Table of Contents

1. Cài đặt chương trình và thực thi

Chương trình được cài đặt và thực thi trơn tru, không có lỗi cú pháp hay lỗi nào khiến nó không chạy được. Kết quả chạy ra là:

```
(base) PS F:\CODE\HCMUS\Intro_AI\BTTH_TUAN5> python .\BTTH_TUAN5.py
Path complete
[0, 2, 3, 1, 0]
Ans is 14
```

2. Trình bày những gì đã hiểu được trong bài thực hành

Bước vào chương trình là khai báo các class mà sử dụng, đầu tiên với class TreeNode. Mỗi một node đại diện cho một thành phố có các thuộc tính như `c_no` là số thành phố, `c_id` là mã id của thành phố và id này là khóa chính, `f_value` là chi phí tích lũy và `parent_id` là id của thành phố mà từ thành phố đó đến node hiện tại và cũng có ràng buộc là chỉ có 1 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
```

Tiếp theo là class FringeNode đại diện cho các node trong hàng đợi *fringe* của thuật toán A-star. Mỗi node có thuộc tính là `c_no` là số thành phố và `f_value` là chi phí tích lũy cộng với giá trị heuristic.

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

Tiếp theo là class Graph được sử dụng để biểu diễn đồ thị và các mối liên kết mà trong đó các đỉnh là các thành phố, trong đó bao gồm phương thức MST sử dụng thuật toán *Prim*.

Trong đó có các phương thức cơ bản như `__init__` là để khởi tạo đồ thị nhận vào số đỉnh V và sau đó khởi tạo ma trận vuông $n \times n$, với n là số đỉnh.

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

Phương thức `printMST` nhận vào tham số là `parent`, `d_temp`, `t`. Nó sẽ in ra MST được xây dựng bởi thuật toán *Prim*. Nói về các tham số:

- Tham số `parent` là một danh sách lưu trữ các node cha của các node trong MST, trong hàm có graph là đồ thị hiện tại.
- Tham số `d_temp` là dictionary lưu trữ quan hệ giữa số thành phố và số thứ tự trong MST.
- Tham số `t` là thành phố đích đến.

```python
# A utility function to print the constructed MST stored in parent[]
def printMST(self, parent, d_temp, t):
    sum_weight = 0
    min1 = 0
    min2 = 10000
    r_temp = {} # Reverse dictionary
    for k in d_temp:
        r_temp[d_temp[k]] = k

    for i in range(1, self.V):
        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
```

Phương thức `minKey` dùng để tìm đỉnh có giá trị khoảng cách nhỏ nhất mà chưa được thêm vào MST. Các tham số tham gia là `key` và `mstSet` trong đó các tham số có ý nghĩa:

- Tham số `key` là danh sách lưu trữ các giá trị khoảng cách nhỏ nhất từ đỉnh hiện tại tới tất cả các đỉnh khác.
- Tham số `mstSet` là biến True, False để kiểm tra xem một đỉnh đã được thêm vào cây hay chưa.

```python
# 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):

    # Initialize 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

```

Phương thức `primMST` là phương thức chính để thực hiện thuật toán *Prim* để xây dựng lên MST với các tham số đầu vào là:

- Tham số `d_temp` là dictionary tương tự phương thức ở trên.
- Tham số `t` là thành phố đích.

```python
# 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 src 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 distance is greater than new distance and
            # the vertex is not in the shortest path tree
            for v in range(self.V):
                # graph[u][v] is non zero only 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)
```

Vậy là đã xong class Graph, tiếp theo là hàm tính toán `heuristic`, các tham số đầu vào là:

- Tham số `tree` là cây mà chứa các TreeNode đã được mở rộng trong thuật toán A-star.
- Tham số `p_id` là id của node cha.
- Tham số `t` là nút đích.
- Tham số `v` là số đỉnh trong đồ thị.
- Tham số `graph` là đồ thị đang xét.
  Các bước thực hiện:
- Đầu tiên khởi tạo tập `visited` để theo dõi các node đã được thăm trong `tree`, và trong quá trình tìm kiếm, ban đầu được khởi tạo có chứa node bắt đầu là 0 và node kết thúc là `t`.
- Nếu `p_id` khác -1, nghĩa là có một node cha , ta sẽ tiếp tục theo dõi các node đã được thăm trong cây từ node cha trở về. Duyệt qua hết các node trong cây, sẽ lấy node cha từ cây và lặp qua các node từ node cha về gốc, thêm nó vào `visited`.

- Tính toán số lượng node chưa được thăm là biến `num` là số đỉnh trừ đi số nút trong `visited`.
- Tạo đồ thị con chỉ chứa các node chưa được thăm, sau đó tạo dictionary `d_temp` phục vụ cho thuật toán MST.
- Cập nhật ma trận kề cho đồ thị con.
- Tính toán trọng số bao trùm cây tối thiểu, dùng hàm `primMST` để tính toán, sau đó trả về trọng số
- Trường hợp không còn node nào chưa được thăm `num==0`, nghĩa là tất cả các node đã được thăm, hàm trả về chi phí trực tiếp từ node bắt đầu tới node gốc là `graph[t][0]`.

```python
# Idea here is to form a graph of all unvisited nodes and make MST from that.
# Determine weight of that MST and connect it with the visited node and 0th node
# Prim's Algorithm used for MST (Greedy approach)
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]

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

```

Tiếp theo là hàm `checkPath`, mục đích của hàm này là kiểm tra xem đường đi có "hoàn chỉnh" hay không, nghĩa là đường đó có đủ tất cả đỉnh hay không. Các tham số:

- Tham số `tree` là cây chứa các node.
- Tham số `toExpand` là node cần mở rộng tiếp theo trong cây (node cần kiểm tra).
- Tham số `v` là số đỉnh trong đồ thị, dùng để so sánh.
  Các bước thực hiện:
- Lấy thông tin từ node kiểm tra `tnode = tree.get_node(str(toExpand.c_id))`.
- Xét trường hợp đặc biệt là nếu node được lấy ra là node gốc thì trả về False
- Duyệt từ node hiện tại tới node gốc (theo quan hệ cha con), đồng thời thêm các node đã thăm vào tập hợp `s` và danh sách `list1`
- Kiểm tra điều kiện hoàn thành, nếu chiều sâu `depth` = `v` nghĩa là đã đủ các node và đường đi cũng phải bắt đầu từ node gốc thì trả về True, ngược lại trả về False.

```python
def checkPath(tree, toExpand, V):
    tnode=tree.get_node(str(toExpand.c_id)) # Get the node to expand from the tree
    list1=[] # 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()
        # Go up in the tree using the parent pointer and add all
        # nodes in the way to 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
```

Cuối cùng là hàm chính chạy thuật toán A-star, hàm `startTSP` có các tham số đầu vào:

- Tham số `graph` là đồ thị cần thực hiện thuật toán.
- Tham số `tree` là cây chứa các node đã được mở rộng.

- Tham số `v` là tổng số đỉnh trong đồ thị.
  Các bước thực hiện:
- Khởi tạo các giá trị ban đầu, như
  - `goalState=0`: Biến kiểm tra trạng thái kết thúc.
  - `times=0`: Biến đếm số lần lặp.
  - `toExpand=TreeNode(0, 0, 0, 0, 0)`: Node mở rộng ban đầu.
- Tính toán giá trị Heuristic ban đầu (cho node gốc).
- Tạo node đầu tiên trong cây `tree.create_node("1", "1", data=TreeNode(0, 1, heu, heu, -1))` đại diện cho thành phố bắt đầu.
- Tạo hàng đợi ưu tiên *fringe* và thêm vào node trên với giá trị heuristic là `heu`.
- Lặp đến khi nào mà `goalState` thay đổi, trong đó triển khai thuật toán A-star, mở rộng các node và tính toán giá trị `f = g + h`. Node có giá trị `f` nhỏ nhất sẽ là node ưu tiên trong hàng đợi ưu tiên *fringe*.

```python
def startTSP(graph,tree,V):
    goalState = 0
    times = 0
    toExpand = TreeNode(0,0,0,0,0) # Node to expand
    key = 1
    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 0 and path is complete, we are done
        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
            # Evaluate f_values and h_values of adjacent nodes of the node to expand
            j = 0
            while(j < V):
                if(j!=toExpand.c_no):
                    h = heuristic(tree, toExpand.c_id, j, V, graph)
                    f_val = val + graph[j][toExpand.c_no] + h
                    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
```

NHẬN XÉT: Thuật toán chạy không bị lỗi, khá phức tạp nhưng nhờ các chú thích đã có thể dễ hiểu hơn.