1. Cấu trúc TreeNode

```
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
```

Cấu trúc TreeNode để biểu diễn một node trong tree.

- c_no đại diện cho mỗi node trong đồ thị (graph), ví dụ graph có 4 node thì c_no sẽ có giá trị từ 0 đến 3 (node 0 đến node 3).
- c_id đại diện cho mỗi node trong cây (tree), nghĩa là mỗi node expand trong tree sẽ có một c_id khác nhau, mặc dù nó có thể có cùng c_no.
- f_value là giá trị f , f_value = h + g.
- h_value là giá trị h , tức là hàm heuristic.
- parent_id là c_id của node cha của node đó trong cây.

2. Cấu trúc FringeNode

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

Cấu trúc FringeNode để biểu diễn một node trong fringe (hàm đợi ưu tiên). Mặc dù fringe_list trong hàm startTSP không được cài đặt bằng hàm đợi ưu tiên, nó chỉ đơn giản là 1 dictionary, nhưng ta sử dụng nó như là 1 hàm đợi ưu tiên (lấy ra node có giá trị f_value nhỏ nhất, sau đó xóa node đó).

- c_no ,f_value có ý nghĩa tương tự trong cấu trúc TreeNode

3. Hàm startTSP

Hàm startTSP có mục đích là để chạy thuật toán A* để tìm ra chu trình hamilton có độ dài nhỏ nhất (tức là chu trình đi qua các đỉnh đúng 1 lần với độ dài nhỏ nhất) , hàm heuristic ở đây chính là MST(cây khung tối tiểu).

```
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))
    fringe_list = {}
    fringe_list[key] = FringeNode(0, heu)
    key = key + 1
```

Các tham số :

- graph : đồ thị gốc
- tree : cây tìm kiếm
- V : số đỉnh của graph

- goalState là cờ, ban đầu gán =0, khi tìm được kết quả thì gán goalState=1, vòng lặp while dừng lại và thuật toán kết thúc.
- toExpand là node để expand trong cây, ban đầu đều gán các giá trị =0.
- key dùng để gán c_id cho các node trong cây,mỗi node trong cây đều có 1 key (c_id) khác nhau , ban đầu key =1 , tức c_id của node gốc trong cây là 1
- heu là giá trị heuristic của các node trong cây, ban đầu là heurictis của node gốc , ở đây p_id =-1 tức node gốc thì không có parent, t=0 tức là node 0, t=0 ở đây tương ứng với c_no=0, không phải c_id.
- Tiếp theo là tạo cây với node gốc (c_no=0, c_id=1).

```
create_node(arg1,arg2,data=,parent=)
```

Theo như document của thư viện treelib, hàm create_node có cấu trúc như hình, arg1 là tên của node, arg2 là identifier (định danh của node),mỗi node có thể có tên giống nhau nhưng định danh phải khác nhau, định danh ở đây còn dùng để node con tham chiếu đến node cha.

```
tree.create_node(str(toExpand.c_no), str(key), parent=str(
    toExpand.c_id), data=TreeNode(j, key, f_val, h, toExpand.c_id))
```

Theo như câu lệnh create_node bên trong while, có thể thấy cấu trúc của một node trong cây gồm có :

- Tên là c_no của node cha (expand.c_no),
- Định danh là key (c_id của node con),

- parent sẽ là định danh của node cha (expand.c_id),
- data thì như đã trình bày ở cấu trúc TreeNode.

```
tree.create_node("1", "1", data=TreeNode(0, 1, heu, heu, -1))
```

Theo như cấu trúc ở trên, thì ở đây hàm tạo node đầu tiên, tham số thứ nhất đáng ra nên là "-1" (toExpand.c_no=-1) tức là node gốc thì chưa có cha. Dù sao thì tên cũng không ảnh hưởng đến thuật toán !!!

- fringe_list ở đây là 1 dictionary đóng vai trò như là 1 hàng đợi ưu tiên, key chính là các c_id của các node trong cây, value là (c_no, f) tương ứng.
- Mỗi lần tạo một node trong cây thì key sẽ tăng lên 1, tức là c_id của các node được tạo trong cây sẽ có giá trị lần lượt từ 1 đến n với n là số node trong cây.

```
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

    # Heuristic value of selected node
    h = tree.get_node(str(toExpand.c_id)).data.h_value
    val = toExpand.f_value - h                          # g value of selected node
    # Check path of selected node if it is complete or not
    path = checkPath(tree, toExpand, V)
```

Đoạn code này chỉ đơn giản là lấy ra node có f_value nhỏ nhất trong fringe và gán vào toExpand. Sau đó tính giá trị h và g của toExpand và check xem toExpand đã phải là goal hay chưa bằng hàm checkPath.

```
    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
                # g(parent) + g(parent->child) + h(child)
                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
```

- Nếu toExpand.c_no ==0 tức toExpand là node gốc và path==1 tức đường đi đã hoàn thành thì gán goalState =1, cost lúc này chính là toExpand.f_value(vì toExpand.c_no =0 nên h(toExpand)=0 , do đó toExpand.f_value chính là g(toExpand) tức là quãng đường thực tế của chu trình).
- Còn nếu không thì xóa toExpand ra khỏi fringe, tính f_value của các node kề với toExpand trong graph và thêm các node kề vào fringe, thêm các node con của toExpand vào cây cứ thế lặp lại cho đến khi tìm được kết quả. Ở đây như đã trình bày ở trên thì mỗi node con của toExpand được thêm vào cây sẽ có
    - Tên là str(toExpand.c_no)
    - Định danh(identifier) là str(key)
    - parrent sẽ là định danh của toExpand (toExpand.c_id)
    - data của node con sẽ gồm có c_no chính là j, c_id như đã nói ở trên chính là key, f_val, h đã tính ở trên, parent chính là toExpand.c_id.

4. Hàm checkPath

Mục tiêu của hàm checkPath là để kiểm tra xem toExpand có phải goal node hay chưa.

```
def checkPath(tree, toExpand, V):
    # Get the node expand from the tree
    tnode = tree.get_node(str(toExpand.c_id))
    list1 = list()  # 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 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
```

Các tham số :

- tree : được xây dựng trong hàm startTSP.
- toExpand : node toExpand trong hàm startTSP.
- V : số đỉnh của graph.

- tnode là để truy ngược các node từ toExand lên trên trong cây thông qua parent_id.
- list1 là để lưu các tnode từ tương ứng bên trên.
- depth để lưu độ sâu của toExpand.
- s là set để lưu các node trong path
- Ban đầu tnode là toExpand, list1 rỗng, ta sẽ check nếu tnode.data.c_id ==1 thì return False vì nếu tnode.data.c_id ==1 thì toExpand là node gốc trong cây, nghĩa là path chưa thể hoàn thành được.
- Ngược lại, dùng tnode để truy ngược lên trên các node trong cây thông qua parent id, đến node gốc (tnode.data.c_id==1) thì dừng, trong quá trình đó lưu tnode.data.c_no vào list1. list 1 ở đây chính là path từ node toExpand đến node gốc.

- check nếu độ sâu của toExpand = số cạnh của đồ thị và số phần tử của s = số cạnh của đồ thị và list1[0]=0 (tức toExpand.c_no=0 hay đã hoàn thành chu trình) thì nghĩa là đã tìm được chu trình, ta đảo ngược list1 và in ra list1 sau đó return 1, ngược lại return 0.

5. Hàm heuristic

Mục tiêu của hàm heuristic đúng như tên gọi là tìm heuristic của một node.

```
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)
```

Các tham số :

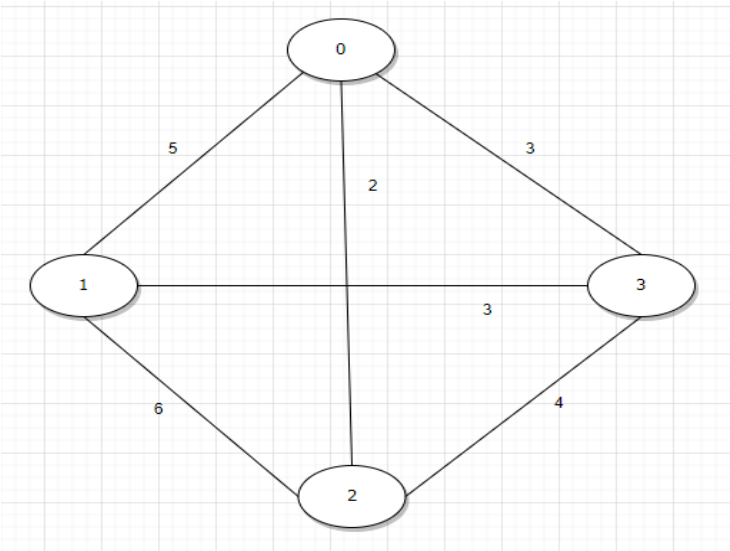
- tree : là tree được tạo ra ở trong hàm startTSP
- p_id : chính là toExpand.c_id trong hàm startTSP
- t chính là c_no của các node con (j) của toExpand trong hàm startTSP
- V là số đỉnh của đồ thị
- graph là đồ thị

Đoạn code trên để thêm các node bao gồm : 0, node đang xét (t), các node đã đi qua vào visited, từ đó xây dựng cây khung tối tiểu (MST) của đồ thị chứa các node không nằm trong visited.

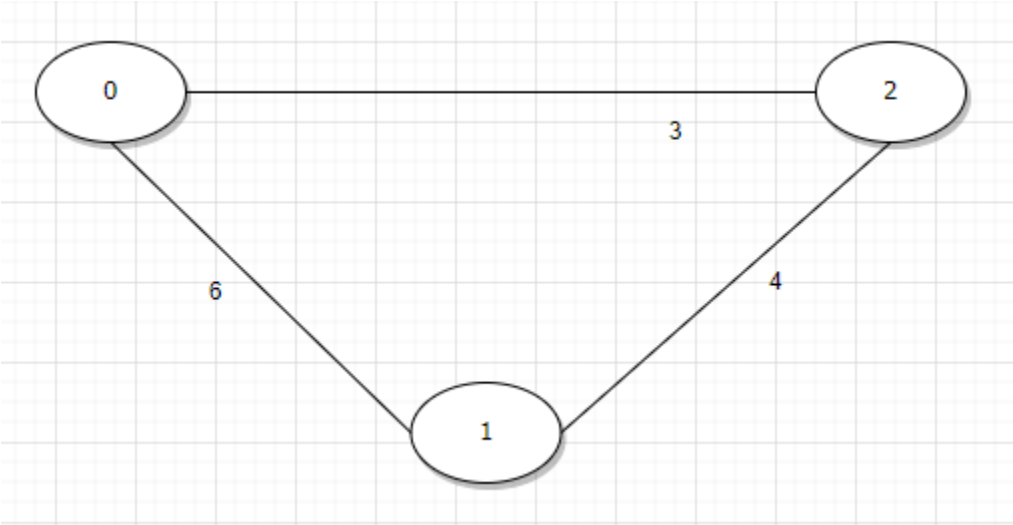
```
num = V - 1  # No of unvisited nodes
if (num != 0):
    g = Graph(num)
    d_temp = {}
    key = 0
    # d_temp dictionary stores mappings of original city no as (key) and new sequential
    for i in range(V):
        if (i not in visited):
            d_temp[i] = key
            key = key + 1

    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(graph, d_temp, t)
    return mst_weight
else:
    return graph[t][0]
```

Đoạn code trên xây dựng đồ thị g gồm các node không nằm trong visited, d_temp là 1 dictionary để mapping các node(c_no) của đồ thị gốc (graph) với g. mst_weight sẽ được tính trong hàm primMST.

Ở đây, để dễ minh họa code, dùng đồ thị cụ thể như hình dưới



Ví dụ , ban đầu , khi tìm heurisctic của node 0 thì visited sẽ là {0}, g sẽ như hình bên dưới, d_temp là {1:0,2:1,3:2} , d_temp ở đây có ý nghĩa là node 1 trong graph tương ứng với 0 trong g, tương tự cho 2, 3.



6. Hàm primMST và minKey

Mục tiêu của hàm primMST là tìm cây khung tối tiểu(MST) bằng thuật toán prim. Hàm minKey chỉ đơn giản là hàm nhỏ để hỗ trợ cho hàm primMST.

```python
def primMST(self, g, 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 distar
        # the vertex in not in the shotest 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, g, d_temp, t)
```

Các tham số của hàm primMST :

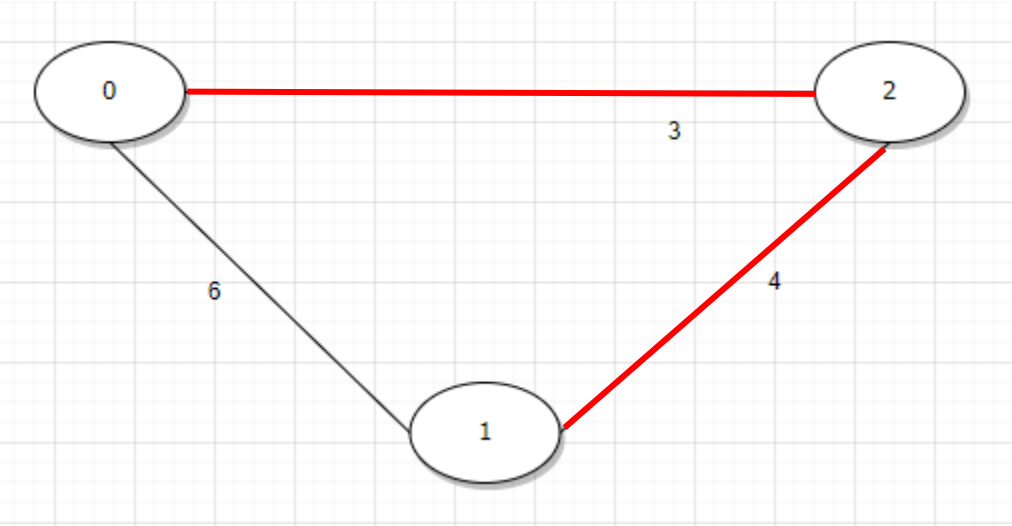
- self : đồ thị g được xây dựng trong hàm heuristic
- g : truyền từ hàm heuristic, g ở đây là đồ thị gốc (graph), không phải là g được xây dựng trong hàm heuristic(self).
- d_temp : truyền từ hàm hàm heuristic.
- t : truyền từ hàm heuristic.

Đoạn code trên để tìm ra cây khung tối tiểu của g bằng thuật toán prim.

- key dùng để lưu và pick các cạnh có giá trị nhỏ nhất trong g.
- parent để lưu các giá trị dùng để xây dựng cây khung tối tiểu(xem ví dụ cụ thể bên dưới).
- mstSet để lưu các đỉnh trong cây khung tối tiểu.

- hàm minKey là để chọn đỉnh có khoảng cách ngắn nhất mà không nằm trong mstSet.
- hàm trả về giá trị là printMST(sẽ được trình bày ý nghĩa trong hàm printMST).

Tiếp tục với ví dụ tìm heuristic của node 0 bên trên , đây là cây khung tối tiểu (tô đỏ) của self (g trong hàm heuristic) sau khi chạy thuật toán.



- parent lúc này là [-1,2,0], tức parent[0]=-1, parent[1]=2,parent[2]=0 theo như đồ thị.

7. Hàm printMST

Mục tiêu của hàm là từ parent trong hàm primMST, in ra heuristic của t từ cây khung tối tiểu được xây dựng bằng parent.

```python
def printMST(self, parent, g, 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 (g[0][r_temp[i]] < min1):
            min1 = g[0][r_temp[i]]
        if (g[0][r_temp[parent[i]]] < min1):
            min1 = g[0][r_temp[parent[i]]]
        if (g[t][r_temp[i]] < min2):
            min2 = g[t][r_temp[i]]
        if (g[t][r_temp[parent[i]]] < min2):
            min2 = g[t][r_temp[parent[i]]]

    return (sum_weight + min1 + min2) % 10000
```

Hàm printMST sau khi chỉnh sửa

Các tham số :

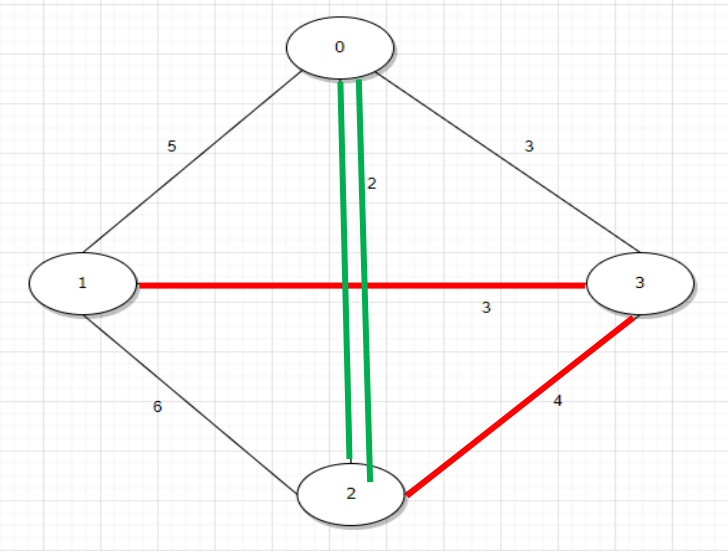
- self : đồ thị g được xây dựng trong hàm heuristic.
- parent : được xây dựng trong hàm primMST.
- g : truyền từ hàm primMST (chính là đồ thị gốc graph).
- d_temp : truyền từ hàm primMST(xây dựng trong hàm heuristic).
- t : truyền từ hàm primMST.

Trong file code gốc g ở đây là graph, điều này không sai vì trong graph ở đây được hiểu là graph được khai báo trong if __name__==’__main__’, tuy nhiên ở đây có tham số truyền vào là g thì dùng g thay graph sẽ hợp lý hơn, g và graph đều giống nhau (g = graph != self.graph).

- min1 để tính khoảng cách ngắn nhất từ node 0 đến MST.
- min 2 để tính khoảng cách ngắn nhất từ node t đến MST.
- sum_weight là tổng trọng số của MST.

Kết quả trả về của hàm printMST sẽ là tổng trọng số của MST + khoảng cách ngắn nhất từ 0 đến MST + khoảng cách ngắn nhất từ t đến MST.

Tiếp tục với ví dụ cụ thể bên trên:



heristic của 0 được tính là : 3+4+2+2=11 với :

- 3+4 là tổng trọng số của MST.
- 2 là khoảng cách ngắn nhất từ 0 đến MST.
- 2 là khoảng cách ngắn nhất từ t(lúc này là 0) đến MST.