BÀI 5: CÁC THUẬT TOÁN TÌM KIẾM (tiếp theo)

I. MỤC TIÊU:

Sau khi thực hành xong, sinh viên nắm được:

- Áp dụng các thuật toán tìm kiếm vào các bài toán thực tế.

II. TÓM TẮT LÝ THUYẾT:

1. Traveling Salesperson Problem - TSP:

Cho trước n thành phố và các khoảng cách d_{ij} giữa mỗi cặp thành phố, tìm tour ngắn nhất sao cho mỗi thành phố được viếng thăm chỉ một lần.

2. Cây khung nhỏ nhất (Minimum Spanning Tree – MST):

- Cho đồ thị $G = (V, E)$ vô hướng với các cạnh d_{ij}
- Một cây khung T là một đồ thị con của G mà nó là
  - 1 cây (đồ thị không tuần hoàn liên thông)
  - Mở rộng tất cả các đỉnh
- Mỗi cây khung có $(n-1)$ cạnh
- Chiều dài của mỗi cây khung T là $\sum_{(i,j)\in T} d_{ij}$
- Bài toán cây khung nhỏ nhất là tìm 1 cây khung có chiều dài nhỏ nhất.

3. Thuật toán cho MST:

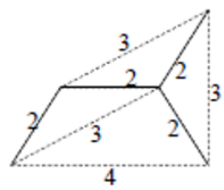
- **Bước 1:** tìm cạnh ngắn nhất trong đồ thị. Nếu có nhiều hơn 1 cạnh như vậy thì chọn 1 ngẫu nhiên 1 cạnh. Đánh dấu cạnh này và các đỉnh được kết nối.
- **Bước 2:** Chọn cạnh ngắn nhất tiếp theo, trừ khi nó tạo thành 1 chu trình với các cạnh đã được đánh dấu trước đó. Đánh dấu cạnh đó và các đỉnh được kết nối.
- **Bước 3:** Nếu tất cả các cạnh được kết nối thì khi đó ta đã hoàn thành. Ngược lại, lặp lại Bước 2.

4. **Cây khung nhỏ nhất dựa vào heuristic:**

- **Bước 1:** Xây dựng một cây khung nhỏ nhất
- **Bước 2:** Chọn nút gốc là nút bất kỳ.
- **Bước 3:** Duyệt qua tất cả các đỉnh bằng tìm kiếm theo chiều sâu, ghi lại tất các đỉnh (đỉnh đã viếng thăm và đỉnh chưa viếng thăm)
- **Bước 4:** Sử dụng chiến lược nhanh chóng trực tiếp hơn để khởi tạo một tour khả thi.

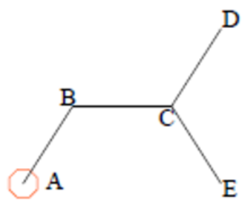
5. **Ví dụ:**

- **Bước 1:** Xây dựng một cây khung nhỏ nhất

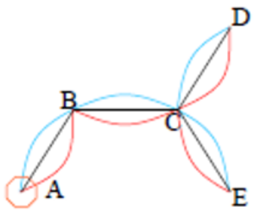


MST có thể được giải trong $O(n^2)$, cũng là chặn dưới cho TSP, $W^* \leqslant L^*$.

- **Bước 2:** Chọn nút gốc là 1 nút bất kỳ

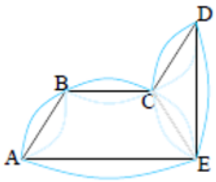


- **Bước 3:** Duyệt qua tất cả các đỉnh



Chuỗi là: $A - B - C - D - C - E - C - B - A$, chiều dài của tour là $2W^*$

- **Bước 4:** Sử dụng chiến lược nhanh chóng trực tiếp hơn để khởi tạo một tour MST $A - B - C - D - (C) - E - (C - B) - A$



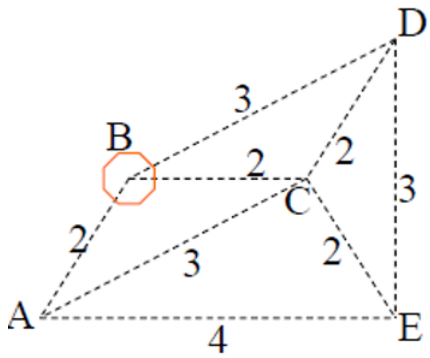
Tour MST là $A - B - C - D - E - A$, chiều dài của tour TSP nhỏ hơn bằng $2W^*$.

6. Heuristic chèn gần nhất:

- **Bước 1:** Chọn 1 nút v bất kỳ và cho chu trình C chỉ chứa v
- **Bước 2:** Tìm một nút bên ngoài C gần nhất với 1 nút trong C, gọi là k.
- **Bước 3:** Tìm 1 cạnh $\{ij\}$ trong C sao cho $d_{ik} + d_{kj} - d_{ij}$ là tối tiểu.
- **Bước 4:** Xây dựng một chu trình C mới bằng việc thay thế $\{ij\}$ với $\{ik\}$ và $\{k, j\}$.
- **Bước 5:** Nếu chu trình C hiện tại chứa tất cả các đỉnh thì dừng. Ngược lại, quay lại Bước 2.

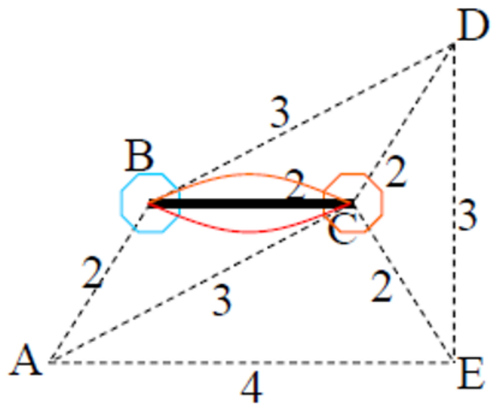
7. Ví dụ:

- **Bước 1:** Chọn 1 nút v bất kỳ và cho chu trình C chỉ chứa v



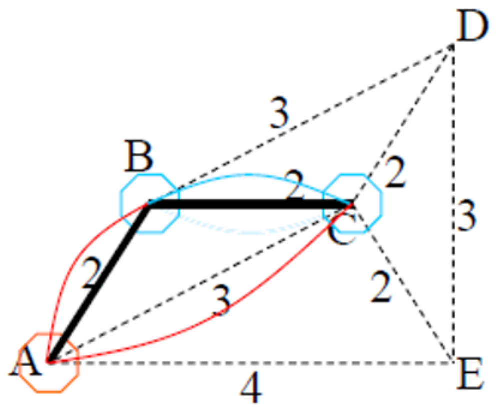
Chiều dài của tour hiện tại là 0.

- **Bước 2 - 3 - 4:** Lần lặp đầu tiên



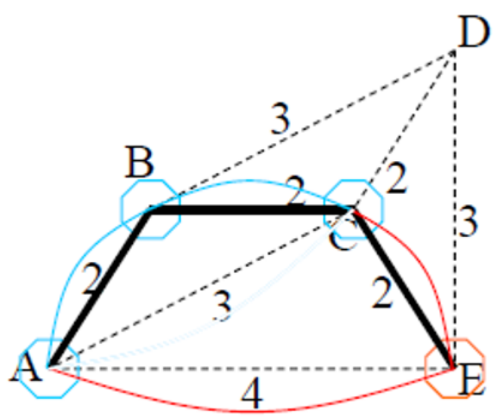
Chiều dài của C không lớn hơn gấp đôi chiều dài của đường in đậm (bằng nhau trong lần lặp này).

Lần lặp thứ 2:



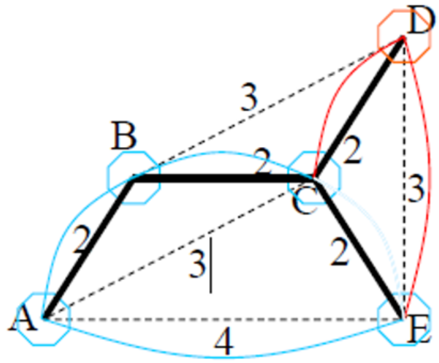
chiều dài của tour C hiện tại là không lớn hơn gấp đôi chiều dài đường in đậm.

Lần lặp thứ ba:



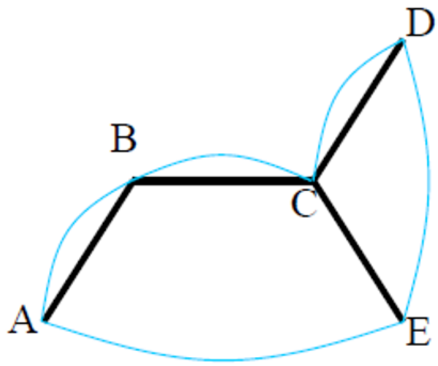
Chiều dài của tour C hiện tại không lớn hơn 2 lần chiều dài đường in đậm.

Lần lặp thứ 4:



Chiều dài của tour C hiện tại không lớn hơn 2 lần chiều dài đường in đậm.

- **Bước 5:** Kết quả cuối cùng là $A - B - C - D - E - A$



Chiều dài của tour C hiện tại không lớn hơn 2 lần chiều dài đường in đậm.

8. Thuật toán A^* cho việc giải bài toán TSP:

- **Trạng thái ban đầu:** Agent ở thành phố bắt đầu và không viếng thăm bất kỳ thành phố nào khác.
- **Trạng thái kết thúc:** Agent đã viếng thăm tất cả các thành phố và đến thành phố bắt đầu 1 lần nữa.
- **Hàm successor:** khởi tạo tất cả các thành phố chưa viếng thăm.
- **Chi phí cạnh:** khoảng cách giữa các thành phố được biểu diễn bởi các nút, sử dụng chi phí này để tính $g(n)$.

- **h(n):** khoảng cách tới thành phố chưa viếng thăm gần nhất ước lượng khoảng cách đi từ tất cả thành phố bắt đầu.

III. NỘI DUNG THỰC HÀNH:

1. Bài toán:

Sử dụng thuật toán A^* để giải bài toán TSP và heuristic được sử dụng là cây khung nhỏ nhất.

2. Cài đặt:

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

# Structure to represent tree nodes in the A* expansion
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

# Structure to represent fringe nodes in the A* fringe list
class FringeNode(object):
        def __init__(self, c_no,  f_value):
            self.f_value = f_value
            self.c_no = c_no


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):

        # Initilaize 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 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 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,d_temp,t)


# Idea here is to form a grpah 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)):
```

```python
                    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]


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
        # 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

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)  # Heurisitic 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(Dictionary)(FL)
    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
        # 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
if __name__ == '__main__':

    V=4
    graph=[[0,5,2,3],[5,0,6,3],[2,6,0,4],[3,3,4,0]]

    tree = Tree()
    ans = startTSP(graph,tree,V)
    print("Ans is "+str(ans))
```

3. Yêu cầu:

- Cài đặt và thực thi chương trình. Nếu chương trình bị báo lỗi thì lỗi ở dòng nào và sửa lại như thế nào?
- Viết báo cáo trình bày lại tất cả những gì em hiểu liên quan tới bài thực hành. Nhận xét?