Thực hành Nhập môn Trí tuệ Nhân tạo
Tuần 5 – Các thuật toán tìm kiếm (tiếp theo)

Nguyễn Hồng Yến – MSSV: 23280099

Tháng 11, 2025

Mục lục

1 Tổng quan về bài thực hành

1.1 Mục tiêu

Bài thực hành tuần 5 tập trung vào việc áp dụng **thuật toán tìm kiếm có thông tin A* (A-star)** để giải quyết bài toán **người bán hàng TSP (Traveling Salesperson Problem)** với hàm heuristic dựa trên **cây khung nhỏ nhất MST (Minimum Spanning Tree)**.

Đây là bài toán kinh điển trong khoa học máy tính và tối ưu hóa tổ hợp, có nhiều ứng dụng thực tế trong logistics, lập lịch sản xuất, thiết kế mạch điện tử, và nhiều lĩnh vực khác.

1.2 Yêu cầu

1. Hiểu rõ bài toán TSP và các thành phần của thuật toán A*
2. Nắm vững cách xây dựng hàm heuristic admissible dựa trên MST
3. Cài đặt và thực thi chương trình Python
4. Phát hiện và sửa lỗi trong code (nếu có)
5. Phân tích độ phức tạp và hiệu quả của thuật toán
6. Đưa ra nhận xét, đánh giá và hướng cải tiến

2 Lý thuyết nền tảng

2.1 Bài toán TSP (Traveling Salesperson Problem)

2.1.1 Định nghĩa

Cho trước:

- Tập V gồm n thành phố: $V = \{0, 1, 2, \ldots, n-1\}$
- Ma trận khoảng cách $D = [d_{ij}]_{n \times n}$, trong đó d_{ij} là khoảng cách từ thành phố i đến thành phố j

Mục tiêu: Tìm một tour (chu trình Hamilton) π sao cho:

$$\pi = (v_0, v_1, v_2, \ldots, v_{n-1}, v_0) \tag{1}$$

với mỗi thành phố được viếng thăm đúng một lần và tổng chi phí là nhỏ nhất:

$$C(\pi) = \sum_{i=0}^{n-1} d_{v_i, v_{i+1}} + d_{v_{n-1}, v_0} \to \min \tag{2}$$

2.1.2 Đặc điểm và độ phức tạp

- **Loại bài toán:** NP-hard (không tồn tại thuật toán đa thức để tìm nghiệm tối ưu trong trường hợp tổng quát)
- **Không gian nghiệm:** Với n thành phố, có $(n-1)!/2$ tour khác nhau (với đồ thị đối xứng)
- **Tăng trưởng:** Factorial growth - với $n = 20$ có khoảng 10^{17} tour
- **Phương pháp giải:**
  - *Exact algorithms:* Branch-and-bound, Dynamic Programming (Held-Karp)
  - *Approximation algorithms:* Nearest Neighbor, MST-based heuristic
  - *Metaheuristics:* Genetic Algorithm, Simulated Annealing, Ant Colony
  - *Informed search:* A* với heuristic admissible (bài thực hành này)

2.1.3 Ứng dụng thực tế

1. **Logistics và vận tải:**
   - Tối ưu hóa lộ trình giao hàng cho xe tải, shipper
   - Lập kế hoạch thu gom rác thải đô thị
   - Điều phối xe taxi, Grab/Uber
2. **Sản xuất công nghiệp:**
   - Lập lịch khoan PCB (Printed Circuit Board)
   - Điều khiển robot hàn, lắp ráp
   - Tối ưu hóa đường đi của máy CNC
3. **Sinh học và y học:**
   - DNA sequencing - sắp xếp các đoạn gene
   - Phân tích chuỗi protein
4. **Mạng máy tính:**
   - Routing protocols
   - Network design và optimization

2.2 Thuật toán A* (A-star)

2.2.1 Khái niệm cơ bản

A* là thuật toán tìm kiếm có thông tin (informed search) sử dụng hàm đánh giá:

$$f(n) = g(n) + h(n) \tag{3}$$

trong đó:

- $g(n)$: Chi phí thực tế từ nút xuất phát đến nút n (backward cost)
- $h(n)$: Chi phí ước lượng từ nút n đến đích (forward cost - heuristic)
- $f(n)$: Tổng chi phí ước lượng của đường đi qua n

2.2.2 Thuật toán tổng quát

Thuật toán A*

Input: Đồ thị G, nút xuất phát *start*, nút đích *goal*, hàm heuristic $h(n)$
Output: Đường đi ngắn nhất từ *start* đến *goal*
Các bước:

1. Khởi tạo:
   - OPEN = {start} (fringe list)
   - CLOSED = $\emptyset$ (explored set)
   - $g(start) = 0$, $f(start) = h(start)$
2. Lặp cho đến khi OPEN rỗng hoặc tìm thấy goal:
   - Chọn $n \in$ OPEN có $f(n)$ nhỏ nhất
   - Nếu $n = goal$ thì trả về đường đi
   - Chuyển n từ OPEN sang CLOSED
   - Với mỗi nút con n' của n:
     - Tính $g(n') = g(n) + cost(n, n')$
     - Tính $f(n') = g(n') + h(n')$
     - Nếu $n' \notin$ OPEN và $n' \notin$ CLOSED: thêm vào OPEN
     - Nếu $n' \in$ OPEN với f lớn hơn: cập nhật

2.2.3 Tính chất quan trọng

1. **Admissibility:** Nếu $h(n)$ là admissible ($h(n) \leq h^*(n)$ với mọi n), thì A* đảm bảo tìm được nghiệm tối ưu.

2. **Consistency (Monotonicity):** Nếu $h(n) \leq cost(n, n') + h(n')$, thì A* không cần xét lại các nút đã explored.

3. **Optimally efficient:** Không có thuật toán nào khác (sử dụng cùng heuristic) mở rộng ít nút hơn A*.

4. **Độ phức tạp:**

   - Thời gian: $O(b^d)$ trong trường hợp xấu nhất, tốt hơn nhiều với heuristic tốt
   - Không gian: $O(b^d)$ - cần lưu tất cả các nút trong OPEN
   - b: branching factor, d: độ sâu nghiệm

2.2.4 Áp dụng A* cho TSP

Định nghĩa không gian trạng thái:

- **Trạng thái:** Một trạng thái s được biểu diễn bởi:
  - Thành phố hiện tại: $current$
  - Tập các thành phố đã thăm: $visited \subseteq V$
  - Đường đi từ thành phố xuất phát: $path = [v_0, v_1, \ldots, current]$
- **Trạng thái ban đầu:** $s_0 = (0, \{0\}, [0])$ (xuất phát từ thành phố 0)
- **Trạng thái đích:** $s_{goal} = (0, V, [v_0, v_1, \ldots, v_{n-1}, 0])$ (quay về thành phố 0 sau khi thăm hết)
- **Hành động (Actions):** Từ trạng thái $(current, visited, path)$, có thể di chuyển đến bất kỳ thành phố $j \notin visited$
- **Chi phí $g(n)$:** Tổng khoảng cách đã đi:

$$g(n) = \sum_{i=0}^{k-1} d_{path[i],path[i+1]} \tag{4}$$

- **Heuristic $h(n)$:** Ước lượng chi phí còn lại (sẽ trình bày chi tiết ở phần sau)

2.3 Cây khung nhỏ nhất (Minimum Spanning Tree - MST)

2.3.1 Định nghĩa

Cho đồ thị vô hướng liên thông $G = (V, E)$ với hàm trọng số $w : E \to \mathbb{R}^+$.

Cây khung $T = (V, E_T)$ với $E_T \subseteq E$ là một đồ thị con của G thỏa mãn:

1. Là cây (đồ thị không tuần hoàn liên thông)
2. Bao phủ tất cả các đỉnh: $V(T) = V(G)$
3. Có đúng $|V| - 1$ cạnh

Trọng số cây khung:

$$W(T) = \sum_{e \in E_T} w(e) \tag{5}$$

Cây khung nhỏ nhất (MST): Là cây khung có trọng số nhỏ nhất:

$$T^* = \arg \min_{T \text{ là cây khung}} W(T) \tag{6}$$

2.3.2 Tính chất quan trọng

1. **Tính duy nhất:** Nếu tất cả trọng số cạnh khác nhau, MST là duy nhất.
2. **Cut property:** Với mọi cut $(S, V \setminus S)$, cạnh nhẹ nhất bắc cầu qua cut thuộc MST.
3. **Cycle property:** Trong mọi chu trình, cạnh nặng nhất không thuộc MST.
4. **Quan hệ với TSP:**

$$W(MST) \leq L(TSP^*) \leq 2 \cdot W(MST) \tag{7}$$

trong đó $L(TSP^*)$ là độ dài tour TSP tối ưu.

Chứng minh:

- Chặn dưới: Nếu loại bỏ một cạnh khỏi tour TSP, ta được một cây khung $\Rightarrow$ $W(MST) \leq L(TSP^*)$
- Chặn trên: Duyệt MST theo DFS và shortcut $\Rightarrow$ tour $\leq 2W(MST)$

2.3.3 Thuật toán Prim

Thuật toán Prim xây dựng MST bằng cách mở rộng dần cây từ một đỉnh xuất phát.

Thuật toán Prim

Input: Đồ thị $G = (V, E)$, trọng số w

Output: Cây khung nhỏ nhất T

Các bước:

1. Khởi tạo:
   - $T = \emptyset$ (cây rỗng)
   - $S = \{v_0\}$ (tập đỉnh trong cây, bắt đầu từ đỉnh bất kỳ)
   - $key[v] = \infty$ với mọi $v \in V \setminus S$
   - $key[v_0] = 0$
2. Lặp $|V| - 1$ lần:
   - Tìm đỉnh $u \notin S$ có $key[u]$ nhỏ nhất
   - Thêm u vào S
   - Thêm cạnh $(parent[u], u)$ vào T
   - Cập nhật key và $parent$ cho các đỉnh kề với u:

$$\text{Nếu } v \notin S \text{ và } w(u,v) < key[v] \text{ thì } key[v] = w(u,v), parent[v] = u \tag{8}$$

Độ phức tạp:

- Với ma trận kề: $O(V^2)$
- Với min-heap: $O(E \log V)$
- Với Fibonacci heap: $O(E + V \log V)$

Ví dụ minh họa: Xét đồ thị với 4 đỉnh:

	0	1	2	3
0	0	5	2	3
1	5	0	6	3
2	6	2	0	4
3	3	3	4	0

Các bước xây dựng MST:

- Bước 1: Chọn đỉnh 0. Cạnh nhẹ nhất: (0,2) = 2
- Bước 2: Từ {0,2}, cạnh nhẹ nhất: (0,3) = 3

- Bước 3: Từ {0,2,3}, cạnh nhẹ nhất: (3,1) = 3
- MST: {(0,2), (0,3), (3,1)} với trọng số = 8

3 Hàm Heuristic dựa trên MST

3.1 Ý tưởng chính

Giả sử tại nút n trong cây tìm kiếm A*, ta đã thăm một tập con $S \subset V$ các thành phố. Cần ước lượng chi phí còn lại để:

1. Thăm tất cả các thành phố chưa thăm $V \setminus S$
2. Quay về thành phố xuất phát

Chiến lược:

- Xây dựng MST trên tập các thành phố chưa thăm: $MST(V \setminus S)$
- Chi phí để kết nối thành phố hiện tại *current* với các thành phố chưa thăm
- Chi phí để quay về thành phố xuất phát từ các thành phố chưa thăm

3.2 Công thức heuristic

$$h(n) = W(MST_{unvisited}) + \min_{v \in unvisited} d_{current,v} + \min_{v \in unvisited} d_{0,v} \tag{9}$$

trong đó:

- $unvisited = V \setminus visited$: tập thành phố chưa thăm
- $W(MST_{unvisited})$: trọng số MST của các thành phố chưa thăm
- $\min_{v \in unvisited} d_{current,v}$: khoảng cách nhỏ nhất từ thành phố hiện tại đến một thành phố chưa thăm
- $\min_{v \in unvisited} d_{0,v}$: khoảng cách nhỏ nhất từ thành phố xuất phát đến một thành phố chưa thăm

3.3 Chứng minh tính Admissible

Mệnh đề: Hàm heuristic $h(n)$ không bao giờ overestimate chi phí thực tế, tức là:

$$h(n) \leq h^*(n) \quad \forall n \tag{10}$$

với $h^*(n)$ là chi phí tối ưu thực tế từ n đến trạng thái đích.

Chứng minh:

Gọi π^* là tour TSP tối ưu hoàn chỉnh từ nút n.

Bước 1: Tour π^* phải đi qua tất cả các thành phố chưa thăm. Nếu loại bỏ một cạnh khỏi phần tour này, ta được một cây khung trên *unvisited*. Do đó:

$$W(MST_{unvisited}) \leq \text{tổng trọng số các cạnh của } \pi^* \text{ trong } unvisited \tag{11}$$

Bước 2: Tour π^* phải có ít nhất một cạnh từ *current* đến một thành phố chưa thăm, và chi phí cạnh này $\geq \min_{v \in unvisited} d_{current,v}$.

Bước 3: Tour π^* phải có ít nhất một cạnh từ một thành phố chưa thăm về thành phố 0, và chi phí cạnh này $\geq \min_{v \in unvisited} d_{0,v}$.

Kết hợp ba bước trên:

$$h(n) = W(MST_{unvisited}) + \min d_{current,v} + \min d_{0,v} \leq h^*(n) \tag{12}$$

$\Rightarrow$ Hàm heuristic là admissible. $\square$

3.4 Trường hợp đặc biệt

Khi chỉ còn lại thành phố xuất phát chưa thăm:

$$h(n) = d_{current,0} \tag{13}$$

Đây chính là chi phí thực tế để hoàn thành tour.

4 Phân tích chi tiết code

4.1 Cấu trúc tổng thể

Chương trình được tổ chức thành các thành phần chính:

1. **Lớp TreeNode:** Đại diện cho một nút trong cây tìm kiếm A*
2. **Lớp FringeNode:** Đại diện cho một nút trong fringe list (OPEN list)
3. **Lớp Graph:** Cài đặt thuật toán Prim để tính MST
4. **Hàm heuristic:** Tính giá trị $h(n)$ cho mỗi nút
5. **Hàm checkPath:** Kiểm tra đường đi có hoàn chỉnh hay chưa
6. **Hàm startTSP:** Hàm chính thực hiện thuật toán A*

4.2 Chi tiết từng thành phần

4.2.1 Lớp TreeNode

```
class TreeNode(object):
    def __init__(self, c_no, c_id, f_value, h_value, parent_id):
        self.c_no = c_no           # Số thứ tự thành phố (0 đến V-1)
        self.c_id = c_id           # ID duy nhất của nút trong cây
        self.f_value = f_value     # f(n) = g(n) + h(n)
        self.h_value = h_value     # Giá trị heuristic h(n)
        self.parent_id = parent_id # ID của nút cha (để truy vết)
```

Giải thích:

- `c_no`: Lưu thành phố mà nút này đại diện (0, 1, 2, ...)
- `c_id`: Mỗi nút trong cây có một ID duy nhất (key trong tree)
- `f_value`: Tổng chi phí ước lượng $f(n) = g(n) + h(n)$
- `h_value`: Giá trị heuristic (cần lưu riêng để tính $g(n) = f(n) - h(n)$)
- `parent_id`: Con trỏ đến nút cha để có thể truy vết đường đi

4.2.2 Lớp FringeNode

```
class FringeNode(object):
    def __init__(self, c_no, f_value):
        self.f_value = f_value  # Giá trị f(n) để so sánh
        self.c_no = c_no        # Số thứ tự thành phố
```

Giải thích:

- Fringe list (OPEN list) chứa các nút đang chờ được mở rộng
- Chỉ cần lưu `f_value` để so sánh và chọn nút có f nhỏ nhất
- `c_no` để biết nút này đại diện cho thành phố nào

4.2.3 Lớp Graph và thuật toán Prim

Khởi tạo:

```
class Graph():
    def __init__(self, vertices):
        self.V = vertices  # Số đỉnh
```

```
        # Ma trận kề V x V, khởi tạo toàn 0
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
```

Hàm minKey: Tìm đỉnh chưa thuộc MST có key nhỏ nhất

```
def minKey(self, key, mstSet):
    min = sys.maxsize  # Khởi tạo min = vô cùng
    for v in range(self.V):
        # Nếu đỉnh v chưa thuộc MST và key[v] < min
        if key[v] < min and mstSet[v] == False:
            min = key[v]
            min_index = v
    return min_index
```

Giải thích logic:

- `key[v]`: Trọng số cạnh nhỏ nhất từ MST hiện tại đến đỉnh v
- `mstSet[v]`: `True` nếu v đã thuộc MST, `False` nếu chưa
- Hàm trả về đỉnh chưa thuộc MST có `key` nhỏ nhất

Hàm primMST: Xây dựng MST bằng thuật toán Prim

```
def primMST(self, d_temp, t):
    # Mảng key: lưu trọng số nhỏ nhất đến mỗi đỉnh
    key = [sys.maxsize] * self.V
    parent = [None] * self.V  # Lưu cây MST
    key[0] = 0                # Bắt đầu từ đỉnh 0
    mstSet = [False] * self.V # Đánh dấu đỉnh đã thuộc MST
    parent[0] = -1            # Gốc không có cha

    # Lặp V lần để thêm V đỉnh vào MST
    for c in range(self.V):
        # Chọn đỉnh u chưa thuộc MST có key nhỏ nhất
        u = self.minKey(key, mstSet)
        mstSet[u] = True  # Đánh dấu u đã thuộc MST

        # Cập nhật key cho các đỉnh kề với u
        for v in range(self.V):
            # Nếu có cạnh (u,v), v chưa thuộc MST,
            # và trọng số (u,v) < 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)
```

Giải thích chi tiết:

1. **Khởi tạo:**
   - `key[v] = ∞`: Chưa biết cạnh nào nối MST với v
   - `key[0] = 0`: Bắt đầu từ đỉnh 0
   - `mstSet[v] = False`: Tất cả đỉnh chưa thuộc MST
   - `parent[v] = None`: Chưa xác định cha
2. **Vòng lặp chính:** Lặp V lần, mỗi lần thêm 1 đỉnh vào MST
   - Chọn đỉnh u chưa thuộc MST có `key[u]` nhỏ nhất
   - Đánh dấu u đã thuộc MST
   - Cập nhật `key` cho các đỉnh kề: nếu cạnh (u, v) rẻ hơn `key[v]` hiện tại thì cập nhật
3. **Kết quả:** Mảng `parent` biểu diễn cây MST

Hàm printMST: Tính tổng trọng số MST và các cạnh kết nối

```
def printMST(self, parent, d_temp, t):
    sum_weight = 0
    min1 = 10000  # Min từ thành phố 0 đến unvisited
    min2 = 10000  # Min từ thành phố t (hiện tại) đến unvisited

    # Tạo reverse dictionary để ánh xạ ngược
    r_temp = {}
    for k in d_temp:
        r_temp[d_temp[k]] = k

    # Tính tổng trọng số MST và tìm min
    for i in range(1, self.V):
        sum_weight += self.graph[i][parent[i]]

        # Tìm cạnh nhỏ nhất từ thành phố 0
```

```
        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]]]

        # Tìm cạnh nhỏ nhất từ thành phố hiện tại t
        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]]]

    # Trả về h(n) = W(MST) + min1 + min2
    return (sum_weight + min1 + min2) % 10000
```

Giải thích:

- `sum_weight`: Tổng trọng số của MST = $W(MST_{unvisited})$
- `min1`: Cạnh nhỏ nhất từ thành phố xuất phát (0) đến các thành phố chưa thăm
- `min2`: Cạnh nhỏ nhất từ thành phố hiện tại (t) đến các thành phố chưa thăm
- `d_temp`: Dictionary ánh xạ từ index thành phố gốc sang index trong đồ thị con
- `r_temp`: Reverse mapping để lấy lại index gốc
- Kết quả: $h(n) = sum\_weight + min1 + min2$ (đúng theo công thức heuristic)

4.2.4 Hàm heuristic

Đây là hàm quan trọng nhất, tính giá trị $h(n)$ cho mỗi nút trong cây tìm kiếm.

```
def heuristic(tree, p_id, t, V, graph):
    visited = set()  # Tập các thành phố đã thăm
    visited.add(0)   # Thành phố xuất phát luôn đã thăm
    visited.add(t)   # Thành phố hiện tại đang xét

    # Truy ngược lên cây để tìm tất cả thành phố đã thăm
    if(p_id != -1):
        tnode = tree.get_node(str(p_id))
        # Đi ngược lên đến gốc (c_id = 1)
        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  # Số thành phố chưa thăm

    if (num != 0):
        # Còn thành phố chưa thăm: tính MST
        g = Graph(num)
        d_temp = {}  # Mapping: index gốc -> index mới
        key = 0

        # Tạo mapping cho các thành phố chưa thăm
        for i in range(V):
            if(i not in visited):
                d_temp[i] = key
                key = key + 1

        # Xây dựng đồ thị con gồm các thành phố chưa thăm
        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]

        # Tính MST và trả về heuristic
        mst_weight = g.primMST(d_temp, t)
        return mst_weight
    else:
        # Không còn thành phố chưa thăm: chỉ cần quay về 0
        return graph[t][0]
```

Phân tích từng bước:

1. **Thu thập thành phố đã thăm:**
   - Khởi tạo `visited` với thành phố 0 và t
   - Truy ngược từ nút cha (`p_id`) lên đến gốc
   - Mỗi nút trên đường đi tương ứng một thành phố đã thăm
2. **Tính số thành phố chưa thăm:**
   - `num = V - len(visited)`
   - Nếu `num = 0`: tất cả đã thăm, chỉ cần quay về

- Nếu `num > 0`: cần tính heuristic cho các thành phố chưa thăm

3. **Xây dựng đồ thị con:**
   - Tạo đồ thị G' chỉ gồm các thành phố chưa thăm
   - Dictionary `d_temp`: ánh xạ index gốc sang index mới (0, 1, 2, ...)
   - Copy khoảng cách giữa các thành phố chưa thăm vào `g.graph`
4. **Gọi Prim's algorithm:**
   - Tính MST của đồ thị con
   - Hàm `primMST` trả về: $W(MST) + min_1 + min_2$
   - Đây chính là giá trị $h(n)$ cần tìm

Ví dụ minh họa:
Giả sử $V = 5$, đã thăm {0, 2, 3}, đang ở thành phố 3, chưa thăm {1, 4}.

- `visited = {0, 2, 3}`
- `unvisited = {1, 4}`
- `d_temp = {1: 0, 4: 1}` (ánh xạ sang index 0, 1)
- Xây dựng đồ thị con 2x2 với các cạnh giữa thành phố 1 và 4
- Tính MST (chỉ có 1 cạnh nối 1-4)
- Tìm `min1` = min(d[0][1], d[0][4])
- Tìm `min2` = min(d[3][1], d[3][4])
- Trả về: MST + min1 + min2

4.2.5 Hàm checkPath

Hàm này kiểm tra xem đường đi từ nút gốc đến nút hiện tại có phải là một tour TSP hoàn chỉnh hay không.

```
def checkPath(tree, toExpand, V):
    tnode = tree.get_node(str(toExpand.c_id))
    list1 = list()  # Lưu đường đi

    # Trường hợp nút gốc
    if(tnode.data.c_id == 1):
        return 0
```

```
else:
    depth = tree.depth(tnode)  # Độ sâu của nút
    s = set()  # Tập các thành phố đã thăm (không kể 0)

    # Truy ngược lên gốc, thu thập đường đi
    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)  # Thêm thành phố xuất phát

    # Kiểm tra điều kiện tour hoàn chỉnh
    if(depth == V and len(s) == V-1 and list1[0] == 0):
        print("Path complete")
        list1.reverse()
        print(list1)
        return 1
    else:
        return 0
```

LỖI PHÁT HIỆN VÀ SỬA:

Lỗi trong code gốc

Dòng lỗi (ban đầu):

```
if(depth == V and len(s) == V and list1[0] == 0):
```

Phân tích lỗi:

- Điều kiện `len(s) == V` là **SAI**
- Tập hợp `s` chỉ chứa các thành phố **đã thăm trên đường đi**, không bao gồm thành phố xuất phát ban đầu (thành phố 0 ở gốc cây)
- Ví dụ: với $V = 4$, tour hoàn chỉnh là $[0, 2, 3, 1, 0]$
  - Khi truy ngược từ nút cuối (thành phố 0), ta thu được: `s = {0, 1, 3, 2}`... **KHÔNG!**
  - Thực tế: khi truy ngược, ta bắt đầu từ nút thành phố 0 (ở lá), đi ngược lên, thu được `s = {1, 3, 2}` (không đếm thành phố 0 ở gốc)
  - Do đó `len(s) = 3 = V - 1`, không phải `len(s) = V`

Code sửa đúng:

```
if(depth == V and len(s) == V-1 and list1[0] == 0):
```

Giải thích điều kiện đúng:

1. `depth == V`: Độ sâu của nút $= V$
   - Nghĩa là đã đi qua V bước từ gốc (đếm cả gốc)
   - Tương ứng với việc thăm V thành phố (bao gồm cả việc quay về thành phố 0)
2. `len(s) == V-1`: Tập `s` chứa $V - 1$ thành phố khác nhau
   - Là các thành phố 1, 2, ..., $V - 1$ (hoặc bất kỳ $V - 1$ thành phố nào khác 0)
   - Kết hợp với thành phố 0 (gốc), ta có đủ V thành phố
3. `list1[0] == 0`: Nút hiện tại đang mở rộng là thành phố 0
   - Nghĩa là đã quay về thành phố xuất phát
   - Hoàn thành chu trình Hamilton

Ví dụ cụ thể với V=4:

- Tour: $0 \to 2 \to 3 \to 1 \to 0$

- Cây tìm kiếm: Gốc (0) → Nút (2) → Nút (3) → Nút (1) → Nút (0)
- Khi mở rộng nút (0) cuối cùng:
  - `depth` = 4 (có 5 nút, nhưng depth của nút lá = 4)
  - Truy ngược: thu được `s = {1, 3, 2}`, `len(s) = 3`
  - `list1 = [0, 1, 3, 2, 0]` → `list1[0] = 0`
  - Điều kiện: `depth == 4 and len(s) == 3 and list1[0] == 0`

4.2.6 Hàm startTSP

Đây là hàm chính thực hiện thuật toán A*.

```
def startTSP(graph, tree, V):
    goalState = 0  # Flag để dừng khi tìm thấy goal
    toExpand = TreeNode(0,0,0,0,0)  # Nút sẽ được mở rộng
    key = 1  # ID duy nhất cho mỗi nút

    # Tính heuristic cho nút gốc (thành phố 0)
    heu = heuristic(tree, -1, 0, V, graph)

    # Tạo nút gốc trong cây
    tree.create_node("1", "1",
                     data=TreeNode(0, 1, heu, heu, -1))

    # Fringe list (OPEN list) - dictionary
    fringe_list = {}
    fringe_list[key] = FringeNode(0, heu)
    key = key + 1

    # Vòng lặp chính của A*
    while(goalState == 0):
        minf = sys.maxsize

        # Tìm nút có f(n) nhỏ nhất trong 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
```

```
        # Lấy g(n) và h(n) của nút được chọn
        h = tree.get_node(str(toExpand.c_id)).data.h_value
        val = toExpand.f_value - h  # g(n) = f(n) - h(n)

        # Kiểm tra đường đi có hoàn chỉnh không
        path = checkPath(tree, toExpand, V)

        # Nếu quay về thành phố 0 và tour hoàn chỉnh
        if(toExpand.c_no == 0 and path == 1):
            goalState = 1
            cost = toExpand.f_value
        else:
            # Xóa nút khỏi fringe list (chuyển sang CLOSED)
            del fringe_list[toExpand.c_id]

            # Mở rộng: sinh các nút con
            j = 0
            while(j < V):
                if(j != toExpand.c_no):
                    # Tính heuristic cho nút con j
                    h = heuristic(tree, toExpand.c_id, j, V, graph)

                    # f(child) = g(parent) + cost(parent->child) + h(child)
                    f_val = val + graph[j][toExpand.c_no] + h

                    # Thêm vào fringe list
                    fringe_list[key] = FringeNode(j, f_val)

                    # Thêm vào cây
                    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
```

Phân tích chi tiết thuật toán:

1. Khởi tạo:

- Tính $h(0)$ cho thành phố xuất phát (MST của tất cả V thành phố + chi phí kết nối)

- Tạo nút gốc với $f(0) = 0 + h(0) = h(0)$ (vì $g(0) = 0$)
- Thêm nút gốc vào fringe list

2. Vòng lặp chính:

1. **Chọn nút tốt nhất:**
   - Duyệt qua tất cả nút trong fringe list
   - Chọn nút có $f(n)$ nhỏ nhất → `toExpand`
   - Đây là đặc trưng của A*: luôn mở rộng nút có f nhỏ nhất
2. **Tính $g(n)$:**
   - $g(n) = f(n) - h(n)$
   - `val = toExpand.f_value - h`
   - Đây là chi phí thực tế đã đi từ gốc đến nút hiện tại
3. **Kiểm tra goal:**
   - Gọi `checkPath()` để xem tour đã hoàn chỉnh chưa
   - Nếu `toExpand.c_no == 0` và `path == 1`: đã tìm thấy nghiệm
   - Dừng và trả về `cost = f(goal)`
4. **Mở rộng nút:**
   - Xóa nút khỏi fringe list (chuyển sang CLOSED)
   - Với mỗi thành phố $j \neq$ thành phố hiện tại:
     - Tính $h(j)$ bằng hàm heuristic
     - Tính $f(j) = g(toExpand) + d(toExpand, j) + h(j)$
     - Thêm nút con vào fringe list và cây tìm kiếm

Đặc điểm quan trọng:

- **Không kiểm tra trùng lặp:** Code này không kiểm tra nếu nút đã tồn tại trong fringe list với f lớn hơn. Điều này có thể dẫn đến nhiều nút trùng nhau trong fringe list, tốn bộ nhớ.
- **Mở rộng tất cả con:** Mỗi lần mở rộng, sinh ra $V - 1$ nút con (tất cả thành phố khác thành phố hiện tại). Tuy nhiên, nhiều nút có thể vi phạm ràng buộc (đã thăm rồi).
- **Kiểm tra goal khi mở rộng:** Theo chuẩn A*, kiểm tra goal khi lấy nút ra khỏi fringe list (không phải khi sinh nút), đảm bảo tính tối ưu.

4.3 Minh họa thực thi với ví dụ cụ thể

Dữ liệu đầu vào:

$$V = 4, \quad graph = \begin{bmatrix} 0 & 5 & 2 & 3 \\ 5 & 0 & 6 & 3 \\ 2 & 6 & 0 & 4 \\ 3 & 3 & 4 & 0 \end{bmatrix}$$

Bước 1: Khởi tạo

- Tính $h(0)$: MST của {0,1,2,3} + min đến 0 + min từ 0
- MST: (0-2)=2, (0-3)=3, (3-1)=3 $\rightarrow W(MST) = 8$
- $h(0) = 8 + 0 + 0 = 8$ (vì đã ở 0)
- Nút gốc: `TreeNode(c_no=0, c_id=1, f=8, h=8, parent=-1)`
- Fringe: {`1: FringeNode(0, 8)`}

Bước 2: Mở rộng nút (0)

- Chọn nút 1 (f=8, c_no=0)
- $g(0) = 8 - 8 = 0$
- Sinh 3 nút con: (1), (2), (3)
- Tính h và f cho từng nút:
  - Nút (1): visited={0,1}, unvisited={2,3}
    - MST({2,3}) = 4
    - min từ 0: min(2,3) = 2
    - min từ 1: min(6,3) = 3
    - $h(1) = 4 + 2 + 3 = 9$
    - $f(1) = 0 + 5 + 9 = 14$
  - Nút (2): visited={0,2}, unvisited={1,3}
    - MST({1,3}) = 3
    - $h(2) = 3 + 3 + 3 = 9$
    - $f(2) = 0 + 2 + 9 = 11$
  - Nút (3): visited={0,3}, unvisited={1,2}
    - MST({1,2}) = 6
    - $h(3) = 6 + 2 + 3 = 11$

* $f(3) = 0 + 3 + 11 = 14$

- Fringe: {`2:(1,14), 3:(2,11), 4:(3,14)`}

Bước 3: Chọn nút có f nhỏ nhất

- Chọn nút 3 (c_no=2, f=11) - thành phố 2
- Mở rộng: sinh các nút con từ thành phố 2
- Tiếp tục quá trình tương tự...

Kết quả cuối cùng:

Output

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

Xác minh nghiệm:

- Tour: $0 \to 2 \to 3 \to 1 \to 0$
- Chi phí: $d_{0,2} + d_{2,3} + d_{3,1} + d_{1,0} = 2 + 4 + 3 + 5 = 14$
- Đây là tour tối ưu cho bài toán TSP 4 thành phố này

5 Phân tích độ phức tạp

5.1 Độ phức tạp thời gian

Phân tích từng thành phần:

1. **Hàm heuristic:**
   - Thu thập thành phố đã thăm: $O(V)$ (truy ngược cây)
   - Xây dựng đồ thị con: $O(V^2)$ (duyệt ma trận)
   - Thuật toán Prim: $O(V^2)$ (với ma trận kề)
   - Tổng: $O(V^2)$ cho mỗi lần gọi
2. **Số lần gọi heuristic:**
   - Trong trường hợp xấu nhất, A* mở rộng $O((V-1)!)$ nút
   - Mỗi nút mở rộng sinh ra $O(V)$ nút con

- Tổng số lần gọi heuristic: $O(V \cdot (V-1)!)$

3. **Chọn nút có f nhỏ nhất:**
   - Duyệt qua fringe list: $O(|fringe|)$
   - Trong trường hợp xấu nhất: $O(V!)$
4. **Tổng độ phức tạp:**

$$T(V) = O(V! \cdot V^2) \tag{14}$$

So sánh với các phương pháp khác:

Phương pháp	**Độ phức tạp**	**Tối ưu?**
Brute Force	$O((V-1)!)$	Có
Dynamic Programming (Held-Karp)	$O(V^2 \cdot 2^V)$	Có
A* với MST heuristic	$O(V! \cdot V^2)$	Có
Nearest Neighbor	$O(V^2)$	Không
Genetic Algorithm	$O(g \cdot p \cdot V^2)$	Không

Bảng 1: So sánh độ phức tạp các thuật toán TSP

Nhận xét:

- A* với MST heuristic vẫn có độ phức tạp exponential
- Tuy nhiên, trong thực tế, với heuristic tốt, A* cắt tỉa được nhiều nhánh không cần thiết
- Hiệu quả hơn brute force nhưng vẫn chậm với $V > 15$

5.2 Độ phức tạp không gian

- **Cây tìm kiếm:** $O(V!)$ nút trong trường hợp xấu nhất
- **Fringe list:** $O(V!)$ nút
- **Ma trận khoảng cách:** $O(V^2)$
- **Tổng:** $S(V) = O(V!)$

Vấn đề bộ nhớ:

- Với $V = 15$: cần lưu khoảng 10^{12} nút $\rightarrow$ không khả thi
- Cần cải tiến: sử dụng IDA* (Iterative Deepening A*) để giảm bộ nhớ

6 Kết quả thực thi và test cases

6.1 Test case 1: Đồ thị 4 đỉnh (trong đề bài)

Input:

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

Output:

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

Xác minh:

- Tour: $0 \xrightarrow{2} 2 \xrightarrow{4} 3 \xrightarrow{3} 1 \xrightarrow{5} 0$
- Tổng: $2 + 4 + 3 + 5 = 14$
- Các tour khác:
  - $0 \to 1 \to 2 \to 3 \to 0$: $5 + 6 + 4 + 3 = 18$
  - $0 \to 1 \to 3 \to 2 \to 0$: $5 + 3 + 4 + 2 = 14$ (cũng tối ưu)
  - $0 \to 3 \to 1 \to 2 \to 0$: $3 + 3 + 6 + 2 = 14$ (cũng tối ưu)
- Nghiệm tối ưu: 14

6.2 Test case 2: Đồ thị đối xứng đơn giản

Input:

```
V = 3
graph = [[0,1,2],
         [1,0,3],
         [2,3,0]]
```

Kết quả mong đợi:

- Tour: $0 \to 1 \to 2 \to 0$ hoặc $0 \to 2 \to 1 \to 0$
- Chi phí: $1 + 3 + 2 = 6$

6.3 Test case 3: Ma trận khoảng cách Euclidean

Các điểm trong mặt phẳng:

- $A(0,0)$, $B(3,0)$, $C(3,4)$, $D(0,4)$
- Ma trận khoảng cách Euclidean:

$$graph = \begin{bmatrix} 0 & 3 & 5 & 4 \\ 3 & 0 & 4 & 5 \\ 5 & 4 & 0 & 3 \\ 4 & 5 & 3 & 0 \end{bmatrix}$$

Tour tối ưu: $0 \to 1 \to 2 \to 3 \to 0$ với chi phí $= 3 + 4 + 3 + 4 = 14$

7 Đánh giá chi tiết

7.1 Ưu điểm

1. **Đảm bảo tính tối ưu:**
   - Hàm heuristic MST là admissible ($h(n) \leq h^*(n)$)
   - A* với heuristic admissible đảm bảo tìm được nghiệm tối ưu
   - Đã chứng minh toán học rõ ràng
2. **Heuristic mạnh:**
   - MST cung cấp ước lượng sát với chi phí thực tế
   - Giảm số lượng nút cần mở rộng đáng kể so với tìm kiếm mù
   - Khai thác được cấu trúc đồ thị (tính liên thông, trọng số cạnh)
3. **Dễ hiểu và cài đặt:**
   - Thuật toán Prim đơn giản, ổn định
   - Code có cấu trúc rõ ràng, dễ maintain
   - Sử dụng thư viện `treelib` để quản lý cây hiệu quả
4. **Tính tổng quát:**
   - Áp dụng được cho mọi loại đồ thị (đối xứng, bất đối xứng)
   - Không yêu cầu điều kiện đặc biệt (tam giác, metric)
   - Dễ dàng mở rộng cho các biến thể TSP (với cửa sổ thời gian, ràng buộc dung lượng, ...)

7.2 Nhược điểm

1. **Độ phức tạp exponential:**
   - Thời gian: $O(V! \cdot V^2)$ trong trường hợp xấu nhất
   - Không gian: $O(V!)$ - phải lưu toàn bộ cây và fringe list
   - Chỉ khả thi với $V \leq 15$ thành phố
   - Với $V = 20$: thời gian chạy có thể lên đến hàng ngày
2. **Không tận dụng symmetry:**
   - Với đồ thị đối xứng, tour $0 \to 1 \to 2 \to 0$ và $0 \to 2 \to 1 \to 0$ là như nhau
   - Code hiện tại xét cả hai trường hợp → lãng phí
3. **Không kiểm tra visited trong successor:**
   - Khi mở rộng nút, sinh ra tất cả $V - 1$ nút con
   - Nhiều nút con vi phạm ràng buộc (thành phố đã thăm)
   - Nên kiểm tra và chỉ sinh các nút con hợp lệ
4. **Không có pruning:**
   - Không cắt tỉa các nhánh có $f(n) > f_{best}$ (best solution found so far)
   - Có thể dùng branch-and-bound để cải thiện
5. **Fringe list không tối ưu:**
   - Sử dụng dictionary, tìm min mất $O(|fringe|)$
   - Nên dùng priority queue (heap) để tìm min trong $O(\log |fringe|)$
6. **Lỗi logic trong checkPath:**
   - Điều kiện kiểm tra tour hoàn chỉnh ban đầu bị sai
   - Cần sửa `len(s) == V` thành `len(s) == V-1`

7.3 So sánh với các phương pháp khác

Tiêu chí	**A* + MST**	**Dynamic Programming**	**Genetic Algorithm**
Tối ưu	Có	Có	Không đảm bảo
Độ phức tạp	$O(V! \cdot V^2)$	$O(V^2 \cdot 2^V)$	$O(g \cdot p \cdot V^2)$
Bộ nhớ	$O(V!)$	$O(V \cdot 2^V)$	$O(p \cdot V)$
Quy mô khả thi	$V \leq 15$	$V \leq 25$	$V > 100$
Dễ cài đặt	Trung bình	Khó	Dễ
Chất lượng nghiệm	Tối ưu	Tối ưu	Gần tối ưu (95-98%)

Bảng 2: So sánh A* với các phương pháp khác

7.4 Nhận xét tổng kết

Bài thực hành tuần 5 đã cung cấp một cái nhìn toàn diện về việc áp dụng thuật toán tìm kiếm có thông tin vào bài toán tối ưu hóa tổ hợp. A* với heuristic MST là một phương pháp elegant, kết hợp giữa lý thuyết đồ thị và tìm kiếm heuristic.

Những điểm mạnh:

- Đảm bảo tìm được nghiệm tối ưu nhờ heuristic admissible
- Heuristic MST mạnh, khai thác được cấu trúc bài toán
- Code dễ hiểu, cấu trúc rõ ràng

Những điểm cần cải thiện:

- Độ phức tạp exponential, chỉ khả thi với V nhỏ
- Cần tối ưu hóa code (priority queue, pruning)
- Có lỗi logic cần sửa

Kết luận:

"Không có thuật toán hoàn hảo cho mọi bài toán. Việc chọn phương pháp phù hợp phụ thuộc vào yêu cầu cụ thể: cần nghiệm tối ưu hay nghiệm gần đủ tốt? Có giới hạn về thời gian hay bộ nhớ? Quy mô bài toán như thế nào?"

Với TSP quy mô nhỏ ($V \leq 15$) và cần nghiệm tối ưu, A* + MST là lựa chọn tốt. Với quy mô lớn hơn, nên xem xét DP, metaheuristics, hoặc approximation algorithms.