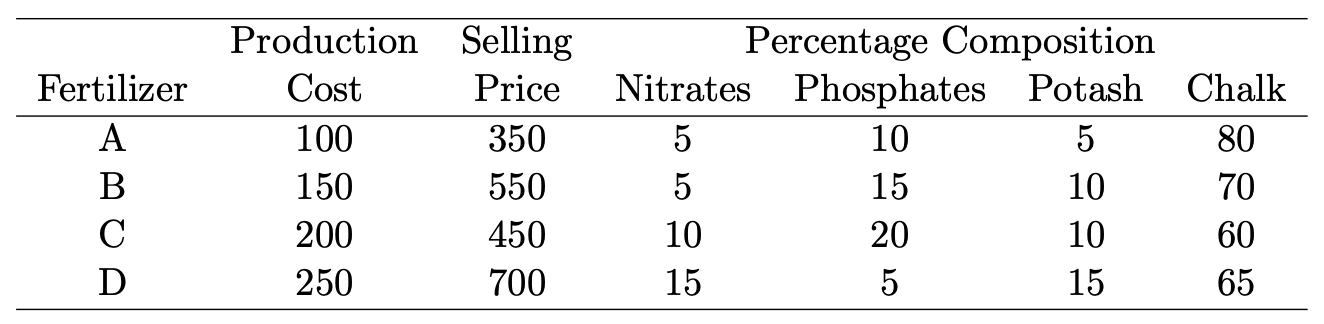
**HOMEWORK 2**

**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)*

****

先假設四種肥料A, B, C, D的生產量為 , , , ，且四種原料進口頓數為 , , , 。

各原料與肥料間的成分比例可整理成以下等式：

|  |
| --- |
|  |

原料進口成本：

|  |
| --- |
|  |

生產成本：

|  |
| --- |
|  |

營業收入：

|  |
| --- |
|  |

目標函數：

|  |
| --- |
|  |

限制式：

|  |
| --- |
|  |

目標函數可以發現僅有肥料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 後，代入限制式：

|  |
| --- |
|  |

在進口條件的限制下最多為3000。

因此目標函數的最大值為

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

****

1. 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： 2. 至少要各帶一件knife、pistol、equipment： |
| Objective Function |
| 假設15種武器分別為 ，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：  總生存點數為：   * 根據題目額外獲得點數bonus條件：  1. 當， 2. 當時， 3. 當時， 4. 當，  * 題目目標為使生存點數最大化： |

1. 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 |
|  |

1. 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 |

1. 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 |

1. 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 |
| 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。 |
| 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)  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 |
| With random walk method, we can obtain survival points = 807. |

1. 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 | |
| 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。

1. **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.

1. 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 |

1. 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 |

1. 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 |
|  |

1. 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 |
|  |

1. 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 |
|  |

1. 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 |
| 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 |
|  |

1. 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  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 |
|  |

1. 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 |
| # 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 3. 優勢：演算法非常簡單直觀且快速 4. 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。 5. Random Walk 6. 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， 7. 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。 8. Simulated Annealing 9. 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。 10. 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。 11. Tabu Search 12. 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。 13. 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。 14. Ant Colony 15. 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。 16. 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.

1. **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 |
| 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 |
|  |
| 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。 |