
            possible_set.append(two_all)

print('The number of possible combinations:',len(possible_set))
```

程式執行結果

```
The number of possible combinations: 6455
```

(c) Write a program of Genetic Algorithm. We consider the roulette-wheel selection, uniform crossover (crossover probability = 0.1) and multi bit flip mutation (consecutive based on item types) as three operators.

程式碼

```
def point_count(init_set):
    point = 0
    for i in range(len(init_set)):
        if init_set[i] == '1':
            point += points_list[i]
    if init_set[0] == 1 and init_set[5] == 1:
        point += 5
    if init_set[3] == 1 and init_set[8] == 1:
        point += 15
    elif init_set[3] == 1 and init_set[9] == 1:
        point += 15
    if init_set[7] == 1 and init_set[5] == 1 and init_set[14] == 1:
        point += 25
    elif init_set[10] == 1 and init_set[5]  == 1 and init_set[14] == 1:
        point += 25
    if init_set[12] == 1 and init_set[13]  == 1 and init_set[14] == 1:
        point += 70
    return point


def check_possible(new_set):
    new_str = ''.join(new_set)
    #print(type(new_str))
```

```
    if new_str in impossible_set:
        return False
    else:
        return True


def wheel(init_populate):
    point_gene = []
    for i in range(len(init_populate)):
        point_gene.append(point_count(init_populate[i]))
    a = random.randint(1, sum(point_gene))
    point_sum = 0
    k = 0
    while a > point_sum:
        point_sum += point_gene[k]
        k += 1
    #print(k)
    return init_populate[k - 1]

def mutation(father, mother):
    length = 3
    s = random.randint(0, 14)
    for i in range(length):
        if father[(i + s) % 15] == '1':
            father[(i + s) % 15] = '0'
        elif father[(i + s) % 15] == '0':
            father[(i + s) % 15] = '1'
        if mother[(i + s) % 15] == '1':
            mother[(i + s) % 15] = '0'
        elif mother[(i + s) % 15] == '0':
            mother[(i + s) % 15] = '1'
    return father, mother


def cross(father, mother):
    rate = 0.1
    father = list(father)
    mother = list(mother)
    for i in range(len(father)):
        r = random.random()
        if r <= rate:
            c = father[i]
            father[i] = mother[i]
```

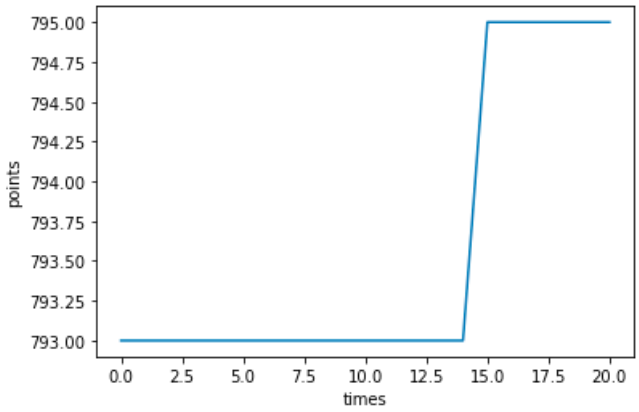
```
            mother[i] = c
    father, mother = mutation(father, mother)
    return father, mother


def compare(init_populate):
    popu_point = []
    for i in range(len(init_populate)):
        popu_point.append(point_count(init_populate[i]))
    for j in range(2):
        a = popu_point.index(min(popu_point))
        popu_point.pop(a)
        init_populate.pop(a)
    max_point = max(popu_point)
    return init_populate, max_point


def gene(steps, size, possible_set):
    #random.seed(49)
    popu_point = []
    max_point_list = []
    x_gene = []
    init_populate = random.choices(possible_set, k = size)
    for i in range(len(init_populate)):
        popu_point.append(point_count(init_populate[i]))
    max_point_list.append(max(popu_point))
    for i in range(steps):
        father = wheel(init_populate)
        init_populate.remove(father) #扣掉已抽的 set 再做第二次 wheel
        mother = wheel(init_populate)
        init_populate.append(father) #把第一次的 set 加回來
        a = False
        while a == False:
            son, daughter = cross(father, mother)
            a = check_possible(son) and check_possible(daughter)
        init_populate.append(son)
        init_populate.append(daughter)
        init_populate, max_point = compare(init_populate)
        max_point_list.append(max_point)
        x_gene.append(i)
    return init_populate, max_point_list, x_gene
```

(d) Use your GA program to optimize your survival points. The population size is set at 10 and the maximum number of iterations is set at 20 steps. No variation convergence criterion is used for termination.

程式碼

```
steps = 20
size = 10
populate, max_point_list, x_gene = gene(steps, size, possible_set)
x_gene.append(20)
#print(x_gene)
#print(max_point_list)
plt.plot(x_gene, max_point_list)
plt.xlabel('times')
plt.ylabel('points')
plt.show()
print(max(max_point_list))
```

程式執行結果



The maximized survival points: 795

(e) Use the hill climbing algorithm and random walk algorithms with the mutation operator defined by your own on this problem. The maximum number of iterations is set at 200 steps.

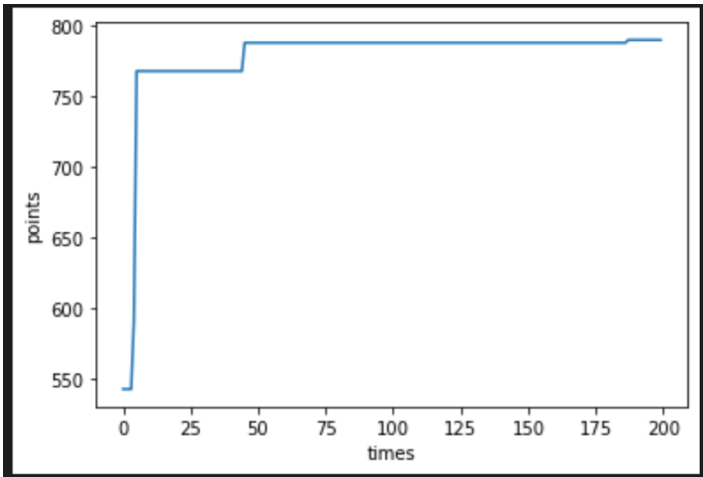
HILL CLIMBING

移動邏輯：隨機決定是否多拿一件或少拿一件裝備，若此次移動代入目標函數計算出來的 survival points 減少，則再移動回原本組合。

```
def mountain_move(ori_point, init_set):
    new_set = list(getbinary(0, 15))
    while check_possible(new_set) == False:
        new_set = init_set
        #print(new_set)
        change_unit = random.randint(0, 14)
```

```
        if new_set[change_unit] == '1':
            new_set[change_unit] = '0'
        elif new_set[change_unit] == '0':
            new_set[change_unit] = '1'
    new_point = point_count(new_set)

    #print(new_point)
    if new_point > ori_point:
        init_set = new_set
        ori_point = new_point
    else:
        new_set = init_set
    return init_set, ori_point


#random.seed(42)
init_num = 1
y_mount = []
x_mount = []
while getbinary(init_num, 15) in impossible_set:
    #print(True)
    init_num = random.randint(1, all_set)
    init_set = getbinary(init_num, 15)
init_set = list(init_set)
ori_point = point_count(init_set)
for i in range(200):
    init_set, ori_point = mountain_move(ori_point, init_set)
    #print(y)
    y_mount.append(ori_point)
    x_mount.append(i)
plt.plot(x_mount, y_mount)
plt.xlabel('times')
plt.ylabel('points')
plt.show()
print("best set = ", init_set, "best survival point = ", ori_point)
#print(new_set)
```

程式執行結果




```
best set =  ['1', '0', '1', '0', '0', '1', '0', '0', '1', '0', '1',
'0', '0', '1', '0'] best survival point =  790
```

With hill climbing method, we can obtain survival points = 790.

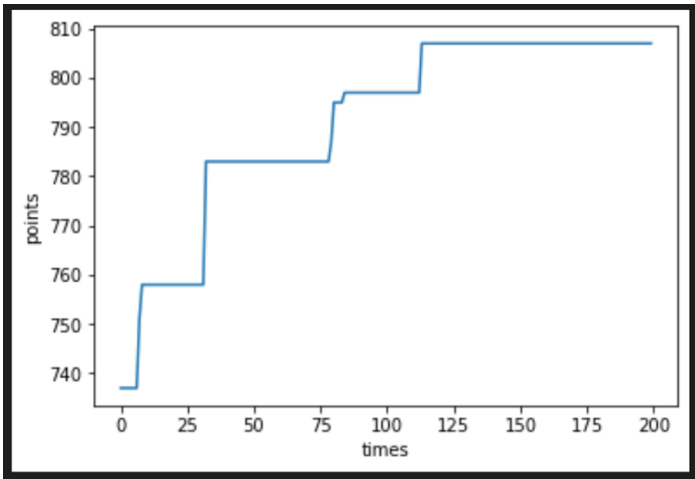
RANDOM WALK

移動邏輯：隨機決定多拿一件或少拿一件裝備，不管 survival points 如何增減，接下來就以新的組合繼續移動。

```python
def walk_move(ori_point, init_set):
    new_set = list(getbinary(0, 15))
    while check_possible(new_set) == False:
        new_set = init_set
        change_unit = random.randint(0, 14)
        if new_set[change_unit] == '1':
            new_set[change_unit] = '0'
        elif new_set[change_unit] == '0':
            new_set[change_unit] = '1'
    new_point = point_count(new_set)
    if new_point > ori_point:
        init_set = new_set
        ori_point = new_point
    return init_set, ori_point


#random.seed(42)
init_num = 1
x_walk = []
y_walk = []
while getbinary(init_num, 15) in impossible_set:
    #print(True)
    init_num = random.randint(1, all_set)
    init_set = getbinary(init_num, 15)
init_set = list(init_set)
```

```python
ori_point = point_count(init_set)
for i in range(200):
    init_set, ori_point = walk_move(ori_point, init_set)
    y_walk.append(ori_point)
    x_walk.append(i)
plt.plot(x_walk, y_walk)
plt.xlabel('times')
plt.ylabel('points')
plt.show()
print("best set = ", init_set, "best survival point = ", ori_point)
```

程式執行結果



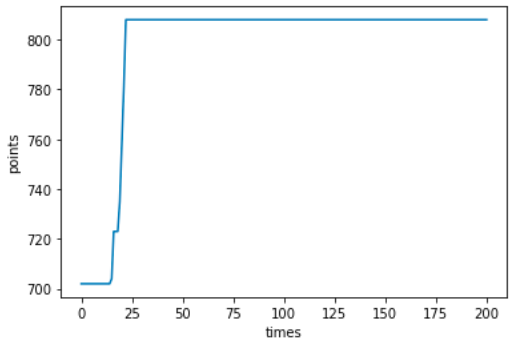
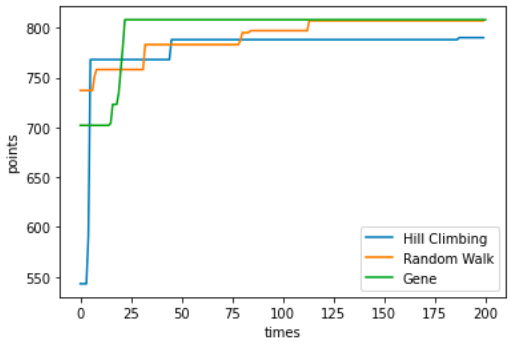

```
best set =  ['1', '0', '1', '1', '1', '0', '1', '0', '1', '1', '0', '0', '0',
'1', '1'] best survival point =  807
```

With random walk method, we can obtain survival points = 807.

(f) Draw the progress diagrams (best objective function value vs number of function evaluation) of GA, Hill Climbing and Random Walk. Comment on how their performances is associated with the choice of their operators.

Python

```python
plt.plot(x_mount, y_mount, label='Hill Climbing')
plt.plot(x_walk, y_walk, label='Random Walk')
plt.plot(x_gene, max_point_list, label='Gene')
plt.xlabel('times')
plt.ylabel('points')
plt.legend()
plt.show()
```

Gene Algorithm	Three Algorithms
	

三種演算法中，Random Walk 及 Gene Algorithm 都可以達到比較高的 Survival Points，而比較 Random Walk 及 Gene Algorithm，Gene Algorithm 在初期的爬升比較快，可以比較快的找到比較好的解，而 Random Walk 雖然爬升速度較慢，但依然可以找到不錯的解。Hill Climbing 相較下，容易受到 initial solution 影響，如果一開始選到的解比較差，就比較容易卡在 local maximum，就不容易找到 global maximum。

3. Optimization of Travel Routes for South Korea Cities. *(50 points)*

Assume you want to organize a travel trip to visit cities in South Korea. It is obvious that you want to minimize the distance travelled. You arrive at and leave from South Korea via Incheon. Here are the 15 cities you plan to visit. Incheon, Seoul, Busan, Daegu, Daejeon, Gwangju, Suwon-si, Ulsan, Jeonju, Cheongju-si, Changwon, Jeju-si, Chuncheon, Hongsung, Muan.

(a) Create the distance of location table.

	Incheon	Seoul	Busan	Daegu	Daejeon	Gwangju	Suwon-si
Seoul	27						
Busan	335	330					
Daegu	244	237	95				
Daejeon	141	144	199	117			
Gwangju	257	268	193	137	137		
Suwon-si	33	31	304	114	114	238	
Ulsan	316	307	54	75	192	222	284
Jeonju	186	195	189	130	61	77	164
Cheongju-si	115	113	221	130	36	173	84
Changwon	304	301	35	72	167	161	274
Jeju-si	439	453	291	324	323	186	423
Chuncheon	102	75	330	236	175	311	91
Hongsung	95	111	271	191	74	162	83
Muan	275	290	233	215	171	44	260

	Ulsan	Jeonju	Cheongju-si	Changwon	Jeju-si	Chuncheon	Hongsung
Jeonju	198						
Cheongju-si	205	96					
Changwon	67	154	190				

Jeju-si	341	263	359	275			
Chuncheon	296	234	139	306	498		
Hongsung	266	97	74	237	344	170	
Muan	265	111	205	202	165	340	180

(b) Calculate how many evaluations of objective function required if one attempts the exhaustive enumeration.

扣除出發點及終點設定為 Incheon，剩下 14 座城市進行窮舉排列，若採用 exhaustive enumeration，則需要經過 14! 次 evaluations。

Incheon	剩餘的 14 座城市...	Incheon

(c) Run a random walk to find the optimal path that passes through these 15 cities with minimum distance. Report the optimal path obtained from the random walk after 100 iterations. Set appropriate values for parameters if needed.

Python

```
np.set_printoptions(linewidth=500, suppress=True)
path = '/Users/wujhejia/Documents/Python/distance.xlsx'
distance_df = pd.read_excel(path, header=0, usecols = "B : P", skiprows = 0)
distance_array = distance_df.to_numpy()
city_dict = {
   0:'Incheon',1:'Seoul',2:'Busan',3:'Daegu',4:'Daejeon'
  ,5:'Gwangju',6:'Suwon-si',7:'Ulsan',8:'Jeonju',9:'Cheongju-si'
  ,10:'Changwon',11:'Jeju-si',12:'Chuncheon',13:'Hongsung',14:'Muan'
}



def random_path(graph):
  N = 14
  path = []
  path.append(0)
  cities_No = list(range(len(graph)))
  # print(len(cities_No)) #14

  for i in range(N):
    randval = random.randint(1, len(cities_No)-1)
    randomCity = cities_No[randval]
    path.append(randomCity)
    cities_No.remove(randomCity)
```

```
  return path


def path_distance(graph, path):
  N = 15
  distance = 0
  for i in range(N):
    distance += graph[path[i-1]][path[i]]
  return distance


def Random_walk():
  iterations = 100
  min_dis = maxsize
  # PLOT: define x_list, y_list
  x_list = []
  y_list = []
  for i in range(iterations):
    path = random_path(distance_array)
    dis = path_distance(distance_array, path)
    if(dis < min_dis):  min_dis = dis

    # PLOT: append i to x_list/ min_dis to y_list
    x_list.append(i+1)
    y_list.append(min_dis)
  return min_dis, x_list, y_list


res_dis, RA_x, RA_y = Random_walk()
print("Random Walk optimal: ", res_dis)
print("RA_x: ", RA_x)
print("RA_y: ", RA_y)
```

程式執行結果

```
✓ def random_path(graph): ···

Random Walk optimal:  2085.0
```

(d) Run a hill climbing to find the optimal path that passes through these 15 cities with minimum distance. Record the distances of the best paths in all 100 iterations. Set appropriate values for parameters if needed.

Python

```
```

```
class HillClimb(object):
    def __init__(self, iter_ceil):
        """
        args:
            n_iteration (int): Number of iterations
            precision (float): Difference of pnew_y and pb_y
            pby_ceil (int): The best pb_y on the record
        """
        self.iter_ceil = iter_ceil

    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 objective(self,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14):
        dist = np.array([distance_array[0,s1],distance_array[s1,s2],distance_array[s2,s3],
                         distance_array[s3,s4],distance_array[s4,s5],distance_array[s5,s6],
                        distance_array[s6,s7],distance_array[s7,s8],distance_array[s8,s9],
                        distance_array[s9,s10],distance_array[s10,s11],distance_array[s11,s12],
                        distance_array[s12,s13],distance_array[s13,s14],distance_array[s14,0]
                    ])
        out = np.sum(dist)
        return out

    def run(self):
        times = 1
        pby_rec = []
        x_list = []
        y_list = []
        s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14 = random.sample(range(1,15),14)
        pb_x = np.array([s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14])
        pb_y = self.objective(pb_x[0],pb_x[1],pb_x[2],pb_x[3],pb_x[4],pb_x[5],pb_x[6],
                         pb_x[7],pb_x[8],pb_x[9],pb_x[10],pb_x[11],pb_x[12],pb_x[13])
        while times <= self.iter_ceil:
            pbx_neighbors = self.getNeighbours(pb_x)
            x_list.append(times)
```

```
            for i in range(len(pbx_neighbors)):
                n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,n13,n14 = pbx_neighbors[i]
                pnew_x = np.array([n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,n13,n14])
                pnew_y =
self.objective(pnew_x[0],pnew_x[1],pnew_x[2],pnew_x[3],pnew_x[4],pnew_x[5],pnew_x[6],

pnew_x[7],pnew_x[8],pnew_x[9],pnew_x[10],pnew_x[11],pnew_x[12],pnew_x[13])
                if pnew_y <= pb_y:
                    pb_x = pnew_x
                    pb_y = pnew_y
                else:
                    pb_x = pb_x
                    pb_y = pb_y

            if times > self.iter_ceil:
                print('Cannot converge. Iteration:', times)
                break
            else:
                times = times+1
                pby_rec.append(pb_y)

        return pb_x, pb_y, x_list, pby_rec

hillclimber = HillClimb(iter_ceil=100)
res_path, res_dis, Hill_x, Hill_y = hillclimber.run()
print("Hill-climbing path: ", res_path)
print("Hill-climbing optimal: ", res_dis, "km")
print("Hill_x: ", Hill_x)
print("Hill_y: ", Hill_y)
```

程式執行結果

```
✓ class HillClimb(object): ···

Hill-climbing path:  [13 14 11  5  8  6  1  9  4  3 10  2  7 12]
Hill-climbing optimal:  1723.0 km
Hill_x:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
Hill_y:  [2695.0, 2311.0, 2062.0, 1859.0, 1736.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0, 1723.0,
1723.0]
```

(e) Write a program of Tabu Search for traveling salesman problem. Set maximum iteration to be 100, maximum length of tabu list to be 10, aspiration criterion as expectation improvement, and we swap during the move. Use a random point as the starting point, run tabu search to find the optimal path that passes through these 15 cities (and return to Incheon at the end) with minimum distance.

Python Code

```
def random_path(graph):
  N = 14
  path = []
  path.append(0)
  cities_No = list(range(len(graph)))
  # print(len(cities_No)) #14

  for i in range(N):
    randval = random.randint(1, len(cities_No)-1)
    randomCity = cities_No[randval]
    path.append(randomCity)
    cities_No.remove(randomCity)

  return path

def path_distance(graph, path):
  N = 15
  distance = 0
  for i in range(N):
    distance = distance + graph[path[i-1],path[i]]

  return distance


def getTabuList(currentPath):
      '''
      Returns a dict of tabu attributes(pair of jobs that are swapped) as keys and [visit_idx,
distance]
      Only record the pair to be swapped, "not really" swapped.
      '''
      dict = {}
      for swap in combinations(currentPath, 2):
        if swap[0] != 0 and swap[1] != 0:
          dict[swap] = {"visit_idx": 0, "distance": 0}
```

```
      return dict

def Swap(currentPath, pair):
  swapped_path = currentPath.copy()
  idx_i = swapped_path.index(pair[0])
  idx_j = swapped_path.index(pair[1])
  swapped_path[idx_i], swapped_path[idx_j] = swapped_path[idx_j], swapped_path[idx_i]
  return swapped_path


def Tabu(graph):
  iterations = 100
  tabu_tenure = 10
  tabu_list = [] # record the swap-pair in tabu list

  # initialize the best point
  bestPath = random_path(distance_array)
  bestDistance = path_distance(distance_array, bestPath)
  # initialize the starting point
  currentPath = random_path(distance_array)
  currentDistance = path_distance(distance_array, currentPath)

  # PLOT: define x_list, y_list
  x_list = []
  y_list = []

  iter = 0
  iter_ = 1
  while iter < iterations:
    # Set tabu list
    tabu_list = getTabuList(currentPath)
    for pair in tabu_list:
      runPath = Swap(currentPath, pair)
      runDistance = path_distance(distance_array, runPath) # ptest.y
      tabu_list[pair]["distance"] = runDistance

    while True:
      # Check acceptable cases
      tabu_best_path = min(tabu_list, key =lambda x: tabu_list[x]["distance"])
      path_dis = tabu_list[tabu_best_path]["distance"] # the minimum distance in all the neighbors.
      visit_idx = tabu_list[tabu_best_path]["visit_idx"]

      if visit_idx < iter_:
```

```
      # Start to move
      currentPath = Swap(currentPath, tabu_best_path)
      currentDistance = path_distance(distance_array, currentPath)

      if path_dis < bestDistance:
        bestPath = currentPath
        bestDistance = currentDistance

      tabu_list[tabu_best_path]["visit_idx"] = tabu_tenure + iter_
      iter += 1
      iter_ += 1

      break

    # Update tabu list (already in tabu list)
    else:
      if path_dis < bestDistance:
        # start to move
        currentPath = Swap(currentPath, tabu_best_path)
        currentDistance = path_distance(currentPath)
        # print("cur Path: ",currentPath)
        bestPath = currentPath
        bestDistance = currentDistance

        iter_ += 1

        break
      else:
        tabu_list[tabu_best_path]["distance"] = maxsize
        continue

   # PLOT: append i to x_list/ bestDistance to y_list
   x_list.append(iter)
   y_list.append(bestDistance)

  return bestPath, bestDistance, x_list, y_list


res_path, res_dis, Tabu_x, Tabu_y = Tabu(distance_array)
print("Tabu Search path: ", res_path)
print("Tabu Search optimal: ", res_dis, "km")
print("Tabu_x: ", Tabu_x)
print("Tabu_y: ", Tabu_y)
```

程式執行結果

```
✓ def random_path(graph): ···

Tabu Search path:  [0, 6, 13, 3, 7, 2, 10, 11, 14, 5, 8, 4, 9, 12, 1]
Tabu Search optimal:  1370.0 km
Tabu_x:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78,
79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
Tabu_y:  [2556.0, 2093.0, 1863.0, 1758.0, 1642.0, 1574.0, 1504.0, 1449.0, 1408.0, 1379.0, 1370.0, 1370.0,
1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0,
1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0,
1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0,
1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0,
1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0,
1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0,
1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0, 1370.0]
```

(f) Write a program of Simulated Annealing for traveling salesman problem. Set appropriate values for parameters if needed. Assume a linear temperature delay. Use a random point as the starting point, run simulated annealing to find the optimal path that passes through these 15 cities (and return to Incheon at the end) with minimum distance.

Python Code

```python
class SimAnn(object):
    def __init__(self, n_iterations):
        """
        args:
            n_iteration (int): Number of iterations
            precision (float): Difference of pnew_y and pb_y
            pby_alltime (int): The best pb_y on the record
        """
        self.n_iterations = n_iterations

    def objective(self, s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14):
        dist = np.array([distance_array[0,s1],distance_array[s1,s2],distance_array[s2,s3],
                    distance_array[s3,s4],distance_array[s4,s5],distance_array[s5,s6],
                    distance_array[s6,s7],distance_array[s7,s8],distance_array[s8,s9],
                    distance_array[s9,s10],distance_array[s10,s11],distance_array[s11,s12],
                    distance_array[s12,s13],distance_array[s13,s14],distance_array[s14,0]
                    ])
        out = np.sum(dist)
        return out

    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, e=1e-30):
        t = 0
        T0 = 100

        s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14 = random.sample(range(1,15),14)
        pcur_x = np.array([s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14])
        pcur_y =
self.objective(pcur_x[0],pcur_x[1],pcur_x[2],pcur_x[3],pcur_x[4],pcur_x[5],pcur_x[6],
                        pcur_x[7],pcur_x[8],pcur_x[9],pcur_x[10],pcur_x[11],pcur_x[12],pcur_x[13])
        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.objective(pnew_x[0],pnew_x[1],pnew_x[2],pnew_x[3],pnew_x[4],pnew_x[5],pnew_x[6],

pnew_x[7],pnew_x[8],pnew_x[9],pnew_x[10],pnew_x[11],pnew_x[12],pnew_x[13])
            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


SimAnneal = SimAnn(n_iterations = 100)


res_x, res_y, SA_x, SA_y = SimAnneal.run()
print("Simulated Annealing path: ", res_x)
print("Simulated Annealing optimal: ", res_y, "km")
print("SA_x: ", SA_x)
print("SA_y: ", SA_y)
```

程式執行結果

```
✓ class SimAnn(object): ···

Simulated Annealing path:  [ 6  4 14  5 11  3  7  2 10  8 13  9 12  1]
Simulated Annealing optimal:  1602.0 km
SA_x:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78,
79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
SA_y:  [2501.0, 2501.0, 2501.0, 2501.0, 2501.0, 2501.0, 2290.0, 2247.0, 2247.0, 2247.0, 2247.0, 2247.0,
2244.0, 2244.0, 2244.0, 2072.0, 1983.0, 1983.0, 1976.0, 1976.0, 1976.0, 1976.0, 1948.0, 1948.0, 1888.0,
1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1888.0,
1888.0, 1888.0, 1888.0, 1888.0, 1888.0, 1763.0, 1763.0, 1763.0, 1763.0, 1763.0, 1763.0, 1763.0, 1763.0,
1763.0, 1763.0, 1763.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0,
1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0,
1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1667.0, 1602.0, 1602.0,
1602.0, 1602.0, 1602.0, 1602.0, 1602.0, 1602.0, 1602.0, 1602.0, 1602.0, 1602.0, 1602.0, 1602.0, 1602.0]
```

(g) Write a program of Ant Colony optimization for traveling salesman problem. Set appropriate values for parameters if needed. Run the ant colony optimization to find the optimal path that passes through these 15 cities (and return to Incheon at the end) with minimum distance

Python Code

```
for i in range(15):
    distance_array[i,i] = np.inf
```

```
class AntColony(object):
    def __init__(self, distances, n_ants, n_best, n_iterations, decay, alpha=1, beta=1):
        """
        Args:
            distances (2D numpy.array): Square matrix of distances. Diagonal is assumed to
be np.inf.
            n_ants (int): Number of ants running per iteration
            n_best (int): Number of best ants who deposit pheromone
            n_iteration (int): Number of iterations
            decay (float): Rate it which pheromone decays. The pheromone value is
multiplied by decay, so 0.95 will lead to decay, 0.5 to much faster decay.
            alpha (int or float): exponenet on pheromone, higher alpha gives pheromone
more weight. Default=1
            beta (int or float): exponent on distance, higher beta give distance more
weight. Default=1
        Example:
            ant_colony = AntColony(distance_array, n_ants=100, n_best=20,
n_iterations=2000, decay=0.95, pby_alltime = 1332)
        """
        self.distances  = distances
        self.pheromone = np.ones(self.distances.shape) / len(distances)
        self.all_inds = range(len(distances))
        self.n_ants = n_ants
        self.n_best = n_best
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta

    def pick_move(self, pheromone, dist, visited):
        pheromone = np.copy(pheromone)
        pheromone[list(visited)] = 0
        row = pheromone ** self.alpha * ((1.0/dist) ** self.beta)
        norm_row = row/row.sum()
        move = np_choice(self.all_inds, 1, p=norm_row)[0]
        return move

    def gen_path(self, start):
        path = []
        visited = set()
        visited.add(start)
        prev = start
```

```
        for i in range(len(self.distances) - 1):
            move = self.pick_move(self.pheromone[prev], distance_array[prev], visited)
            path.append((prev, move))
            prev = move
            visited.add(move)
        path.append((prev, start))
        return path

    def gen_path_dist(self, path):
        total_dist = 0
        for i in path:
            total_dist = total_dist + self.distances[i]
        return total_dist

    def gen_all_paths(self):
        all_paths = []
        for i in range(self.n_ants):
            path = self.gen_path(0)
            all_paths.append((path, self.gen_path_dist(path)))
        return all_paths

    def spread_pheronome(self, all_paths, shortest_path):
        sorted_paths = sorted(all_paths, key=lambda x: x[1])
        for path, dist in sorted_paths[:self.n_best]:
            for move in path:
                self.pheromone[move] = self.pheromone[move] +  (1.0/self.distances[move])

    def main(self):
        shortest_path = []
        all_time_shortest_path = ("placeholder", np.inf)
        x_list = []
        y_list = []
        t = 0
        while t < self.n_iterations:
            all_paths = self.gen_all_paths()
            self.spread_pheronome(all_paths, shortest_path=shortest_path)
            shortest_path = min(all_paths, key=lambda x: x[1])

            if shortest_path[1] < all_time_shortest_path[1]:
                all_time_shortest_path = shortest_path
                t = t+1
                self.pheromone = self.pheromone * self.decay
```

```python
        else:
            t = t+1
            self.pheromone = self.pheromone * self.decay

        # PLOT: append t to x_list/ all_time_shortest_path[1] to y_list
        x_list.append(t)
        y_list.append(all_time_shortest_path[1])

    return all_time_shortest_path[0],  all_time_shortest_path[1], x_list, y_list


antcolony = AntColony(distance_array, n_ants=30, n_best=5, n_iterations=100, decay=0.9)
res_path, res_dis, Ant_x, Ant_y = antcolony.main()
print("Ant Colony path: ", res_path)
print("Ant Colony optimal: ", res_dis, "km")
print("Ant_x: ", Ant_x)
print("Ant_y: ", Ant_y)
```

程式執行結果

```
Ant Colony path:  [(0, 6), (6, 13), (13, 4), (4, 8), (8, 5), (5, 14), (14, 11), (11, 10), (10, 2), (2, 7), (7, 3), (3, 9), (9, 12), (12, 1), (1, 0)]
Ant Colony optimal:  1347.0 km
Ant_x:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
Ant_y:  [1601.0, 1601.0, 1601.0, 1601.0, 1601.0, 1601.0, 1404.0, 1404.0, 1404.0, 1404.0, 1404.0, 1404.0, 1404.0, 1404.0, 1350.0, 1350.0, 1350.0, 1350.0, 1350.0, 1350.0, 1350.0, 1350.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0, 1347.0]
```

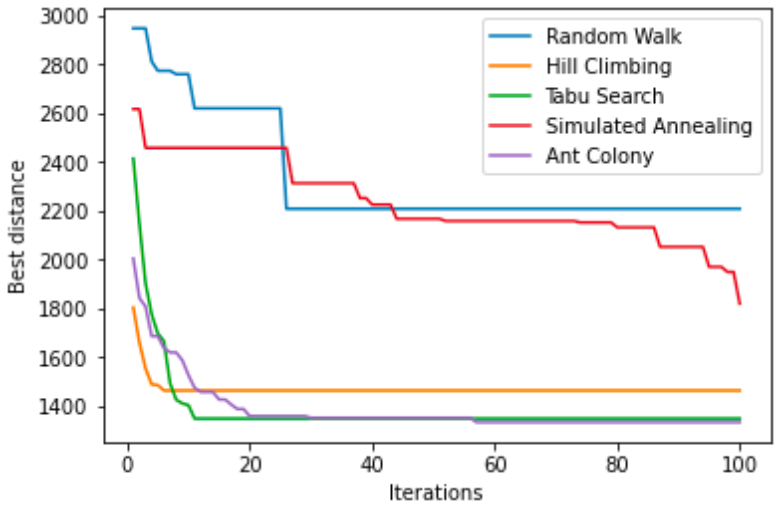
(h) Create the best distance vs iterations plot by plotting the results of Hill Climbing, Random Walk, Simulated Annealing, Tabu Search and Ant Colony Optimization on the same plot. Compare these algorithms and comments on their strength and weakness in this problem.

Python

```python

# Random Walk
RA_dis, RA_x, RA_y = Random_walk()
plt.plot(RA_x, RA_y, label='Random Walk')

# Hill Climbing
```

```
hillclimber = HillClimb(iter_ceil=100)
res_path, res_dis, Hill_x, Hill_y = hillclimber.run()
plt.plot(Hill_x, Hill_y, label='Hill Climbing')

# Tabu Search
res_path, res_dis, Tabu_x, Tabu_y = Tabu(distance_array)
plt.plot(Tabu_x, Tabu_y, label = 'Tabu Search')

# Simulated Annealing
SimAnneal = SimAnn(n_iterations = 100)
res_x, res_y, SA_x, SA_y = SimAnneal.run()
plt.plot(SA_x, SA_y, label = 'Simulated Annealing')

# Ant Colony
#res_path, res_dis, Ant_x, Ant_y = Ant(distance_array)
antcolony = AntColony(distance_array, n_ants=30, n_best=5, n_iterations=100, decay=0.9)
res_path, res_dis, Ant_x, Ant_y = antcolony.main()
plt.plot(Ant_x, Ant_y, label = 'Ant Colony')


plt.ylabel("Best distance")
plt.xlabel("Iterations")
plt.legend()
plt.show()
```

程式執行結果



演算法比較結果

1. 整體來說，在 100 個迭代後，Tabu Search 及 Ant Colony 可以走到最佳解，而兩者再比較的話，Ant Colony 獲得的解會再更好一點。
2. Hill Climbing
   (1) 優勢：演算法非常簡單直觀且快速
   (2) 劣勢：很容易落入 local optimum，如上圖所示，Hill Climbing 在初期很快就找到不錯的解，但就容

易卡在區域的最佳解走不出來，到後期就呈現一條直線，沒有繼續找到最佳解。

3. Random Walk
   (1) 優勢：相對於 Hill Climbing，該演算法不會受到當下所在位置的限制，不會容易卡在 local optimum，
   (2) 劣勢：但整體迭代的效率差很多，要走很久，也需要一點運氣才比較容易走得到最佳解。
4. Simulated Annealing
   (1) 優勢：該演算法克服了 Hill Climbing 容易卡在 local optimum 的弱點，可以有一定的機率能接受 worse solution，所以就比較不會卡在區域最佳解。
   (2) 劣勢：但該演算法的優點同時也是它的缺點，因為有一定的機率容許比較差的解被接受，所以要找到最優解的效率常常會比較慢，如上圖所示，要經過比較多次迭代才會持續下降找到比較好的解。而且該演算法在不同題目受起始設定的溫度的影響滿大，如果一開始的溫度設定地不太合適，也會影響到其找到最佳解的效率。
5. Tabu Search
   (1) 優勢：該演算法找到最佳解的效率可以算是非常優秀，因為它能夠記憶曾經 swap 過的解，避免往重複的方向去找解，有助於提升整體找最佳解的效率。
   (2) 劣勢：雖然它可以記憶曾經找過的方向，但它如果所處的位置不太好，仍然有機會就卡在某個 local optimum 出不來。
6. Ant Colony
   (1) 優勢：該演算法同樣也能夠很迅速地找到最佳解，雖然在迭代的初期效率可能比 Tabu 來得差一點，但經過幾次迭代後，費洛蒙累積的效果會越來越明顯，而且在經過比較多次迭代後，如上圖所示，能夠找到的解，可能還比 Tabu Search 好一點。
   (2) 劣勢：整體迭代的時間比較長，計算速度偏慢，該演算法在初期找解的效率與 Tabu Search 相比可能比較差一點，因為需要經過比較多次迭代，讓螞蟻走過的路徑累積的費洛蒙濃度可以增加的比較明顯，才會逐漸往該方向靠攏。

Hint: Please visit https://www.distancecalculator.net/country/south-korea if you do not know where you can obtain the city distance.

4. **More on Optimization of Travel Routes for South Korea Cities.** *(Bonus 10 points)*
   In fact, Particle Swarm Optimization can also be used to find the optimal path that passes through 15 cities of South Korea and return to Incheon at the end. Write a program of Particle Swarm Optimization and find the optimal path using this program. Compare the algorithm with the previous 5 algorithms and comments on its strength and weakness in this problem.

Python

```python
def routeLength(tsp, x):
    if x.ndim == 1:
        solution = x.copy();
        pathLength = 0
        for i in range(len(solution)):
            pathLength += tsp[solution[i - 1]][solution[i]]
        return pathLength
```

```
    else:
        pop = x.copy()
        obj = np.zeros(pop.shape[0])
        # run over all population and compute the fitness value
        for  j in range(pop.shape[0]):
            solution = pop[j, :]
            pathLength = 0
            for i in range(len(solution)):
                pathLength += tsp[solution[i - 1]][solution[i]]
            obj[j] = pathLength

        return obj


def find_move(best, x_cur):
    # v = best - x_cur
    x = x_cur.copy()
    numofcity = len(best)   # get length
    moveperm = []
    for i in range(numofcity):
        # check if the best[i] equal to x[i]
        idxinx = np.where(x == best[i])[0]

        # if not, swap
        if i != idxinx:
            move = [x[i], x[idxinx][0]]
            moveperm.append(move)
            x[idxinx], x[i] = x[i], x[idxinx]

    return moveperm

def updatepos(x_cur, moves):
    x = x_cur.copy()
    for i in range(len(moves)):
        move = moves[i]
        idx1 = np.where(x == move[0])[0]
        idx2 = np.where(x == move[1])[0]
        x[idx1], x[idx2] = x[idx2], x[idx1]

    return x

def updatemove(w, cr1, cr2, moves_inertia, moves_loc, moves_global, maxv):
    # v = w*moves_inertia + cr1*moves_loc + cr2*moves_global
```

```
    # scalar times velocity
    wint = np.floor(w*len(moves_inertia)).astype(int)
    cr1int = np.floor(cr1*len(moves_loc)).astype(int)
    cr2int = np.floor(cr2*len(moves_global)).astype(int)
    # summation three part of velocity
    moves = moves_inertia[0:wint]
    moves.extend(moves_loc[0:cr1int])
    moves.extend(moves_global[0:cr2int])

    # avoid velocity too large
    if len(moves) > maxv:
        numofdel = len(moves)-maxv
        idx = np.random.permutation(len(moves))
        for i in range(numofdel):
            moves.pop()

    return moves

# max number of step
n = 100

# cognition factor and social factor
c1 = c2 = 1.0 # only can be 1
w = 1.0 # only can be 1
maxv = 15

# determine the population size
pop_size = 20
pop_dim = np.shape(distance_array)[1]


# Initialize x
# x is of size (num of particle)-by-(dim of a point)
x_start = np.zeros((pop_size, pop_dim), dtype=int)
for j in range(pop_size):
    x_start[j, :] = random_path(distance_array)
particle, dim = x_start.shape
# initial velocity by empty list with size of 1-by-pop_size
v = [[] for _ in range(pop_size)]


# locol best & global best
```

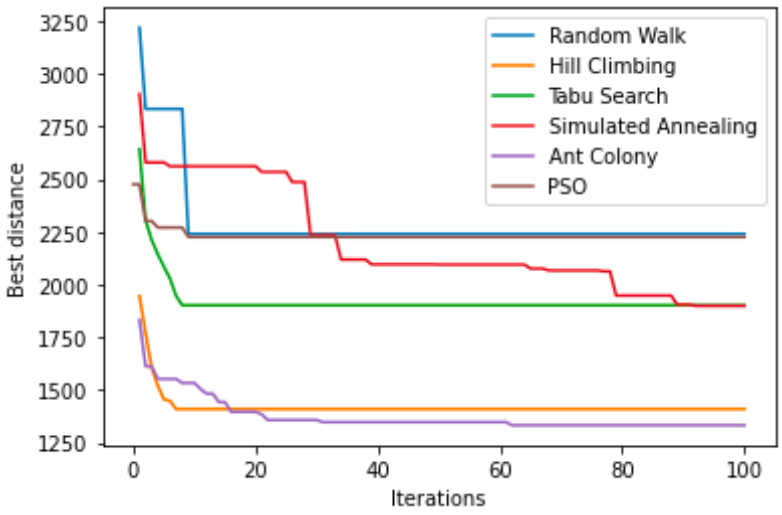
```
x_loc_best = x_start.copy()
loc_best_obj = routeLength(distance_array, x_start)
x_global_best = x_loc_best[loc_best_obj.argmin(), :]
global_best_obj = loc_best_obj.min()

# initial position and velocity
x_cur = x_start.copy()


# record the global best and local current (not best) objective
fb_record = np.zeros(n+1)
fm_record = np.zeros(n+1)
objs = routeLength(distance_array, x_start)
fb_record[0] = np.min(objs)
fm_record[0] = np.mean(objs)


for i in range(n):
    print('Step: ' + str(i) + ' f best: ' + str(fb_record[i]))
    r1, r2 = np.random.rand(2)
    # run over all particles and update the position
    for j, solution in enumerate(x_cur):
        # find velocity and update velocity
        moves_inertia = v[j]
        move_loc = find_move(x_loc_best[j, :], solution)
        move_global = find_move(x_global_best, solution)
        tempv = updatemove(w, c1*r1, c2*r2, moves_inertia, move_loc, move_global, maxv)
        v[j] = tempv
        # update position
        tempx = updatepos(x_cur[j, :], tempv)
        x_cur[j, :] = tempx

    # run over all particles and update the local and global best
    obj = routeLength(distance_array, x_cur)
    for j, solution in enumerate(x_cur):
        # update the local best (min)
        if (obj[j] < loc_best_obj[j]):
            x_loc_best[j, :] = solution.copy()
            loc_best_obj[j] = obj[j].copy()

            # update the global best (min)
            if (obj[j] < global_best_obj):
                x_global_best = solution.copy()
```

```
            global_best_obj = obj[j].copy()


    # check whether converge
    conv = True
    for j in range(1, x_cur.shape[0]):
        if not np.array_equal(x_cur[0, :], x_cur[j, :]):
            conv = False
            break
    if conv:
        print("All particle are at the same position.")
        break

    fb_record[i+1] = global_best_obj
    fm_record[i+1] = np.mean(obj)

print('Best solution: ' + str(x_global_best))
print('Best objective: ' + str(routeLength(distance_array, x_global_best)))
```

程式執行結果

```
Best solution: [ 0  9 10  4 13  6  8  5 14 11  2  3  7 12  1]
Best objective: 1938.0
```

與其他演算法比較



Comments

1. 整體而言，PSO 與前述五種演算法相比，不一定能迅速地找到最佳解，從上圖可以看到，基本上只有開頭能夠順利找到比較短的路徑，接下來跟 Random Walk 演算法一樣就卡在 local minimum。
2. 優勢：在計算的速度上比 Ant Colony 來得快，可以算是比較便於計算的演算法。
3. 劣勢：Traveling Salesman Problem 算是找離散最佳解的問題，而起初 PSO 的設計應該是比較適合應用於找連續型最佳解的問題，所以相較其他最佳化方法，可能相對佔不到優勢。在離散型問題中，也比較容易出現卡在區域最佳解的情況。