Thực Hành Nhập Môn Trí Tuệ Nhân Tạo Tuần 5

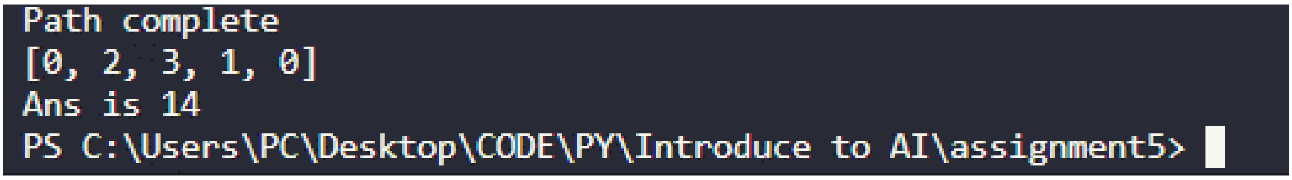
Phan Hồng Trâm - 21110414

December 2023

Mục lục

1 Cài đặt và thực thi chương trình. Nếu chương trình bị báo lỗi thì lỗi ở dòng nào và sửa lại như thế nào?

Chương trình không bị báo lỗi và vẫn chạy ra kết quả.



Hình 1: Kết quả chạy thuật toán ban đầu

2 Trình bày lại tất cả những gì em hiểu liên quan tới bài thực hành

2.1 Traveling Salesperson Problem - TSP

Bài toán người bán hàng (tiếng Anh: travelling salesman problem - TSP) là một bài toán **NP-hard** (bài toán có độ phức tạp tăng theo hàm số mũ). Bài toán được nêu ra lần đầu tiên năm 1930 và là một trong những bài toán được nghiên cứu sâu nhất trong tối ưu hóa. Bài toán được phát biểu như sau:

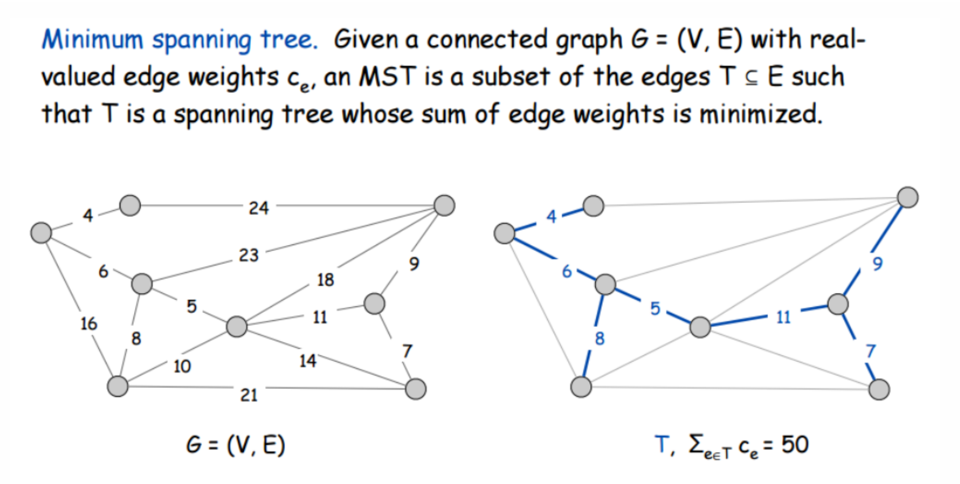
- Có một người giao hàng cần đi giao hàng tại n thành phố. Anh ta xuất phát từ một thành phố nào đó, đi qua các thành phố khác để giao hàng và trở về thành phố ban đầu. Mỗi thành phố chỉ đến một lần, và khoảng cách từ một thành phố đến các thành phố khác đã được biết trước. Hãy tìm một chu trình (một đường đi khép kín thỏa mãn điều kiện trên) sao cho **tổng độ dài các cạnh** là **nhỏ nhất**.
- Phát biểu dưới dạng đồ thị: Bài toán được mô hình hóa như một đồ thị vô hướng có trọng số, trong đó mỗi thành phố là một đỉnh của đồ thị còn đường đi giữa các thành phố là mỗi cạnh. Khoảng cách giữa hai thành phố là độ dài cạnh. Hãy tìm một đường đi bắt đầu từ đỉnh xuất phát, đi qua tất cả các đỉnh của đồ thị đúng 1 lần và quay trở lại đinh xuất phát sao cho **độ dài của dường đi** là **nhỏ nhất.**

Cho đến ngày này, các nhà khoa học đã tìm ra được nhiều phương pháp để giải quyết bài toán. Chúng ta cùng tìm hiểu cách sử dụng thuật toán A^* kết hợp với **cây khung nhỏ nhất** (Minimum Spanning Tree - MST) và thuật toán heuristic **chèn gần nhất** (Nearest Insertion) để giải bài toán TSP và heuristic được sử dụng là cây khung nhỏ nhất.

2.2 Cây khung nhỏ nhất (Minimum Spanning Tree – MST)

Phát biểu lý thuyết: Cây khung nhỏ nhất của đồ thị $G = (V, E)$ vô hướng có trọng số với các cạnh d_{ij} sao cho:

- Tập hợp các cạnh này *không chứa chu trình* và *liên thông* - nghĩa là từ một đỉnh bất kỳ có thể đi tới các đỉnh khác mà chỉ dùng các cạnh trên tập hợp đó.
- Tổng trọng số của các cạnh trong tập hợp này là *nhỏ nhất*.



Hình 2: Minh họa cây khung T nhỏ nhất của đồ thị có trọng số G

Lưu ý: Chúng ta không quan tâm tổng trọng số lớn đến mức nào đối với một đường dẫn nhất định đến và đi từ bất kỳ nút nào trong MST. Chúng ta chỉ quan tâm đến tổng trọng số của tất cả các cạnh trong MST. Cho dù một đường dẫn cụ thể trong MST có lớn đến mức nào đi chăng nữa, nếu tổng của tất cả các trọng số cạnh trong MST là tối thiểu và vẫn có thể đến được mọi nút, thì đó là MST.

2.2.1 Prim's Algorithm cho MST

Thuật toán Prim còn được gọi là thuật toán Jarník. Thuật toán Prim hoạt động bằng cách bắt đầu từ một đỉnh tùy ý, thêm cạnh có trọng số tối thiểu nối cây với một đỉnh mới và lặp lại quá trình này cho đến khi tất cả các đỉnh đều được đưa vào cây.
Nếu bạn đang tìm cây khung nhỏ nhất (MST) của đồ thị bằng thuật toán Prim, thì không được tạo thành chu trình. Nghĩa là, nếu A liên kết với B và B liên kết với C thì C không thể liên kết lại với A vì điều đó sẽ tạo thành một chu trình.

Cách triển khai thuật toán Prim:

- Phải bao gồm tất cả các đỉnh của đồ thị.
- Đỉnh có trọng số nhỏ nhất phải được chọn trước.
- Tất cả các đỉnh phải được kết nối.
- Không được tạo thành một chu trình.

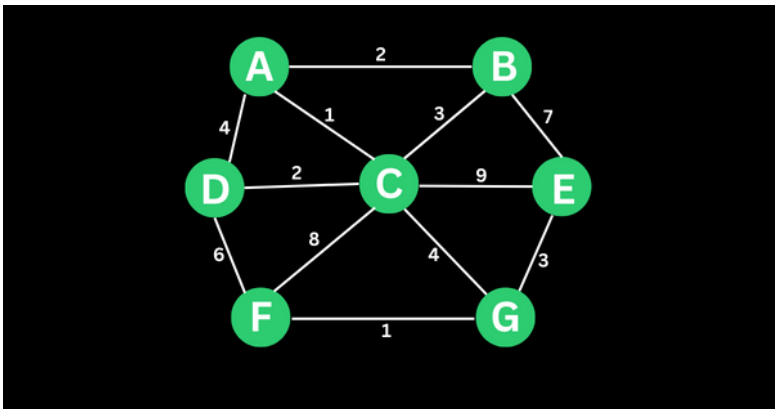
Mã giả của thuật toán Prim:

Thuật toán 1 Thuật toán Prim

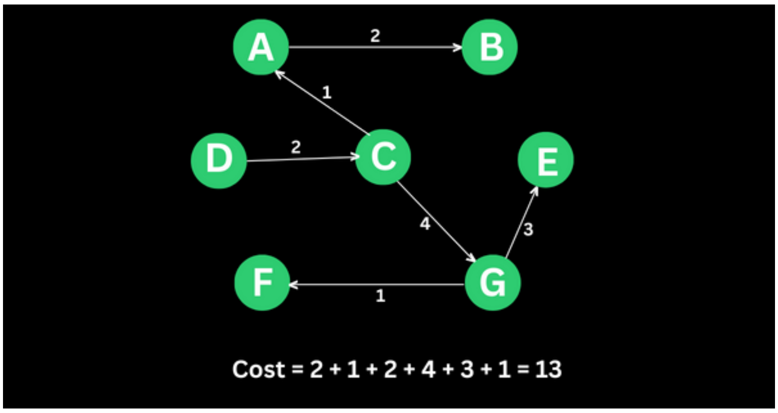
Đầu vào: Đồ thị vô hướng, liên thông, có trọng số $G = (V, U)$
Đầu ra: Cây khung nhỏ nhất T của đồ thị G.

Chọn một đỉnh bất kì $s \in G$.
$D[s] = 0$
for each $v \in V \setminus \{s\}$ **do**
    $D[v] = \infty$
    v.parent = **null**
end for
Khởi tạo $T = \emptyset$
Khởi tạo hàng đợi ưu tiên $Q = (D[v], v)$ **for each** $v \in V$.
T.connect(u)
while Q không rỗng **do**
    $u = Q$.removeMin()
    for each $v \in G$.adjacent[u] **do**
        if $v \in Q$ và $w(u, v) < D[v]$ **then**
            $D[v] = w(u, v)$
            v.parent = u
            T.connect(v)
        end if
    end for
end while

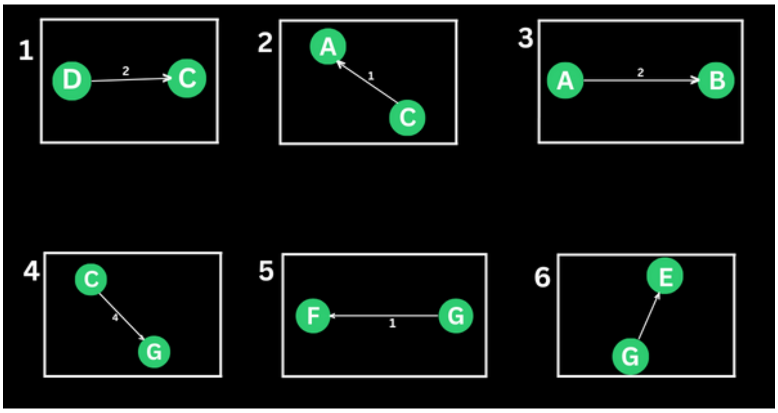
Ví dụ minh họa thuật toán Prim:



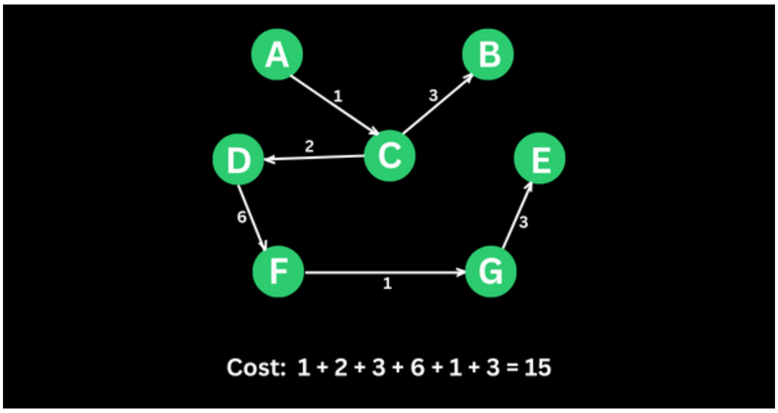
((a)) Đồ thị ban đầu



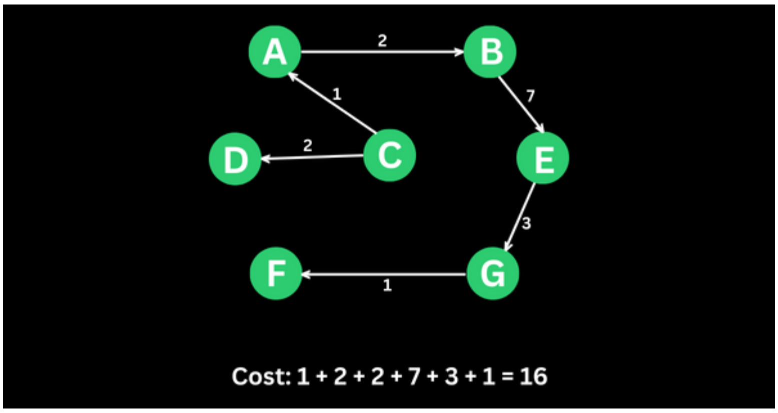
((b)) Kết quả của thuật toán Prim khi chọn D là nút ban đầu



((c)) Các bước làm của thuật toán Prim



((a)) Kết quả của thuật toán Prim khi chọn A là nút ban đầu



((b)) Kết quả của thuật toán Prim khi chọn C là nút ban đầu

2.2.2 Kruskal's algorithm cho MST

Thuật toán Kruskal tìm một tập hợp các cạnh tạo thành một cây chứa tất cả các đỉnh của đồ thị và có tổng trọng số các cạnh là nhỏ nhất.Thuật toán Kruskal là một ví dụ của thuật toán tham lam (greedy algorithm). Thuật toán này xuất bản lần đầu tiên năm 1956, bởi Joseph Kruskal.
Mô tả thuật toán:
Giả sử ta cần tìm cây bao trùm nhỏ nhất của đồ thị G. Thuật toán bao gồm các bước sau:

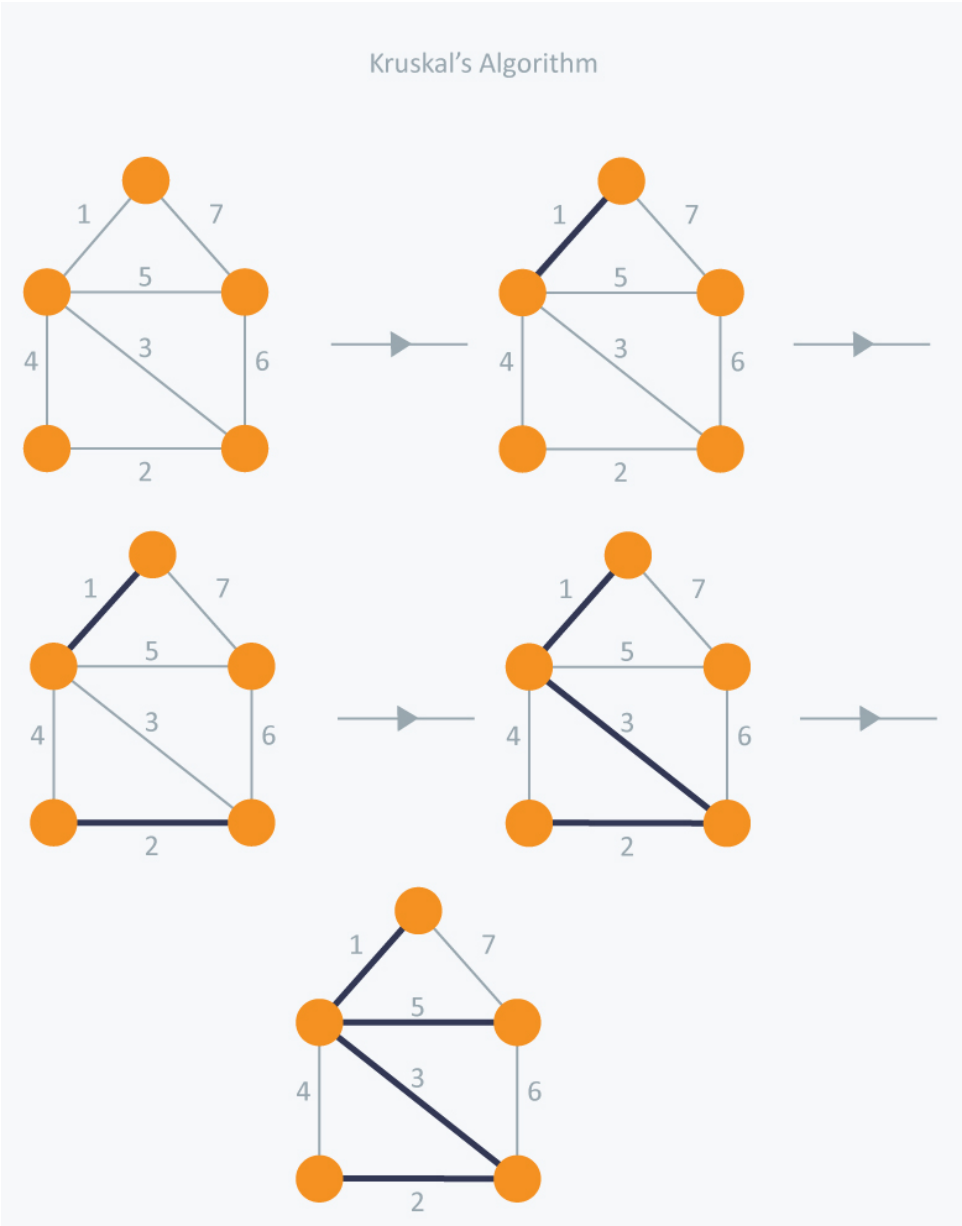
- Khởi tạo rừng F (tập hợp các cây), trong đó mỗi đỉnh của G tạo thành một cây riêng biệt
- Khởi tạo tập S chứa tất cả các cạnh của G
- Chừng nào S còn **khác rỗng** và F gồm hơn một cây
  - Xóa cạnh nhỏ nhất trong S
  - Nếu cạnh đó nối hai cây khác nhau trong F, thì thêm nó vào F và hợp hai cây kề với nó làm một
  - Nếu không thì loại bỏ cạnh đó.

Khi thuật toán kết thúc, rừng chỉ gồm đúng một cây và đó là một cây khung nhỏ nhất của đồ thị G.
Mã giả của thuật toán Kruskal:

Thuật toán 2 Thuật toán Kruskal(G)

$F := \varnothing$
for each v **in** G.V **do**
  MAKE-SET(v)
end for
for each u, v **in** G.E sắp xếp theo trọng số (u, v) tăng dần **do**
  if FIND-SET(u) $\neq$ FIND-SET(v) **then**
    $F := F \cup \{\{u, v\}\}$
    UNION(FIND-SET(u), FIND-SET(v))
  end if
end for
return F

Ví dụ minh họa thuật toán Kruskal:



Hình 5: Minh họa thuật toán Kruskal

2.3 Heuristic chèn gần nhất (Nearest insertion heuristic)

Nearest Insertion là kỹ thuật mở rộng đường đi, gọi là *tour*, bằng cách chèn những điểm mới vào những điểm trong tour trước đó. Giải thuật này tìm những điểm nào không nằm trong tour gần nhất với bất kì điểm nào trong tour, sau đó chèn giữa hai điểm nào đó trong tour sao cho tổng trọng số trong tour là nhỏ nhất. Độ phức tạp của giải thuật này là $O(n^2)$ vì các bước tìm đỉnh và cạnh để chèn có độ phức tạp $O(n)$.
Mã giả cho thuật toán Nearest Insertion:

Thuật toán 3 Nearest Insertion

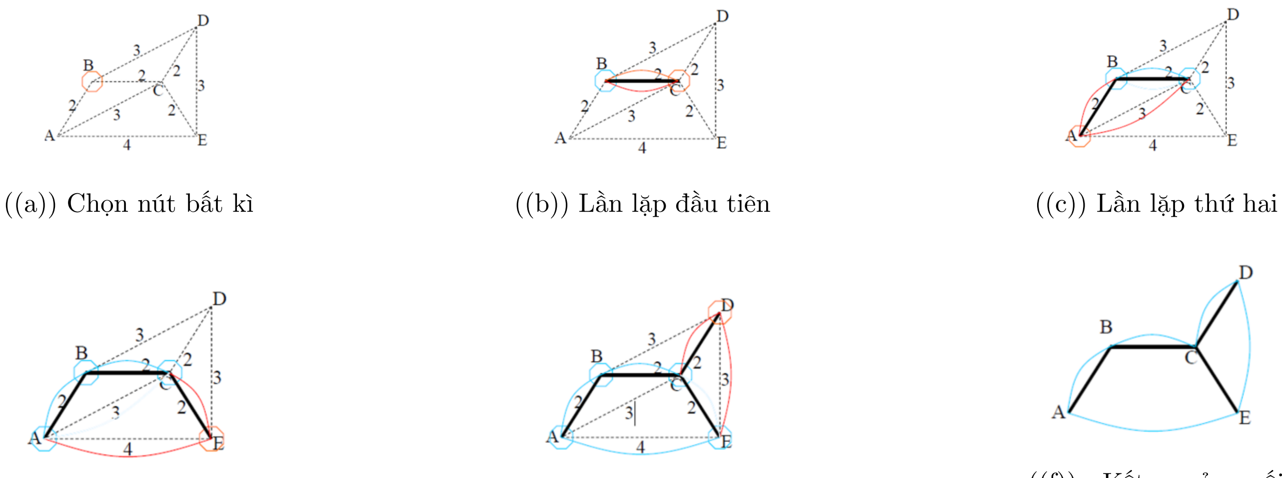
Đầu vào: Đồ thị vô hướng, liên thông, có trọng số $G = (V, U)$
Đầu ra: Đường đi ngắn nhất P

P.addTour(k)
Tìm node r sao cho c_{kr} nhỏ nhất.
P.addTour(r)
for each $v \in V \setminus \{i, r\}$ **do**
    Tìm node $r \notin P.V$ sao cho c_{vr} nhỏ nhất.
    Tìm cạnh $(i, j) \in P.E$ sao cho $c_{ir} + c_{rj} - cij$ nhỏ nhất.
    Chèn r vào giữa i và j.
end for

Trường hợp tệ nhất:

$$\frac{length\_of\_nearest\_insertion\_tour}{length\_of\_optimal\_tour} \leq 2$$

Ví dụ về Heuristic chèn gần nhất:



((a)) Chọn nút bất kì
((b)) Lần lặp đầu tiên
((c)) Lần lặp thứ hai
((d)) Lần lặp thứ ba
((e)) Lần lặp thứ tư
((f)) Kết quả cuối cùng

Hình 6: Minh hoạ giải thuật heuristic Nearest Insertion

2.4 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 một 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.

2.5 Code `Python`: Giải thích và kết quả

Vận dụng thuật toán Prim và Heuristic chèn gần nhất để tìm cây khung nhỏ nhất (MST), ta có thể cài đặt chương trình giải bài toán TSP bằng thuật toán A*.
Một số hàm cần lưu ý:

- `Graph.printMST(self, parent, g, d_temp, t)`: in ra cây khung nhỏ nhất của đồ thị và trả về trọng số của cây khung nhỏ nhất.

```python
# Hàm tiện ích để in MST cấu trúc được lưu trữ trong hàm dict parent[]
def printMST(self, parent, d_temp, t):
    # Xuất ("Edge\ tWeight")
    sum_weight = 0
    min1 = 10000
    min2 = 10000
    r_temp = {}
    for k in d_temp:
        r_temp[d_temp[k]] = k

    for i in range(1, self.V):
        # Xuất ra (parent[i], "-",i, 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
```

- `Graph.minKey(self, key, mstSet)`: tìm giá trị nhỏ nhất trong tập hợp các đỉnh của cây khung nhỏ nhất.

```python
    # Hàm tiện ích để tìm vertex (đỉnh)
    # với giá trị khoảng cách nhỏ nhất từ bộ các đỉnh
    # chưa bao gồm trong đồ thị cây đường đi ngắn nhất
    def minKey(self, key, mstSet):
        #Intialize 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
```

- `Graph.primMST(self, g, d_temp, t)`: thực hiện thuật toán tìm cây khung nhỏ nhất của đồ thị.

```python
# Hàm để xây dựng và in MST cho đồ thị graph
# Đại diện sử dụng ma trận
    def primMST(self, d_temp, t):
        # Giá trị khóa để chọn cạnh có khoảng cách nhỏ nhất
        key = [sys.maxsize] * self.V
        parent = [None] * self.V  # Array to store constructed MST
        # Tạo key 0 để đỉnh đc chọn như là đỉnh đầu tiên
        key[0] = 0
        mstSet = [False] * self.V
        sum_weight = 10000
        parent[0] = -1  # Nút đầu tiên luôn là rễ

        for c in range(self.V):
            # Chọn đỉnh có khoảng cách nho nhất của bộ các đỉnh chưa đc ghé đến
            # u is always equa; 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 m
            #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 is false for vertices not yet included in MST graph
                #Update the key only if graph[u][v] is smaller than key[v]
                if 0 < self.graph[u][v] < key[v] and mstSet[v] == False:
                    key[v] = self.graph[u][v]
                    parent[v] = u
        return self.printMST(parent, d_temp, t)
```

- `heuristic(tree, p_id, t, V, graph)`: thực hiện giải thuật heuristic Nearest Insertion.

```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 the 0th 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)
    num = V - l #no of unvisited nodes
    if num != 0:
        g = Graph(num)
        d_temp = {}
        key = 0
        # d_temp dictionary stores mapping 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]
```

- `checkPath(tree, toExpand, V)`: kiểm tra và in ra đường đi ngắn nhất của bài toán.

```python
def checkPath(tree, toExpand, V):
    tnode = tree.get_node(str(toExpand.c_id))
    # Lấy nuts để mở rộng từ đồ thị cây
    list1 = list()
    # List để lưu trữ đường đi
    # Cho nút đầu tiên:
    if tnode.data.c_id == 1:
        # print("In if")
        return 0
    else:
        # print("In else")
        depth = tree.depth(tnode)  # Kiểm tra độ sâu của cây
        s = set()
        # Đi lên cây sử dụng con trỏ cha và thêm tất cả các nút trên đường đi đến set và 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
```

- `startTSP(graph, tree, V)`: thực hiện việc giải bài toán TSP bằng thuật toán A*.

```python
def startTSP(graph, tree, V):
    goalState = 0
    times = 0
    toExpand = TreeNode(0, 0, 0, 0, 0)  # Nút để mở rộng
    key = 1  # Định danh duy nhất cho nút trên cây
    heu = heuristic(tree, -1, 0, V, graph)
    tree.create_node("1", "1",
                     data=TreeNode(0, 1, heu, heu, -1))  # Tạo nút đầu tiên trên cây nghĩa là  thành phố gốc 0th
    fringe_list = {}
    fringe_list[key] = FringeNode(0, heu)
    key = key + 1
    while goalState == 0:
        minf = sys.maxsize
        # Chọn nút có f_value nhỏ nhất từ 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
        val = toExpand.f_value - h  # giá trị g của nút đc chọn
        path = checkPath(tree, toExpand, V)  # Kiểm tra đường của nút đc chọn nếu nó hoàn thành hay không
        # Nếu nút mở rộng là 0 và path đã hoàn thành thì ngừng bài toán
        # Ta kiểm tra nút ở thời điểm mở rộng và không ở thời điểm của thế hệ
        if toExpand.c_no == 0 and path == 1:
            goalState = 1;
            cost = toExpand.f_value  # Tổng giá trị thật sự
        else:
            del fringe_list[toExpand.c_id]  # Loại bỏ nút từ FL
            j = 0
            # Đánh giá f_values và h_value của các nút liền kề của nút để mở rộngk
            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
```

Chạy chương trình:

```python
if __name__ == '__main__':
    # Trường hợp 1:
    V = 4
    graph = [[0, 5, 2, 3],[5, 0, 6, 3], [2, 6, 0, 4], [3, 3, 4, 0]]

    # # Trường hợp 2:
    # V = 2
    # graph = [[0,300],[300,0]]

    # # Trường hợp 3:
    # V = 3
    # graph = [[0,300,200],[300,0,500],[200,500,0]]

    # # Trường hợp 4:
    # V = 4
    # graph=[[0,10,15,20],[10,0,35,25],[15,35,0,30],[20,25,30,0]]


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

Hình 8: Hàm main với các test case khác nhau

```
Path complete
[0, 2, 3, 1, 0]
Ans is 14
```

((a)) Kết quả test case 1

```
Path complete
[0, 1, 0]
Ans is 600
```

((b)) Kết quả test case 2

```
Path complete
[0, 2, 1, 0]
Ans is 1000
```

((c)) Kết quả test case 3

```
Path complete
[0, 1, 3, 2, 0]
Ans is 80
```

((d)) Kết quả test case 4

Hình 9: Các kết quả của thuật toán chạy trên các test case khác nhau

2.6 Nhận xét

- Chương trình chỉ chạy được khi từng cặp đỉnh trong đò thị được nối với nhau bởi 1 cạnh.
- Nếu số lượng các đỉnh lớn thì thuật toán sẽ chạy chậm.
- Ngoài Heuristic nearest insertion, chúng ta có thể sử dụng những thuật toán khác để giải quyết bài toán.
- Giải quyết thách thức TSP có thể làm cho chuỗi cung ứng hiệu quả và cắt giảm chi phí logistics. Nói tóm lại, TSP là một vấn đề dễ xác định, nhưng là một vấn đề phức tạp để giải quyết.