BỘ GIÁO DỤC VÀ ĐÀO TẠO

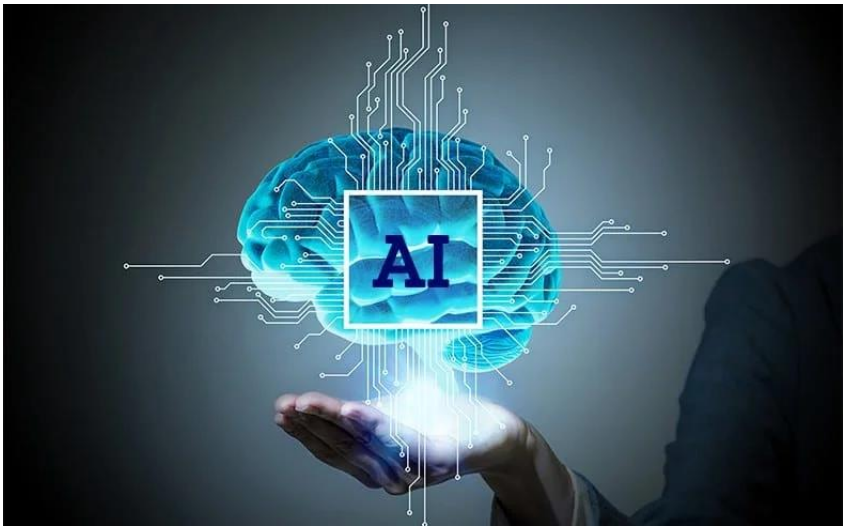
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

KHOA TOÁN - TIN HỌC

-----🕮-----

BÀI TẬP THỰC HÀNH TUẦN 5

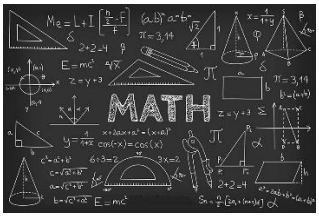


MÔN HỌC: **Nhập môn AI**

Sinh Viên: **Trần Công Hiếu - 21110294**

Lớp: **21TTH_KDL**

TP.HCM, ngày 04 tháng 12 năm 2023



TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TP.HCM
KHOA TOÁN – TIN HỌC

BÀI BÁO CÁO BÀI TẬP THỰC HÀNH TUẦN 5

HK1 - NĂM HỌC: 2023-2024

MÔN: NHẬP MÔN TRÍ TUỆ NHÂN TẠO

SINH VIÊN: TRẦN CÔNG HIẾU

MSSV: 21110294

LỚP: 21TTH_KDL

NHẬN XÉT

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Tp.HCM, Ngày……Tháng……Năm 2021

Giảng viên Bộ môn

MỤC LỤC

I. CÀI ĐẶT VÀ PHÁT HIỆN LỖI.

- Chưa cài đặt thư viện treelib. Cách khắc phục: mở terminal chạy đoạn mã pip install treelib.

- Dư thừa biến times. Trong chương trình, biến times sau khi khởi gán thì không được sử dụng trong xuyên suốt đoạn mã.

```
168 def startTSP(graph,tree,V):
169     goalState = 0
170     times = 0
171     toExpand = TreeNode(0,0,0,0,0)
```

II. TRÌNH BÀY CHI TIẾT.

```
1 from treelib import Node, Tree
2 import sys
```

Từ thư viện treelib ta đã cài đặt trước đó, ta import lần lượt Node và Tree. Node trong treelib đại diện cho một nút trong cây. Mỗi Node có một giá trị và một danh sách con trỏ tới các nút con của nó. Tree là lớp chứa các phương thức để tạo, quản lý và thao tác với cây.

```
4  # Structure to represent tree nodes in the A* expansion
   3 usages
5  class TreeNode(object):
6      def __init__(self, c_no, c_id, f_value, h_value, parent_id):
7          self.c_no = c_no
8          self.c_id = c_id
9          self.f_value = f_value
10         self.h_value = h_value
11         self.parent_id = parent_id
```

Cấu trúc thể hiện các nút của cây trong thuật toán A* mở rộng

Định nghĩa một lớp TreeNode để biểu diễn các nút trong việc mở rộng (expansion) của thuật toán A* trong quá trình tìm kiếm. Đây là một phần của thuật toán A* trong trường hợp đang thao tác với một cây tìm kiếm hoặc một cấu trúc dữ liệu tương tự.

Ta định nghĩa hàm khởi tạo (Constructor function) cho lớp TreeNode để cập nhật lần lượt các thuộc tính của lớp.

“self.c_no = c_no”: Số thứ tự của nút.

“self.c_id = c_no”: Định danh của nút.

“slef.f_value = f_value”: Giá trị gồm tổng chi phí đã di chuyển từ nút gốc đến nút hiện tại và một ước lượng chi phí còn lại để đi từ nút hiện tại đến nút đích. Đây thường là tổng của chi phí thực tế và ước lượng (heuristic cost).

“self.h_value = h_value”: Ước lượng chi phí từ nút hiện tại đến nút đích (heuristic cost).

“self.parent_id = parent_id”: Định danh của nút cha của nút hiện tại trong cây tìm kiếm.

Mỗi nút trong cấu trúc cây tìm kiếm của thuật toán A* được đại diện bởi một thể hiện của lớp TreeNode, với các thuộc tính này lưu trữ thông tin cần thiết để thực hiện việc tìm kiếm thông minh thông qua việc ước lượng chi phí và theo dõi nút cha.

```
13      # Structure to represent fringe nodes in the A* fringe list
        2 usages
14      class FringeNode(object):
15          def __init__(self, c_no, f_value):
16              self.f_value = f_value
17              self.c_no = c_no
```

Cấu trúc thể hiện nút viền trong danh sách viền của thuật toán A*

Định nghĩa một lớp FringeNode để biểu diễn các nút trong danh sách fringe (danh sách viền) của thuật toán A*. Lớp này tương tự như TreeNode trong cấu trúc của danh sách fringe, nhưng chỉ chứa hai thuộc tính là f_value và c_no.

“self.f_value = f_value”: Giá trị gồm tổng chi phí đã di chuyển từ nút gốc đến nút hiện tại và một ước lượng chi phí còn lại để đi từ nút hiện tại đến nút đích.

“self.c_no = c_no”: Số thứ tự của nút trong danh sách fringe.

Ở đây, mỗi FringeNode được đại diện cho một nút trong danh sách fringe của thuật toán A*, nơi mà các nút được sắp xếp theo một tiêu chí nhất định (ở đây là f_value). Điều này giúp thuật toán A* chọn nút tiếp theo để mở rộng dựa trên ước lượng tốt nhất của chi phí từ nút gốc đến nút đích.

```
19  class Graph():
20
21      def __init__(self, vertices):
22          self.V = vertices
23          self.graph = [[0 for column in range(vertices)]
24                        for row in range(vertices)]
25
```

Định nghĩa lớp Graph trong Python để biểu diễn một đồ thị vô hướng có vertices đỉnh. Trong hàm khởi tạo __init__, có hai thuộc tính chính:

“self.V = vertices”: Lưu trữ số lượng đỉnh trong đồ thị.

“self.graph = [[…]…]: Là ma trận kề biểu diễn đồ thị với vertices hàng và vertices cột, ban đầu được khởi tạo với các giá trị 0.

Bên trong câu lệnh trên là 2 vòng lặp for để tạo ra một ma trận 2 chiều (mảng hai chiều) có kích thước vertices x vertices, trong đó mỗi phần tử ban đầu được khởi tạo là 0. Cụ thể, đoạn code sử dụng list comprehension (kỹ thuật tạo list nhanh chóng và ngắn gọn trong Python) để tạo ma trận.

“for row in range(vertices)”: Vòng lặp bên ngoài tạo các hàng của ma trận, mỗi hàng sẽ có vertices phần tử.

“for column in range(vertices)”: Vòng lặp bên trong tạo các phần tử của mỗi hàng, mỗi phần tử đều được khởi tạo bằng 0.

Kết quả là một ma trận vertices x vertices với tất cả các phần tử ban đầu đều có giá trị là 0, để chuẩn bị cho việc lưu trữ thông tin về các cạnh của đồ thị. Nếu self.graph[i][j] khác 0, có một cạnh nối giữa đỉnh i và j, với trọng số hoặc thông tin khác được lưu trữ tại vị trí đó.

```
26      # A utility function to print the constructed MST stored in parent []
        1 usage
27      def printMST(self, parent, d_temp, t):
28          #print("Edge \tWeight")
29          sum_weight = 0
30          min1 = 10000
31          min2 = 10000
32          r_temp = {} #Reverse dictionary
33          for k in d_temp:
34              r_temp[d_temp[k]] = k
```

Phương thức printMST được định nghĩa trong lớp Graph, được sử dụng để in ra Cây khung nhỏ nhất (Minimum Spanning Tree - MST) được lưu trữ trong danh sách parent.

“sum_weight = 0”: Biến này được sử dụng để tính tổng trọng số của cây bao trùm nhỏ nhất.

min1 và min2: Hai biến này được khởi tạo với giá trị lớn (ở đây là 10000) và được sử dụng để tìm hai cạnh nhỏ nhất.

“r_temp = {}”: Đây là một từ điển (dictionary) được sử dụng để tạo một phiên bản ngược (reverse) của từ điển d_temp.

Bên trong vòng lặp “for k in d_temp:” mỗi cặp key-value trong d_temp được đảo ngược và lưu trữ trong r_temp. Ví dụ: d_temp = {0: 1}, thì khi đó d_temp[0] = 1 và r_temp[d_temp[0]] = r_temp[1] = 0. Lúc này, r_temp = {1: 0}.

```
36          for i in range(1, self.V):
37              #print(parent[i], "-", i, "\t", self.graph[i][ parent[i] ])
38              sum_weight = sum_weight + self.graph[i][ parent[i] ]
39              if(graph[0][r_temp[i]] < min1):
40                  min1 = graph[0][r_temp[i]]
41              if(graph[0][r_temp[parent[i]]] < min1):
42                  min1 = graph[0][r_temp[parent[i]]]
43              if (graph[t][r_temp[i]] < min2):
44                  min2 = graph[t][r_temp[i]]
45              if(graph[t][r_temp[parent[i]]] < min2):
46                  min2 = graph[t][r_temp[parent[i]]]
47          return (sum_weight + min1 + min2)%10000
48
```

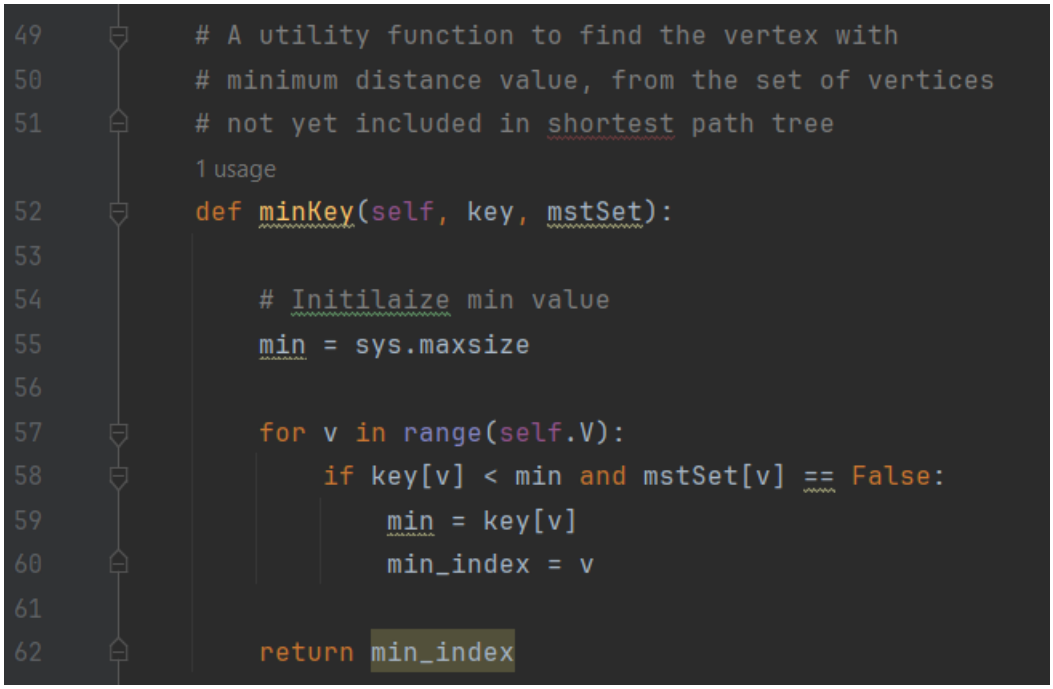
Tiếp đến, vòng lặp “for i in range(1, self.V):” sẽ duyệt qua các đỉnh trong Cây khung nhỏ nhất.

“”sum_weight = sum_weight + self.graph[i][parent[i]]”: Tính tổng trọng số của các cạnh trong MST từ các cạnh đã chọn. self.graph[][] sẽ trả về phần tử thứ i, parent[i] là giá trị trọng số được lưu trước đó.

Trong quá trình tính tổng, code cũng tìm ra hai giá trị nhỏ nhất min1 và min2 trong một số cạnh liên quan đến các đỉnh trong MST.

Đoạn code cuối cùng trả về một giá trị là tổng trọng số của Cây khung nhỏ nhất cộng với min1 và min2, rồi lấy phần dư khi chia cho 10000 (return (sum_weight + min1 + min2)%10000). Điều này đảm bảo rằng nếu

duyệt qua 4 điều kiện if cho min1 và min2 mà một trong số chúng hoặc cả hai đều không thỏa, lúc này tồn tại biến chứa giá trị mặc định ban đầu là 10000, lúc này việc chia lấy dư cho ta biết chỉ lấy phần tổng còn lại.



```
49      # A utility function to find the vertex with
50      # minimum distance value, from the set of vertices
51      # not yet included in shortest path tree
        1 usage
52      def minKey(self, key, mstSet):
53
54          # Initilaize min value
55          min = sys.maxsize
56
57          for v in range(self.V):
58              if key[v] < min and mstSet[v] == False:
59                  min = key[v]
60                  min_index = v
61
62          return min_index
```

Phương thức minKey trên được sử dụng để tìm đỉnh có giá trị khoảng cách nhỏ nhất từ tập hợp các đỉnh chưa được bao gồm trong cây đường đi ngắn nhất.

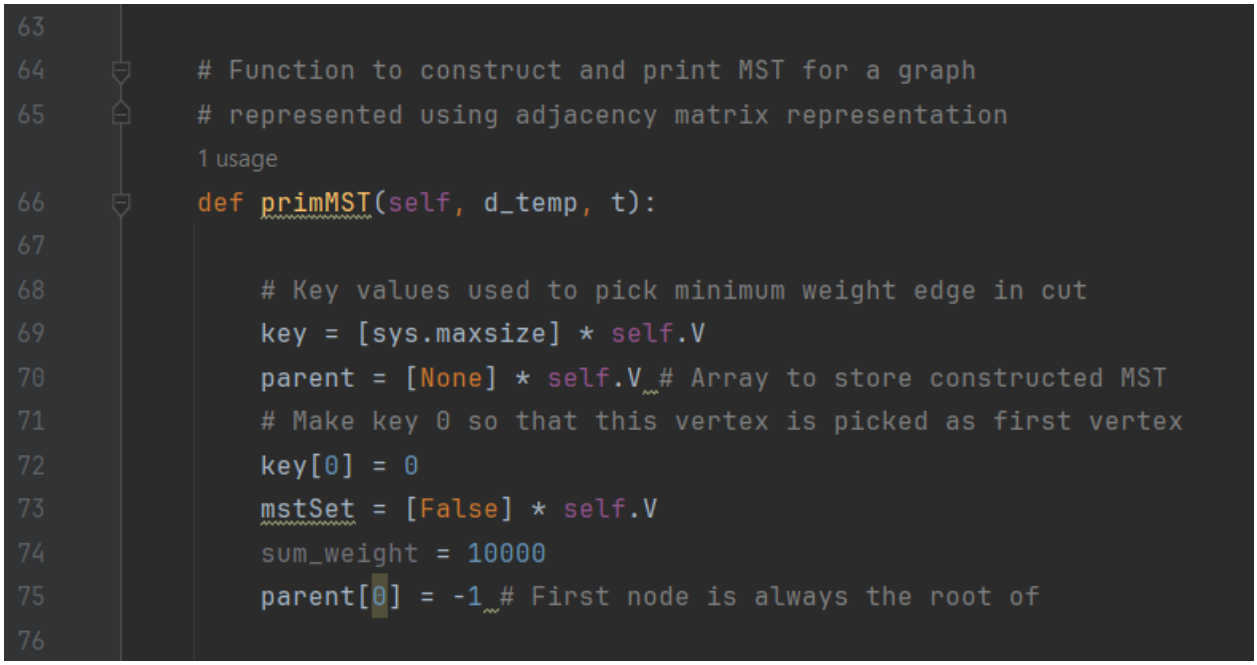
“min = sys.maxsize”: Khởi tạo một giá trị min ban đầu bằng giá trị lớn nhất có thể (sys.maxsize). Biến min sẽ được sử dụng để lưu trữ giá trị khoảng cách nhỏ nhất đã tìm thấy cho đến thời điểm hiện tại.

Duyệt qua tất cả các đỉnh (v) trong đồ thị (có tổng cộng self.V đỉnh).

Kiểm tra nếu giá trị khoảng cách từ đỉnh v (được lưu trong key[v]) nhỏ hơn giá trị hiện tại của min và đỉnh v chưa được thêm vào cây đường đi ngắn nhất (được biểu diễn bởi mstSet[v] == False).

Nếu điều kiện trên thỏa mãn, cập nhật giá trị min và chỉ số min_index cho đỉnh có giá trị khoảng cách nhỏ nhất.

Cuối cùng, hàm trả về chỉ số min_index của đỉnh có khoảng cách nhỏ nhất từ tập hợp các đỉnh chưa được bao gồm trong cây đường đi ngắn nhất.



```
63
64      # Function to construct and print MST for a graph
65      # represented using adjacency matrix representation
        1 usage
66      def primMST(self, d_temp, t):
67
68          # Key values used to pick minimum weight edge in cut
69          key = [sys.maxsize] * self.V
70          parent = [None] * self.V # Array to store constructed MST
71          # Make key 0 so that this vertex is picked as first vertex
72          key[0] = 0
73          mstSet = [False] * self.V
74          sum_weight = 10000
75          parent[0] = -1 # First node is always the root of
76
```

Phương thức primMST là biểu diễn thuật toán Prim để tìm Cây khung nhỏ nhất (Minimum Spanning Tree - MST) trong đồ thị.

Key value được dùng để chọn ra giá trị trọng số nhỏ nhất trong cắt

“key = [sys.maxsize]*self.V”: Mảng key chứa các giá trị khoảng cách từ một đỉnh cụ thể đến Cây khung nhỏ nhất. Ban đầu, tất cả các giá trị khoảng cách được thiết lập là vô cùng lớn (sys.maxsize), trừ đỉnh đầu tiên (key[0] = 0), vì đỉnh đầu tiên sẽ được chọn làm đỉnh gốc của cây bao trùm nhỏ nhất.

“parent = [None]*self.V”: Mảng parent được sử dụng để lưu trữ các đỉnh cha của từng đỉnh trong Cây khung nhỏ nhất. Ban đầu, tất cả các giá trị của parent đều là None, ngoại trừ đỉnh đầu tiên (parent[0] = -1).

Mảng parent này dùng để lưu trữ cấu trúc MST

“mstSet = [False] * self.V”: Mảng mstSet được sử dụng để đánh dấu các đỉnh đã được bao gồm trong cây bao trùm nhỏ nhất (True nếu đã được bao gồm và False nếu chưa).

“sum_weight = 10000”: Biến này ban đầu được khởi tạo là 10000. Có thể sẽ được sử dụng để tính tổng trọng số của cây bao trùm nhỏ nhất.

```
76
77          for c in range(self.V):
78
79              # Pick the minimum distance vertex from the set of vertices not yet processed.
80              # u is always equal to src in first iteration
81              u = self.minKey(key, mstSet)
82
83              # Put the minimum distance vertex in the shortest path tree
84              mstSet[u] = True
85
```

Vòng lặp “for c in range(self.V):” được sử dụng để duyệt qua tất cả các đỉnh của đồ thị (tổng cộng self.V đỉnh).

Mỗi lần lặp, thuật toán Prim sử dụng hàm minKey để chọn ra đỉnh có khoảng cách nhỏ nhất từ tập hợp các đỉnh chưa được bao gồm vào Cây khung nhỏ nhất. Biến u sau đó chứa chỉ số của đỉnh này.

Sau khi đã chọn ra đỉnh có khoảng cách nhỏ nhất, đỉnh đó được đánh dấu là đã xử lý (mstSet[u] = True), tức là đã được bao gồm vào cây bao trùm nhỏ nhất.

```
86              # Update dist value of the adjacent vertices of the picked vertex only if the
87              # current distance is greater than new distance and
88              # the vertex in not in the shotest path tree
89              for v in range(self.V):
90                  # graph[u][v] is non zero only for adjacent vertices of m
91                  # mstSet[v] is false for vertices not yet included in MST
92                  # Update the key only if graph[u][v] is smaller than key[v]
93                  if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
94                      key[v] = self.graph[u][v]
95                      parent[v] = u
96
97          return self.printMST(parent, d_temp, t)
98
```

Tiếp đó, thực hiện việc cập nhật giá trị khoảng cách key của các đỉnh kề v với đỉnh vừa được chọn (đỉnh u). Mục tiêu là cập nhật các giá trị khoảng cách này nếu giá trị khoảng cách hiện tại key lớn hơn giá trị khoảng cách mới (tính từ đỉnh u đến các đỉnh kề v) và đỉnh kề đó chưa được bao gồm vào cây đường đi ngắn nhất.

Vòng lặp for v in range(self.V) duyệt qua tất cả các đỉnh trong đồ thị.

Đối với mỗi đỉnh v, nếu self.graph[u][v] khác 0 (nghĩa là có cạnh nối từ đỉnh u đến v), và đỉnh v chưa được bao gồm vào Cây khung nhỏ nhất (mstSet[v] == False), và giá trị khoảng cách từ đỉnh u đến v (self.graph[u][v]) nhỏ hơn giá trị khoảng cách hiện tại key[v], thì giá trị

khoảng cách key[v] sẽ được cập nhật với giá trị mới self.graph[u][v]. Đồng thời, đỉnh u sẽ được gán là đỉnh cha của đỉnh v trong cây đường đi ngắn nhất (parent[v] = u).

Quá trình này giúp xác định các cạnh có trọng số nhỏ nhất để mở rộng Cây khung nhỏ nhất từ đỉnh đã chọn tới các đỉnh kề chưa được bao gồm, để dần dần xây dựng cây bao trùm nhỏ nhất. Kết quả sẽ được trả về từ hàm dưới dạng một cây bao trùm nhỏ nhất.

```
99  # Idea here is to form a graph of all unvisited nodes and make MST from that.
100 # Determine weight of that mst and connect it with the visited node and 0th node
101 # Prim's Algorithm used for MST (Greedy approach)
    2 usages
102 def heuristic(tree, p_id, t, V, graph):
103     visited = set()                    # Set to store visited nodes
104     visited.add(0)
105     visited.add(t)
106     if(p_id != -1):
107         tnode=tree.get_node(str(p_id))
108         while(tnode.data.c_id != 1):
109             visited.add(tnode.data.c_no)
110             tnode=tree.get_node(str(tnode.data.parent_id))
111     l = len(visited)
```

Định nghĩa hàm heuristic với tham số đầu vào là tree, một id của nút cha (p_id), một nút mục tiêu t, tổng số nút V và một đồ thị graph.

“visited = set()”: Đây là một tập hợp lưu trữ các nút đã được ghé thăm. Ban đầu, tập này bắt đầu với các nút 0 và t.

“if (p_id != -1)”: Điều kiện này kiểm tra xem p_id có khác -1 không. Nếu đúng, chương trình sẽ vào vòng lặp.

“tnode = tree.get_node(str(p_id))”: Lấy một nút từ cấu trúc dữ liệu cây dựa trên biểu diễn chuỗi của p_id.

“while (tnode.data.c_id != 1)”: Vòng lặp này tiếp tục cho đến khi một điều kiện được đáp ứng (c_id của dữ liệu nút không bằng 1).

“visited.add(tnode.data.c_no)”: Thêm giá trị của thuộc tính c_no của nút vào tập visited.

“tnode = tree.get_node(str(tnode.data.parent_id))”: Cập nhật tnode thành nút cha dựa trên thuộc tính parent_id của nút hiện tại.

“l = len(visited)”: Tính độ dài của tập visited.

“num = V – l”: Tính số lượng nút chưa được ghé thăm bằng cách trừ số lượng nút đã ghé thăm từ tổng số nút V.

```
113     if (num != 0 ):
114         g = Graph(num)
115         d_temp = {}
116         key = 0
117         # d_temp dictionary stores mappings of original city no as (key) and
118         #new sequential no as value for MST to work
119         for i in range(V):
120             if(i not in visited):
121                 d_temp[i] = key
122                 key = key +1
123 
124         i = 0
125         for i in range(V):
126             for j in range(V):
127                 if((i not in visited) and (j not in visited)):
128                     g.graph[d_temp[i]][d_temp[j]] = graph[i][j]
129 
130         #print(g.graph)
131         mst_weight = g.primMST(d_temp, t)
132         return mst_weight
133     else:
134         return graph[t][0]
```

“if (num != 0):”: Nếu num khác không, nghĩa là có các nút chưa được ghé thăm.

“g = Graph(num)”: Tạo một đồ thị mới g với số lượng nút chưa được ghé thăm.

“d_temp = {}”: Tạo một từ điển d_temp để ánh xạ số thứ tự mới của các nút chưa được ghé thăm với số thứ tự ban đầu của chúng, nhằm cho việc làm việc với MST. Đồng thời khởi tạo key = 0.

“for i in range(V)”: Duyệt qua tất cả các nút trong đồ thị.

“if(i not in visited)”: Kiểm tra xem nút i đã được ghé thăm chưa.

“d_temp[i] = key”: Gán giá trị key (số thứ tự mới) cho nút i trong từ điển d_temp.

“key = key + 1”: Tăng giá trị của key lên mỗi khi gán một số thứ tự mới cho một nút chưa được ghé thăm.

Kết quả cuối cùng sẽ là một từ điển d_temp trong đó các khóa là các nút chưa được ghé thăm và các giá trị là số thứ tự mới của chúng. Điều này giúp chuyển đổi giữa các chỉ số của các nút chưa được ghé thăm và chỉ số mới của chúng trong quá trình tạo đồ thị mới để tính toán MST.

“i = 0”: Khởi gán giá trị cho i bằng 0.

Chạy vòng lặp for lồng nhau, với các phần tử i, j thuộc tập V. “if((i not in visited) and (j not in visited)):”: Kiểm tra xem cả hai nút i và j đều chưa được ghé thăm. Nếu cả hai nút đều chưa được ghé thăm, chương trình sẽ thực hiện bước tiếp theo.

“g.graph[d_temp[i]][d_temp[j]] = graph[i][j]”: Gán giá trị trọng số từ đồ thị ban đầu (graph) tới đồ thị mới g dựa trên chỉ số mới của nút (d_temp).

Sau khi cập nhật đồ thị g, chương trình tiếp tục bằng việc gọi hàm primMST để tìm Cây khung tối thiểu từ các nút chưa được ghé thăm đến nút đích t. Nếu không có nút nào chưa được ghé thăm (num == 0), chương trình sẽ trả về trọng số của cạnh từ nút t đến nút 0 trong đồ thị ban đầu.

```
136    def checkPath(tree, toExpand, V):
137        tnode=tree.get_node(str(toExpand.c_id)) # Get the node to expand from the tree
138        list1 = list()                          # List to store the path
139        # For 1st node
140        if(tnode.data.c_id == 1):
141            #print("In If")
142            return 0
```

Định nghĩa hàm checkPath, được sử dụng để kiểm tra đường đi trong một cây hoặc đồ thị. Với tham số đầu vào là:

tree: Cấu trúc dữ liệu biểu diễn cây.

toExpand: Là một node cần được mở rộng hoặc xem xét trong cây.

V: Số lượng các nút đồ thị.

“tnode=tree.get_node(str(toExpand.c_id))”: Lấy node trong cây mà ta muốn mở rộng (hoặc kiểm tra). Tiếp theo, một danh sách (list1) được khởi tạo để lưu trữ đường đi.

“if(tnode.data.c_id == 1):”: Kiểm tra xem node đầu tiên (có c_id bằng 1) có phải là node cần xem xét hay không. Nếu đúng, nó trả về 0 (return 0).

```
143     else:
144         #print("In else")
145         depth = tree.depth(tnode)
146         s = set()                           # Check depth of the tree
147         # Go up in the tree using the parent pointer and add all
148         # nodes in the way to the set and list
149         while(tnode.data.c_id != 1):
150             s.add(tnode.data.c_no)
151             list1.append(tnode.data.c_no)
152             tnode=tree.get_node(str(tnode.data.parent_id))
153         list1.append(0)
154         if(depth == V and len(s) == V and list1[0]==0):
155             print("Path complete")
156             list1.reverse()
157             print(list1)
158             return 1
159         else:
160             return 0
161
```

“depth = tree.depth(tnode)”: Xác định độ sâu của nút hiện tại tnode trong cây, tính từ nút gốc.

s = set(): Khởi tạo một set để kiểm tra sự phong phú của đường đi.

Trong vòng lặp while với điều kiện vòng lặp là kiểm tra xem node đầu tiên có phải là node cần xem xét hay không, chương trình tiếp tục thêm các nút trên đường đi từ nút hiện tại tnode đến nút gốc vào set và list1. Việc này được thực hiện bằng cách di chuyển lên cây thông qua con trỏ cha (parent_id) cho đến khi đến nút gốc.

list1.append(0): Thêm nút gốc (0) vào list1.

Điều kiện if kiểm tra xem đường đi đã hoàn thành hay chưa. Nó so sánh độ sâu của cây (depth), giá trị của phần tử đầu tiên trong list1 (mà dự kiến là 0). Nếu đường đi hoàn chỉnh (tất cả các nút từ gốc đến lá đều được truy cập), chương trình sẽ in ra "Path complete", đảo ngược list1 và trả về 1. Ngược lại, nếu đường đi chưa hoàn thành, chương trình trả về 0.

```
def startTSP(graph,tree,V):
    goalState = 0
    times = 0
    toExpand = TreeNode( c_no: 0, c_id: 0, f_value: 0, h_value: 0, parent_id: 0)       # Node to expand
    key = 1                                  # Unique Identifier for a node in the tree
    heu = heuristic(tree,-1, t: 0,V,graph)     # Heuristic for node 0 in the tree
    tree.create_node("1", "1", data=TreeNode( c_no: 0, c_id: 1,heu,heu,-1))# Create 1st node in the tree i.e. 0th city
    fringe_list = {}                        # Fringe List(Dictionary) (FL)
    fringe_list[key] = FringeNode( c_no: 0, heu)      # Adding 1st node in FL
    key = key + 1
```

Định nghĩa hàm startTSP để chạy thuật toán giải quyết bài toán TSP với tham số đầu vào là đồ thị graph, cây tree và số đỉnh V.

goalState được khởi tạo là 0, chỉ định trạng thái mà thuật toán đang cố gắng đạt được.

times được khởi tạo là 0, có thể dùng để theo dõi số lần lặp hoặc một biến đếm nào đó.

toExpand được tạo ra như một đối tượng TreeNode đại diện cho nút cần được mở rộng trong quá trình tìm kiếm.

key được sử dụng làm định danh duy nhất cho các nút trong cây tìm kiếm. Ban đầu nó được đặt là 1.

“heu = heuristic(tree, -1, 0, V, graph)”: Tính toán giá trị heuristic cho nút 0 trong cây. Giá trị heuristic được tính dựa trên các thông số cho trước (tree, -1, 0, V, graph).

“tree.create_node("1", "1", data=TreeNode(0, 1, heu, heu, -1))”: Tạo nút đầu tiên (nút 0) trong cây tìm kiếm với một số dữ liệu khởi tạo.

“fringe_list = {}”: Khởi tạo một từ điển để biểu diễn danh sách Fringe (FL) chứa các nút trong fringe cho quá trình tìm kiếm.

fringe_list[key] = FringeNode(0, heu) thêm nút ban đầu vào danh sách fringe với giá trị heuristic tương ứng.

“key = key + 1”: Tăng giá trị key để sử dụng cho việc xác định nút tiếp theo.

```
    while(goalState == 0):
        minf = sys.maxsize
        # Pick node having min f_value from the fringe list
        for i in fringe_list.keys():
            if(fringe_list[i].f_value < minf):
                toExpand.f_value = fringe_list[i].f_value
                toExpand.c_no = fringe_list[i].c_no
                toExpand.c_id = i
                minf = fringe_list[i].f_value
```

“while (goalState == 0):”: Vòng lặp này sẽ tiếp tục chạy cho đến khi goalState không còn bằng 0.

“minf = sys.maxsize”: Khởi tạo minf với giá trị lớn nhất có thể trong hệ thống, dùng để so sánh và tìm giá trị nhỏ nhất.

Trong vòng lặp for, chương trình duyệt qua các khóa của fringe_list.

“if (fringe_list[i].f_value < minf)”: Nếu giá trị f_value của nút hiện tại trong fringe_list nhỏ hơn minf, chương trình thực hiện các bước sau:

Gán giá trị f_value, c_no, và c_id của nút hiện tại trong fringe_list cho toExpand.

Cập nhật giá trị minf thành giá trị f_value của nút hiện tại, để tiếp tục tìm nút có giá trị f_value nhỏ nhất trong fringe.

```
        h = tree.get_node(str(toExpand.c_id)).data.h_value # Heuristic valueof selected node
        val=toExpand.f_value - h        # g value of selected node
        path = checkPath(tree, toExpand, V) # Check path of selected node if it is complete or not
        # If node to expand is 0 and path is complete, we are done
        # We check node at the time of expansion and not at the time of generation
        if(toExpand.c_no==0 and path==1):
            goalState=1
            cost=toExpand.f_value               # Total actual cost incurred
        else:
            del fringe_list[toExpand.c_id]      # Remove node from FL
            j=0
            # Evaluate f_values and h_values of adjacent nodes of the node to expand
            while(j<V):
                if(j!=toExpand.c_no):
                    h = heuristic(tree, toExpand.c_id, j, V, graph) # Heuristic calc
                    f_val = val + graph[j][toExpand.c_no] + h # g(parent) + g(parent->child) + h(child)
                    fringe_list[key] = FringeNode(j, f_val)
                    tree.create_node(str(toExpand.c_no), str(key),parent=str(toExpand.c_id),\
                                                    data=TreeNode(j, key, f_val, h, toExpand.c_id))
                    key = key + 1
                j=j+1
    return cost
```

“h = tree.get_node(str(toExpand.c_id)).data.h_value”: Lấy giá trị heuristic của nút được chọn để mở rộng từ cây tìm kiếm.

“val = toExpand.f_value – h”: Tính toán giá trị g của nút được chọn. Giá trị g thường được tính bằng cách trừ đi giá trị heuristic từ giá trị f của nút, trong đó f biểu diễn tổng chi phí tính từ nút gốc đến nút hiện tại thông qua nút được chọn.

“path = checkPath(tree, toExpand, V)”: Gọi hàm checkPath để kiểm tra đường đi của nút được chọn xem nó đã hoàn chỉnh chưa.

“if(toExpand.c_no==0 and path==1):”: Kiểm tra xem nút cần mở rộng có phải là nút 0 và đường đi đã hoàn chỉnh hay không. Dòng mã này kiểm tra xem nút cần mở rộng có phải là nút 0 và đường đi đã hoàn chỉnh không. Nếu cả hai điều kiện đều đúng, chương trình đặt giá trị của biến goalState thành 1, đồng thời gán giá trị của chi phí (cost) bằng giá trị f của nút được chọn (toExpand.f_value).

Điều này thường diễn ra khi thuật toán tìm kiếm đã tìm ra một đường đi hoàn chỉnh từ nút 0 (điểm bắt đầu) đến nút 0 (điểm kết thúc) và goalState được sử dụng để chỉ ra rằng mục tiêu đã được đạt được và quá trình tìm kiếm có thể kết thúc. cost được cập nhật để chứa chi phí của đường đi tìm thấy.

Trong trường hợp không phải nút 0 hoặc đường đi chưa hoàn chỉnh:

“del fringe_list[toExpand.c_id]”: Loại bỏ nút đã được mở rộng khỏi danh sách fringe.

Trong vòng lặp while, chương trình duyệt qua các nút kề của nút được mở rộng.

“if(j!=toExpand.c_no)”: Kiểm tra xem j có phải là nút kề với nút hiện tại không.

“h = heuristic(tree, toExpand.c_id, j, V, graph)”: Tính toán giá trị heuristic cho nút kề j.

“f_val = val + graph[j][toExpand.c_no] + h”: Tính giá trị f cho nút kề j. Giá trị f thường là tổng chi phí từ gốc đến nút hiện tại, qua nút kề j và giá trị heuristic của j.

“fringe_list[key] = FringeNode(j, f_val)”: Thêm nút kề mới vào danh sách fringe với giá trị f tương ứng.

“tree.create_node(str(toExpand.c_no), str(key), parent=str(toExpand.c_id), data=TreeNode(j, key, f_val, h, toExpand.c_id))”: Tạo nút kề trong cây tìm kiếm, đồng thời thiết lập mối quan hệ cha-con với nút hiện tại.

“key = key + 1”: Tăng giá trị key để sử dụng cho nút tiếp theo.

Cuối cùng hàm return về chi phí cost.

```
204 ▶ if __name__ == '__main__':
205
206       V=4
207       graph=[[0,5,2,3],[5,0,6,3],[2,6,0,4],[3,3,4,0]]
208
209       tree = Tree()
210       ans = startTSP(graph,tree,V)
211       print("Ans is "+str(ans))
```

“V = 4”: Đây là số lượng đỉnh hoặc thành phố trong bài toán, được đặt là 4.

graph: Ma trận này biểu diễn các khoảng cách giữa các thành phố. Ví dụ, graph[0][1] chứa khoảng cách giữa thành phố 0 và thành phố 1, graph[1][2] chứa khoảng cách giữa thành phố 1 và thành phố 2, và cứ thế.

Một cấu trúc Tree được tạo ra để xây dựng và duy trì cây tìm kiếm trong quá trình tìm kiếm TSP.

Ans = startTSP(graph,tree,V): Đây là việc khởi đầu quá trình tìm kiếm TSP sử dụng hàm startTSP với đồ thị, cây và số lượng đỉnh cung cấp.

sprint("Ans is "+str(ans)): Cuối cùng, kết quả của thuật toán TSP, lưu trong biến ans, được in ra màn hình.

II. TRÌNH BÀY Ý HIỂU.

Bài thực hành đang nói cụ thể về giải quyết bài toán TSP. Sau đó cung cấp một số thuật toán để tìm cây khung nhỏ nhất. Trong đoạn mã cài đặt thì có thuật toán Prim, cho ta cách giải quyết như sau:

- Khởi tạo tập S là cây khung hiện tại, ban đầu S chưa có đỉnh nào.
- Khởi tạo mảng D trong đó D_i là khoảng cách ngắn nhất từ đỉnh i đến 1 đỉnh được kết nạp vào tập S, ban đầu $D[i] = +\infty$.
- Lặp lại các thao tác sau n lần (Với n là số cạnh của đồ thị)

- Tìm đỉnh u không thuộc S có D_u nhỏ nhất, thêm u vào tập S.
- Xét tất cả các đỉnh v kề u, cập nhật $D_v = \min(D_v, w_{u,v})$ với $w_{u,v}$ là trọng số cạnh $u - v$. Nếu D_v được cập nhật theo $w_{u,v}$ thì đánh dấu $trace_v = u$.
- Thêm cạnh $u - trace[u]$ vào tập cạnh thuộc cây khung nhỏ nhất.

Ngoài thuật toán Prim ra, có thuật toán "song song" với Prim chính là thuật toán Kruskal. Khác ở chỗ, nếu như thuật toán Prim lại kết nạp từng đỉnh vào đồ thị theo tiêu chí: đỉnh được nạp vào tiếp theo phải chưa được nạp và gần nhất với các đỉnh đã được nạp vào đồ thị thì thuật toán Kruskal xây dựng cây khung nhỏ nhất bằng cách kết nạp từng cạnh vào đồ thị.

Tiếp đó, bài thực hành cho em biết về Thuật toán mô tả một cách tiếp cận để giải quyết bài toán TSP (Traveling Salesman Problem) sử dụng phương pháp heuristic, cụ thể là Heuristic Chèn Gần Nhất (Nearest Insertion Heuristic) với 5 bước thực thiện được thể hiện chi tiết trong ví dụ. Từ đó, em tìm hiểu thêm được rằng thuật toán này là một phương pháp heuristic để giải quyết bài toán TSP, không đảm bảo tìm ra lời giải tối ưu nhưng thường cho kết quả tốt trong thực tế, đặc biệt là cho các bài toán lớn với số lượng lớn các thành phố.

Có nhiều phương pháp để giải quyết bài toán TSP được đề cập trong bài, liệt kê ra như: phương pháp vét cạn, kĩ thuật ham ăn, thuật toán di truyền, thuật toán tối ưu hóa linh hoạt, thuật toán lập lịch động, phương pháp cắt giảm,…Chúng đều là những thuật toán có thể giải quyết bài toán trên, song mỗi phương pháp đều có ưu nhược riêng biệt. Đối với bài toán TSP với số lượng thành phố lớn, việc kết hợp các phương pháp này hoặc sử dụng các thuật toán heuristics có thể tạo ra các giải pháp gần tối ưu một cách hiệu quả.

Có thể thấy bài thực hành không chỉ dừng lại ở việc hiểu, biết những gì có trong file thực hành, mà từ những gợi mở ấy giúp em tự khám phá, tìm tòi ra những điều hay về chủ đề bài thực hành. Em xin cảm ơn cô rất nhiều.

IV. NHẬN XÉT.

Từ list graph trong đoạn mã cung cấp. Vì đồ thị khá đơn giản nên ta có thể chạy tay và so sánh kiểm chứng với kết quả của đoạn mã. Ta thu được

đường đi là từ $0 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0$ với tổng độ dài các cạnh trong cây khung nhỏ nhất là 14.