21110324 – Lương Đăng Khôi

- Chương trình cài đặt và thực thi không chứa lỗi nào, trả kết quả như hình bên dưới.

```
D:\Documents\AI>python -u "d:\Documents\AI\BTTH_NMTTNT_Tuan5_21110324.py"
Path Complete
[0, 2, 3, 1, 0]
Ans is 14
```

- Thuật toán khia báo các thư viện là sys và Node, Tree trong treelib tuy nhiên trong code ta thấy duy nhất Node trong treelib không được sử dụng đến, điều này rất dễ thấy khi IDE hoặc text editor thông báo như bên dưới.

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

- Chương trình bắt đầu với lớp 'TreeNode' là đại diện cho các node trong quá trình mở rộng thuật toán A*. Mỗi node có các thuộc tính như 'c_no" là số thành phố, 'c_id' là định danh thành phố và mang ràng buộc là tính duy nhất, 'f_value' là chi phí tích luỹ và 'parent_id' là định danh của node cha và định danh node cha có thể không mang ràng buộc duy nhất vì một node có thể dẫn đến nhiều node, tức các node có thể có cùng node cha.

```
class TreeNode(object):
    def __init__(self, c_no, c_id, f_value, h_value,
parent_id):
        self.c_no = c_no
        self.c_id = c_id
        self.f_value = f_value
        self.h_value = h_value
        self.parent_id = parent_id
```

- Lớp tiếp theo là lớp 'FringeNode' đại diện cho các node trong danh sách node thuật toán A*. Mỗi node có các thuộc tính: 'c_no' là số thành phố và 'f_value' là chi phí tích luỹ cộng với giá trị heuristic

```
class FringeNode(object):
    def __init__(self, c_no, f_value):
        self.f_value = f_value
        self.c_no = c_no
```

- Lớp 'Graph' được sử dụng để biểu diện đồ thị vị trí và các mối liên kết giữa các thành phố. Lớp này bao gồm các phương thức để tìm MST bằng thuật toán Prim.

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)] for
row in range(vertices)]

    # A utility function to print the constructed MST stored
in parent[]
```

```
    def printMST(self, parent, d_temp, t):
        # print("Edge \tWeight")
        sum_weight = 0
        min1 = 10000
        min2 = 10000
        r_temp = {} # Reverse dictionary
        for k in d_temp:
            r_temp[d_temp[k]] = k

        for i in range(1, self.V):
            # print(parent[i], "-", i, "\t",
self.graph[i][parent[i]])
            sum_weight = sum_weight + self.graph[i][parent[i]]
            if graph[0][r_temp[i]] < min1:
                min1 = graph[0][r_temp[i]]
            if graph[0][r_temp[parent[i]]] < min1:
                min1 = graph[0][r_temp[parent[i]]]
            if graph[t][r_temp[i]] < min2:
                min2 = graph[t][r_temp[i]]
            if graph[t][r_temp[parent[i]]] < min2:
                min2 = graph[t][r_temp[parent[i]]]

        return (sum_weight + min1 + min2) % 10000

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):
        # Initilize min value
        min = sys.maxsize

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    # Function to construct and print MST for a graph
    # represented using adjacency matrix representation
    def primMST(self, d_temp, t):
        # Key values used to pick minimum weight edge in cut
```

```
        key = [sys.maxsize] * self.V
        parent = [None] * self.V  # Array to store constructed
MST
        # Make key 0 so that this vertex is picked as first
vertex
        key[0] = 0
        mstSet = [False] * self.V
        sum_weight = 10000
        parent[0] = -1  # First node is always the root of

        for c in range(self.V):

            # Pick the minimum distance vertex from the set of
vertices not yet processed.
            # u is always equal to scr in first iteration
            u = self.minKey(key, mstSet)

            # Put the minimum distance vertex in the shortest
path tree
            mstSet[u] = True

            # Update dist value of the adjacent vertices of
the picked vertex only if the
            # current distanceis greater than new distance and
            # the vertex in not in the shortest path tree
            for v in range(self.V):
                # graph[u][v] is non zero inly for adjacent
vertices of m
                # mstSet[v] is false for vertices not yet
included in MST
                # Update the key only if graph[u][v] is
smaller than key[v]
                if self.graph[u][v] > 0 and mstSet[v] == False
and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        return self.printMST(parent, d_temp, t)
```

- Phương thức đầu tiên là '__init__' dùng để khởi tạo một đối tượng 'Graph' nhận vào số đỉnh của đồ thị ('vertices'). Sau đó thiết lập số đỉnh 'V' và khởi tạo ma trận vuông với cấp nxn với n = tham số 'vertices' được truyền vào.

- Phương thức kế tiếp là ‘printMST’ với tham số đầu vào là parent, d_temp, t. Phương thức này được gọi để in ra MST được xây dựng bởi thuật toán Prim. Quay lại với các tham số đầu vào ta có tham số parent là list lưu trữ các node cha của các node trong MST, tham số ‘g’ là đồ thị đầu vào, tham số ‘d_temp’ là một dang cấu trúc dữ liệu dictonary để lưu trữ ánh xạ giữa số thành phố với các số thứ tự trong MST, và tham số cuối cùng ‘t’ là thành phố đích.

- Phương thức ‘minKey’ là một phương thức tiện tích để tìm đỉnh có giá trị khoảng cách nhỏ nhất mà vẫn chưa được thêm vào MST. Phương thức yêu cầu các tham số truyền vào bao gồm ‘key’ là một list lưu trữ các giá trị khoảng cách nhỏ nhất từ đỉnh hiện tại đến tất cả các đỉnh khác, tham số mstSet là một biến boolean để kiểm tra xem một đỉnh đã được thêm vào cây hay chưa.

- Phương thức cuối cùng là phương thức ‘primMST’, đây là phương thức thực hiện thuật toán Prim để xây dựng MST với ‘g’ là tham số đồ thị đầu vào, ‘d_temp’ là một cấu trúc dữ liệu dictionary lưu trữ ánh xạ giữa số thành phố gốc và số thứ tự trong MST, tham số ‘t’ là thành phố đích.

```
def heuristic(tree, p_id, t, V, graph):
    visited = set()  # Set to store visited nodes
    visited.add(0)
    visited.add(t)
    if p_id != -1:
        tnode = tree.get_node(str(p_id))
        # Find all visited nodes and add them to the set
        while tnode.data.c_id != 1:
            visited.add(tnode.data.c_no)
            tnode = tree.get_node(str(tnode.data.parent_id))
    l = len(visited)
    num = V - l  # No of unvisited nodes
    if num != 0:
        g = Graph(num)
        d_temp = {}
        key = 0
        # d_temp dictionary stores mappings of original city
no as (key) and
        # new sequential no as value for MST to work
        for i in range(V):
            if i not in visited:
                d_temp[i] = key
                key = key + 1

        i = 0
        for i in range(V):
            for j in range(V):
                if (i not in visited) and (j not in visited):
```

```
                    g.graph[d_temp[i]][d_temp[j]] =
graph[i][j]

        # print(g.graph)
        mst_weight = g.primMST(d_temp, t)
        return mst_weight
    else:
        return graph[t][0]
```

- Hàm 'heuristic' là hàm có nhiệm vụ tính giá trị heuristic cho một trạng thái trong bài toán TSP. Hàm này sử dụng thuật toán Prim bằng cách gọi phương thức 'primMST' trong lớp 'Graph' để tính trọng số của MST của các thành phố chưa đến, sau đó nối các thành phố đã đến và thành phố 0.

+ Đối số của hàm là: **'tree'** là cây tìm kiếm – một đối tượng từ thư viện treelib chứa thông tin về các node và mốt quan hệ giữa chúng. **'p_id'** là định danh của node cha trong cây. **'t'** là thành phố đích. **'V'** là số lượng thành phố. **'graph'** là ma trận trọng số của đồ thị.

- Các bước của hàm **'heuristic'**:

+ **Khởi tạo:** Tạo một tập hợp ('visited') để lưu trữ các thành phố đã đến.Thêm thành phố 0 và thành phố đích ('t') vào tập hợp này. Nếu có một node cha (được xác định bởi 'p_id'), thực hiện việc thêm các thành phố đã thăm từ node cha đến node gốc vào tập hợp 'visited'.

+ **Tính số thành phố chưa đến:** Tính số lượng thành phố chưa đến bằng cách trừ số thành phố đã đến từ tổng số thành phố ('V').

+ **Tạo đồ thị con chưa đến:** Nếu có ít nhất một thành phố chưa thăm, tạo một đồ thị con chỉ chứa các thành phố chưa đến bằng cách sao chép các trọng số từ ma trận trọng số ban đầu ('graph').

+ **Tìm MST:** Sử dụng thuật toán Prim trên đồ thị con chưa đến để tìm MST. Phương thức 'primMST' trong lớp 'Graph' được sử dụng cho mục đích này.

+ **Nối cây với thành phố đã đến và thành phố 0:** Tính tổng tổng trọng số của MST. Trong trường hợp có ít nhất một thành phố chưa đến, tính toán trọng số giữa thành phố đã đến, thành phố, thành phố 0 và thành phố đích. Thêm giá trị này vào tổng trọng số của cây.

+ **Trả về giá trị heuristic:** Trả về giá trị heuristic, là tổng số trọng số đã tính.

- Hàm **'heuristic'** này giúp ước lượng giá trị heuristic cho một trạng thái cụ thể trong quá trình tìm kiếm A*. Giá trị heuristic này được sử dụng để ưu tiên các node trong hàng đợi ưu tiên A* dựa trên kỳ vọng chi phí đến mục tiêu.

```
def checkPath(tree, toExpand, V):
    tnode = tree.get_node(str(toExpand.c_id))  # Get the node
to expand from the tree
    list1 = list()  # List to store the path
    # For 1st node
    if tnode.data.c_id == 1:
        # print("In if")
        return 0
    else:
        # print("In else")
        depth = tree.depth(tnode)  # Check depth of the tree
        s = set()  # Set to store nodes in the path
```

```
        # Go up in the tree using the parent pointer and add
all
        # node in the way tho the set and list
        while tnode.data.c_id != 1:
            s.add(tnode.data.c_no)
            list1.append(tnode.data.c_no)
            tnode = tree.get_node(str(tnode.data.parent_id))
        list1.append(0)
        if depth == V and len(s) == V and list1[0] == 0:
            print("Path Complete")
            list1.reverse()
            print(list1)
            return 1
        else:
            return 0
```

- Hàm **'checkPath'** có nhiệm vụ kiểm tra đường đi từ một node cụ thể đến node gốc của cây có hoàn chỉnh hay không. Hàm này được sử dụng trong quá trình tìm kiếm bằng thuật toán A* để đảm bảo rằng chỉ có các đường đihoàn chỉnh (chứa tất cả các thành phố) mới được xem xét.

+ Các đối số của hàm: **'tree'** là cây tìm kiếm – một đối tượng từ thư viện treelib chứa thông tin về các node và mối quan hệ giữa chúng. **'toExpand'** là nốt cần kiểm tra – một đối tượng của lớp **'TreeNode'** chứa thông tin về các node hiện tại cần mở rộng. **'V'** là số lượng thành phố hay số lượng các node

- Các bước của hàm **'checkPath'**:

+ **Khởi tạo:** tìm node trong cây tương ứng với node cần kiểm tra ('toExpand').

+ **Kiểm tra đường đi:** Nếu node vần kiểm tra ('toExpand') là node gốc (có 'c_id' bằng 1) và không có đỉnh nào được visited (được biểu diễn bởi biến 'depth' và 's'), tức là đường đi đã hoàn thành. Trong trường hợp đường đi hoàn chỉnh, nó sẽ in ra thông báo "Path complete" và trả về 1.

+ **Lưu trữ đường đi:** Nếu đường đi chưa hoàn chỉnh, hàm này sẽ duyệt lên từ node cần kiểm tra đến node gốc và thêm các thành phố đã đi qua vào một tập hợp ('s') và một danh sách ('list1').

+ **Kiểm tra độ dài đường đi và đầy đủ thành phố:** Kiểm tra xem độ sâu của node cần kiểm tra có bằng với số lượng thành phố (điều này được kiểm tra bởi biến 'depth') và tập hợp 's' có bao gồm tất cả các thành phố hay không. Nếu điều này đúng và đường đi bắt đầu từ thành phố 0, hàm sẽ in ra thông báo "Path complete" và trả về 1.

+ **Trả về kết quả:** Nếu lộ trình không hoàn chỉnh, hàm trả về 0 để tiếp tục tìm kiếm.

- Hàm **'checkPath'** giúp đảm bảo rằng chỉ những lộ trình hoàn chỉnh mới được xem xét trong quá trình tìm kiếm A*, giúp tối ưu hoá thuật toán và tránh việc xem xét các đường đi không cần thiết.

```
def startTSP(graph, tree, V):
    goalState = 0
    times = 0
    toExpand = TreeNode(0, 0, 0, 0, 0)  # Node to expand
    key = 1  # Unique Identifier for a node in the tree
```

```
    heu = heuristic(tree, -1, 0, V, graph)  # Heuristic for
node 0 in the tree
    tree.create_node(
        "1", "1", data=TreeNode(0, 1, heu, heu, -1)
    )  # Create 1st node in the tree i.e. 0th city
    fringe_list = {}
    fringe_list[key] = FringeNode(0, heu)  # Adding 1st node
in FL
    key = key + 1
    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

        h = tree.get_node(str(toExpand.c_id)).data.h_value  #
Heuristic value of 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 ) and path is complete, we are
done
        # We check node at the time of expansion and not at
the time od generation
        if toExpand.c_no == 0 and path == 1:
            goalState = 1
            cost = toExpand.f_value  # Total actual cost
incured
        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
```

- Hàm **'startTSP'** là hàm chính để khởi động quá trình tìm kiếm trong bài toán TSP bằng thuật toán A*.

+ Các đối của hàm: **'graph'** là ma trận trọng số của đồ thị. **'tree'** là cây tìm kiếm - một đối tượng từ thư viện treelib chứa thông tin về các nút và mối quan hệ giữa chúng. **'V'** là số lượng thành phố, hay số lượng các node.

+ **Khởi tạo:** Tạo một node 'toExpand' để theo dõi node hiện tại đang được mở rộng. Khởi tạo một số biến như 'key' để định danh cho các node trong cây, 'heu' để tính giá trị heuristic của node gốc, và cây 'tree' đã được tạo với node gốc ở thành phố 0.

+ **Vòng lặp hoạt động đến khi đặt được đích:** Trong mỗi lần lặp sẽ chọn node 'toExpand' có giá trị 'f_value' (chi phí tốt nhất tính đến thời điểm đó), nếu 'toExpand'là thành phố 0 và đường đi hoàn chỉnh thì dừng quá trình tìm kiếm và trả về chi phí tối thiểu.

+ **Loại bỏ node khỏi fringe:** Xoá node 'toExpand' khỏi danh sách fringe (danh sách các node chờ xem xét).

+ **Mở rộng các node con:** Duyệt qua tất cả các thành phố có thể đến từ thành phố hiện tại (trừ chính thành phố hiện tại) và tạo các node con. Cập nhật danh sách fringe với các node con mới được tạo.

+ **Kiểm tra điều kiện dừng:** Kiểm tra xem node 'toExpand'có phải là thành phố 0 và đường đi hoàn chỉnh không. Nếu là đường đi hoàn chỉnh, in ra thông báo và trả về chi phí tối thiểu.

+ **Trả về kết quả:** Nếu đến đây mà vẫn chưa tìm thấy đường đi hoàn chỉnh thì sẽ tiếp tục còng lặp cho đến khi đạt điều kiện dừng.

- Hàm **'startTSP'** là trung tâm của thuật toán A* trong bài toán TSP. Nó tìm kiếm qua các trạng thái (các thành phố) và mở rộng các node có chi phí thấp nhất. Các giá trị **'f_value'** của các node được tính đựa trên giá trị **'g_value'** (chi phí tính dến thời điểm đó) và giá trị heuristic để ước lượng chi phí còn lại.

***Nhận xét:** Thuật toán để giải quyết bài toán TSP này khá phức tạp tuy nhiên code đã cho không bị lỗi và đọc các dòng note trong code ta cũng hiểu phần nào về cách hoạt động và logic của chương trình.