ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

BÀI TẬP LỚN
CẤU TRÚC RỜI RẠC CHO KHOA HỌC MÁY TÍNH - CO1007

THE TRAVELLING SALEMAN PROBLEM

GVHD: PGS. TS. Lê Hồng Trang
Ths. Trần Hồng Tài
Sinh viên: Trần Trung Nghĩa
MSSV: 2412278
Lớp: L02
Học kỳ: 242

Thành phố Hồ Chí Minh, ngày 22 tháng 06 năm 2025

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

BÀI TẬP LỚN
CẤU TRÚC RỜI RẠC CHO KHOA HỌC MÁY TÍNH - CO1007

THE TRAVELLING SALEMAN PROBLEM

GVHD: PGS. TS. Lê Hồng Trang
Ths. Trần Hồng Tài
Sinh viên: Trần Trung Nghĩa
MSSV: 2412278
Lớp: L02
Học kỳ: 242

Thành phố Hồ Chí Minh, ngày 22 tháng 06 năm 2025

Mục lục

Lời nói đầu

Em xin gửi lời chào trân trọng đến thầy Trần Hồng Tài và thầy Lê Hồng Trang, người đã giao phó em thực hiện bài tập này. Bài báo cáo này được thực hiện với mục đích nêu rõ phương pháp giải bài toán Người bán hàng (TSP – Traveling Salesman Problem) – một bài toán quan trọng trong tối ưu hóa và khoa học máy tính. Thông qua việc phân tích lý thuyết, triển khai các thuật toán,em hy vọng làm rõ cách thức hoạt động, ưu nhược điểm của thuật toán.

Quá trình thực hiện bài báo cáo không chỉ giúp em củng cố kiến thức về thuật toán đồ thị, lập trình và phân tích độ phức tạp mà còn rèn luyện kỹ năng thực hành, tư duy phân tích dữ liệu và khả năng trình bày vấn đề một cách khoa học. Những kỹ năng này sẽ là hành trang quý báu, hỗ trợ em trong các nghiên cứu chuyên sâu và ứng dụng thực tế sau này.

Ngoài ra, em xin chân thành cảm ơn thầy Lê Hồng Trang đã tận tình hướng dẫn và hỗ trợ em trong suốt quá trình học tập và thực hiện bài tập. Nếu bài báo cáo còn thiếu sót, em kính mong nhận được những góp ý quý báu từ hai thầy để hoàn thiện hơn. Một lần nữa, em xin chân thành cảm ơn!

Chương I

GIỚI THIỆU CHUNG VỀ BÀI TOÁN NGƯỜI GIAO HÀNG

Bài toán Người bán hàng (Travelling Salesman Problem – TSP) là một bài toán tối ưu hóa kinh điển trong lý thuyết đồ thị và khoa học máy tính. Cho trước một tập n các điểm (đỉnh) và ma trận khoảng cách (hoặc chi phí) giữa mỗi cặp đỉnh, mục tiêu của TSP là tìm một chu trình vô hướng hoặc có hướng đi qua tất cả các đỉnh đúng một lần rồi quay về điểm xuất phát, sao cho tổng độ dài (hoặc tổng chi phí) của chu trình là nhỏ nhất.

1. Định nghĩa chính xác

Cho đồ thị đầy đủ $G = (V,E)$ với $V = \{v_1, v_2, \ldots, v_n\}$ và hàm trọng số $d(v_i, v_j) \geq 0$ cho mọi cặp đỉnh v_i, v_j.

Mục tiêu của TSP là tìm một hoán vị $\pi : \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$ sao cho chu trình: $v_{\pi(1)} \to v_{\pi(2)} \to \cdots \to v_{\pi(n)} \to v_{\pi(1)}$ có tổng trọng số:

$$\sum_{i=1}^{n-1} d\big(v_{\pi(i)}, v_{\pi(i+1)}\big) + d\big(v_{\pi(n)}, v_{\pi(1)}\big)$$

là nhỏ nhất.

2. Độ phức tạp

Bài toán TSP thuộc lớp NP-hard, tức không tồn tại thuật toán đa thức được biết để giải chính xác cho mọi đầu vào. Số hoán vị của n đỉnh là $\dfrac{(n-1)!}{2}$ trong trường hợp đồ thị vô hướng (vì chu trình và chu trình ngược được coi là tương tự). Vì thế, khi n tăng

lên, độ phức tạp tăng rất nhanh.

- Các thuật toán chính xác tiêu biểu:
  1. **Held–Karp (Quy hoạch động):** Độ phức tạp $O(n^2 2^n)$. Thường chỉ khả thi với $n \leq\sim 20$.
  2. **Nhánh và cận (Branch-and-Bound):** Xây dựng dần chu trình, cắt bỏ các nhánh không thể đạt chi phí tối ưu. Hiệu quả hơn trong một số trường hợp, nhưng vẫn rơi vào thời gian mũ khi n lớn.
- Với n lớn (hàng trăm, hàng nghìn), người ta thường dùng các giải pháp gần đúng (*heuristic*) hoặc *meta-heuristic*, chất lượng thường gần tối ưu.:
  1. **Nearest Neighbor:** Chọn đỉnh gần nhất chưa thăm.
  2. **Christofides:** Kết hợp cây khung nhỏ nhất và đối ngẫu Eulerian, đảm bảo sai số ≤ 1.5.
  3. **Simulated Annealing, Genetic Algorithms, Ant Colony Optimization, …**

3. Ứng dụng thực tiễn

- **Lập lịch và logistics:** Xác định lộ trình lái xe giao hàng, thu gom rác, bảo trì đường dây...
- **Gia công cơ khí, in ấn:** Tối ưu đường di chuyển của mũi khoan, đầu cắt CNC đi qua nhiều điểm.
- **Sơ đồ mạch điện:** Lên lịch cho đầu hàn (soldering head) trong sản xuất linh kiện điện tử.
- **Thu thập dữ liệu di động:** Quét bản đồ, thu thập cảm biến phân tán.

Tóm lại

Bài toán Người bán hàng TSP là một thách thức lớn về tối ưu hóa với hàng loạt ứng dụng công nghiệp và nghiên cứu. Việc lựa chọn thuật toán phụ thuộc vào quy mô n và yêu cầu về độ chính xác. Đồng thời, TSP cũng là cơ sở để phát triển nhiều phương pháp heuristic, meta-heuristic và tối ưu hóa trong thực tế.

Chương II

HƯỚNG TIẾP CẬN VÀ THUẬT TOÁN BELLMAN-HELD-KARP CHI TIẾT

Bài làm của em dựa trên việc sử dụng thuật toán Bellman-Held-Karp (sử dụng quy hoạch động). Tuy nhiên, như đã nói ở trên, thuật toán Bellman-Held-Karp chỉ có thể áp dụng với đồ thị có ≤ 20 đỉnh, đưa ra kết quả tối ưu trong khoảng thời gian và tài nguyên chấp nhận được. Còn đối với các đồ thị ≥ 25 đỉnh, việc đó tốn quá nhiều thời gian (không thể áp dụng trong thực tế với những đồ thị lên tới hàng nghìn đỉnh). Chính vì thế, để xử lí những đồ thị đó, em sẽ sử dụng tới thuật toán meta-heuristic Ant colony optimization ở phần sau.

1. Hướng tiếp cận thuật toán Bellman-Held-Karp

Khi giải quyết một bài toán liên quan đến việc chọn một tổ hợp tối ưu, việc tiếp cận đơn giản và tự nhiên nhất chính là vét cạn (Brute force). Ý tưởng sử dụng thuật toán vét cạn:

Ta sẽ gọi trạng thái của người giao hàng hiện tại là:

$$\mathbf{s}(\mathbf{v}, \mathbf{S})$$

với:

- **v**: là thành phố hiện tại mà người giao hàng đang ở.
- **S**: là tập hợp chứa những thành phố còn lại mà người giao hàng cần phải đi tới và quay trở về lại điểm bắt đầu.

- $\mathbf{s}(\mathbf{v},\mathbf{S})$: là khoảng cách mà người giao hàng phải đi cho tới khi hoàn thành chu trình.

Ví dụ: $s(0,\{1;2;3\})$ có nghĩa rằng hiện tại người giao hàng đang ở thành phố 0, cần phải đi tiếp tới thành phố $1,2,3$ và quay về điểm bắt đầu và $s(0,\{1;2;3\})$ là chi phí đi quãng đường ấy

Từ trạng thái hiện tại $s(v,S)$, ta cần phải lựa chọn 1 thành phố v_{next} trong tập hợp S để tiếp tục di chuyển tới, khi đó, trạng thái tiếp theo sẽ là:

$$s(v_{next}, S\backslash\{v_{next}\})$$

Với mỗi bước duy chuyển tới 1 thành phố mới, ta phải cộng thêm khoảng cách từ thành phố ban đầu tới thành phố mới, kí hiệu là: $G[v][v_{next}]$. Khi đó, khoảng cách cần đi tới hết chu trình sẽ bằng:

$$s(v,S) = s(v_{next}, S\backslash\{v_{next}\}) + G[v][v_{next}]$$

Với ý tưởng đó, ta sẽ liệt kê toàn bộ các trạng thái có thể đi được từ khi bắt đầu cho tới khi tập $S = \emptyset$.

*** Input bài toán:**

- **edge[][3]**: Một mảng 2D với:
  - **edge[i][0]**: chứa đỉnh xuất phát
  - **edge[i][1]**: chứa đỉnh di chuyển tới
  - **edge[i][3]**: chứa chi phí di chuyển từ đỉnh xuất phát tới đỉnh cần đến.
- **numberOfEdges**: Số lượng cạnh có trong đồ thị
- **star_vertex**: Đỉnh bắt đầu chu trình

*** Output bài toán:**

- **route:** Lộ trình của chu trình ít chi phí nhất
- **min_cost**: Chi phí nhỏ nhất của lộ trình

Để thuận tiện trong quá trình tính toán, ta chuyển đổi input edge list thành một adjacency matrix G, chứa chi phí di chuyển giữa các thành phố với nhau, với G[i][j] là chi phí đi từ thành phố i tới thành phố j

Ví dụ minh họa:

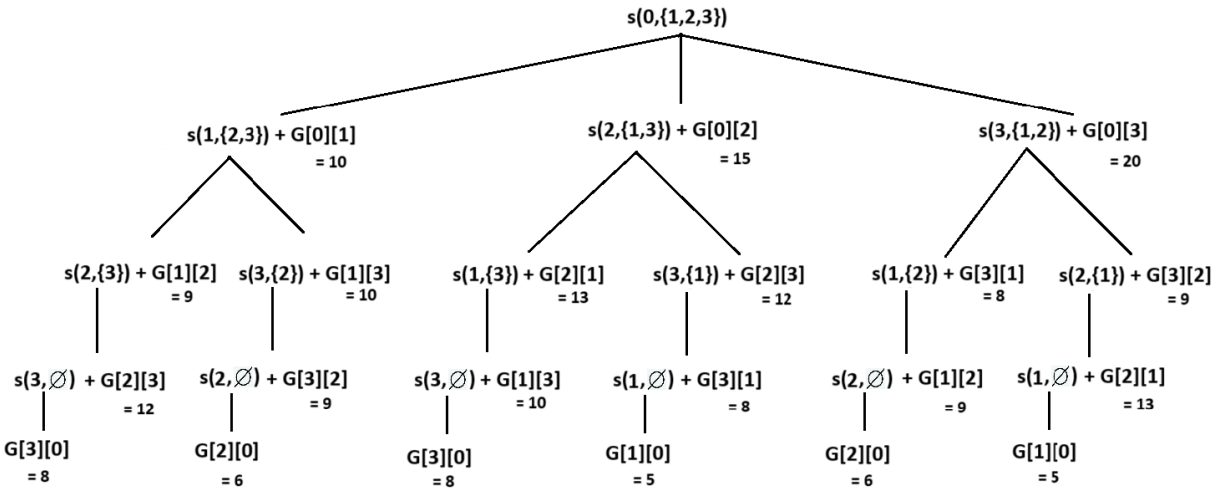
Sau khi biến đổi, ta có ma trận:

$$G = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

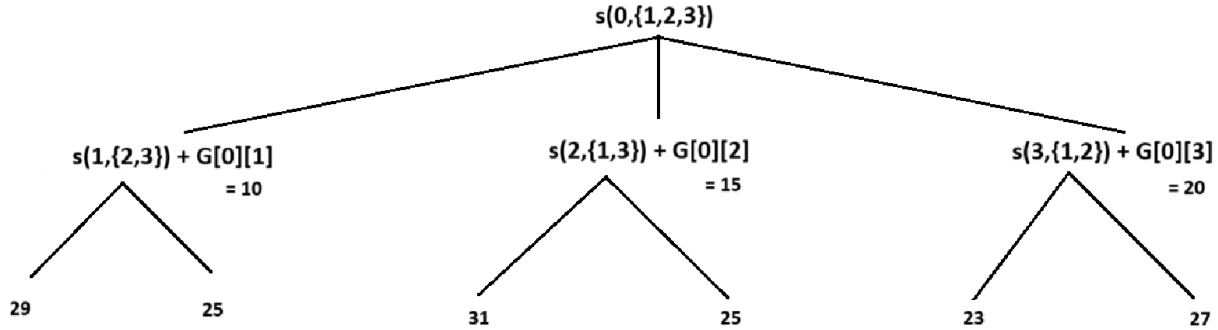
với start_vertex = 0, tìm kiếm chu trình với chi phí ít nhất, đi toàn bộ các đỉnh, mỗi đỉnh chỉ qua 1 lần, và sau đó quay về đỉnh bắt đầu.

Sau đây là hình ảnh minh họa thuật toán vét cạn bằng đệ quy, được tình bày bằng cây đệ quy:

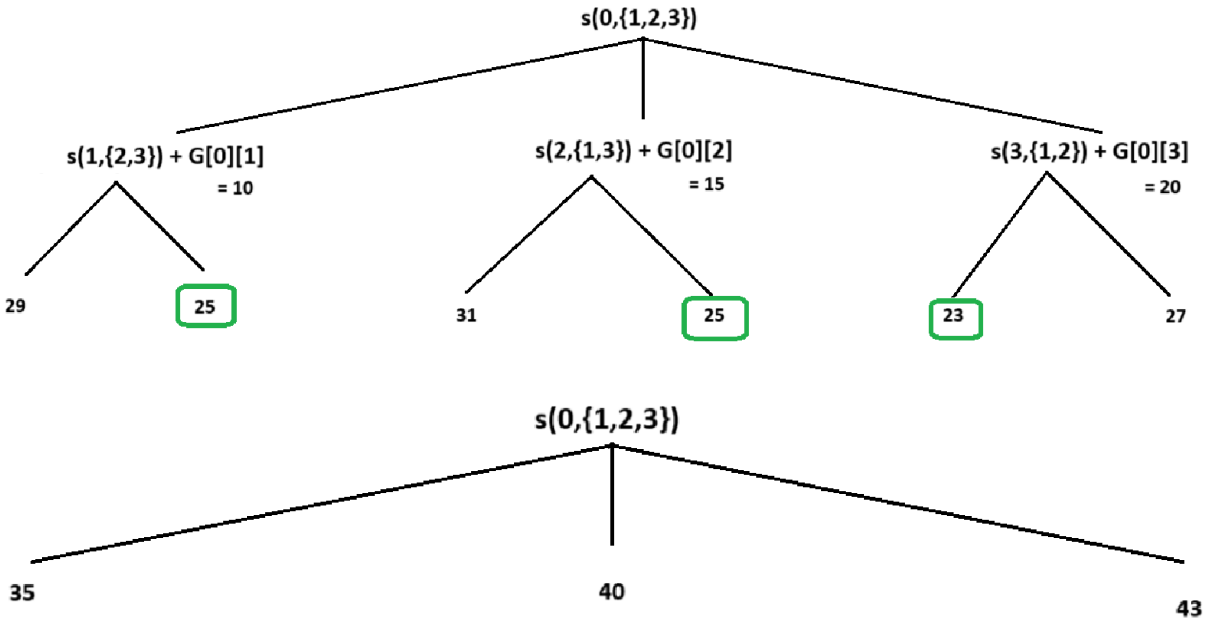
Bước 1: xuất phát từ điểm bắt đầu, liệt kê toàn bộ trạng thái có thể đi:



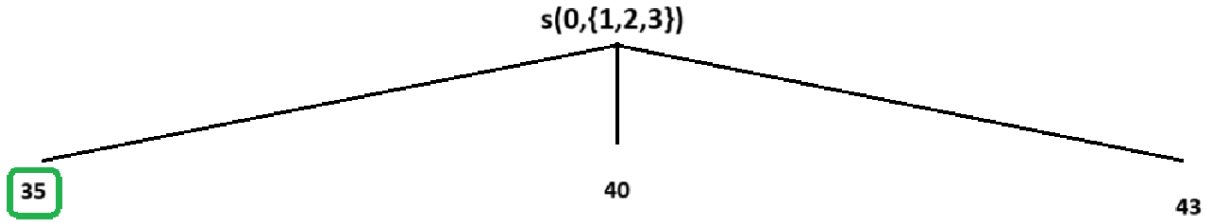
Bước 2: Tính toán chi phí từ lá lên gốc cây đệ quy



Bước 3: Chọn những cách đi có chi phí nhỏ nhất



Bước 4: Lặp lại quá trình cho tới khi đạt được kết quả cuối cùng



Vậy:

$$s(0,\{1,2,3\}) = 35$$

Sau khi tính toán, ta truy ngược lại quá trình di chuyển để tìm kiếm chu trình đã qua:

0 -> 1 -> 3 -> 2 -> 0

Tuy nhiên, đối với một đồ thị với n đỉnh, độ phức tạp của thuật toán vét cạn lên tới $O(n!)$, điều này gây tiêu tốn hiệu suất và tài nguyên lớn nếu n rất lớn.

Chính vì thế, một trong những cách nhằm cải thiện thuật toán đệ quy, giúp tiết kiệm tài nguyên và tăng hiệu suất hơn chính là thuật toán quy hoạch động (Dynamic Programming) với $O(n^2 2^n)$.

Đây cũng chính là phương pháp mà ba nhà khoa học máy tính Richard Ernest Bellman, Held và Richard Manning Karp đề xuất để giải quyết bài toán TSP.

2. Thuật toán Bellman - Held - Karp

2.1 Giới thiệu thuật toán quy hoạch động

Quy hoạch động (Dynamic Programming - DP) là một kỹ thuật thiết kế thuật toán rất mạnh mẽ, được sử dụng để giải quyết các bài toán tối ưu bằng cách chia nhỏ bài toán lớn thành các bài toán con có tính chất lặp lại, sau đó lưu trữ kết quả của các bài toán con để tránh tính toán lặp lại không cần thiết.

2.1.1 Ý tưởng chính

Thuật toán quy hoạch động hoạt động dựa trên hai nguyên lý quan trọng:

- **Tối ưu hóa con (Optimal Substructure):** Lời giải tối ưu của bài toán lớn có thể xây dựng từ lời giải tối ưu của các bài toán con.
- **Chồng lặp bài toán con (Overlapping Subproblems):** Bài toán con được lặp đi lặp lại nhiều lần trong quá trình tính toán, do đó việc lưu trữ và tái sử dụng kết quả là hiệu quả.

2.1.2 Các bước chính khi giải bài toán bằng quy hoạch động

- **Xác định trạng thái (state):** Biểu diễn một cách tổng quát bài toán con cần giải.
- **Xác định công thức truy hồi (recurrence):** Mối quan hệ giữa bài toán lớn và các bài toán con.
- **Xác định điều kiện cơ sở (base case):** Trạng thái ban đầu.
- **Tính toán theo thứ tự:** Thường theo hướng từ dưới lên (bottom-up) hoặc từ trên xuống (top-down với kỹ thuật ghi nhớ - memoization).
- **Truy vết (nếu cần):** Dùng để phục hồi lời giải (đường đi, lựa chọn, v.v.).

2.1.3 Ứng dụng

Quy hoạch động có mặt trong rất nhiều bài toán tin học kinh điển như:

- Bài toán balo (knapsack)
- Bài toán dãy con tăng dài nhất (LIS)

- Chuỗi con chung dài nhất (LCS)
- Bài toán người du lịch (TSP)
- Cắt dây, chia tiền, ghép đoạn...

2.1.4 Ưu điểm

- Giảm thời gian tính toán bằng cách tránh lặp lại.
- Dễ lập trình và dễ kiểm soát độ phức tạp.

2.1.5 Hạn chế

- Cần nhiều bộ nhớ để lưu trữ trạng thái.
- Đôi khi việc xác định trạng thái và công thức quy hoạch động không đơn giản.

2.2 Tiếp cận bài toán người giao hàng với quy hoạch động

Ở phương pháp vét cạn được nêu ở trên, ta sẽ tốn rất nhiều thời gian và tài nguyên mỗi khi gọi một nhánh đệ quy để tính toán từng trạng thái s(v,S), thay vào đó, với dynamic programming, ta sẽ không tính toán lại từng trạng thái nữa mà sẽ lưu thẳng các trạng thái đó vào một mảng giá trị 2D - dp[i][j] để phù hợp sử dụng trong tính toán.

Trong đó, ta sử dụng i để biểu thị trạng thái những đỉnh đã qua hay chưa qua, bằng cách chuyển i từ hệ thập phân sang hệ nhị phân với 1 là đỉnh đã qua và 0 là đỉnh chưa qua. Và j được sử dụng để biểu thị đỉnh hiện tại mà người giao hàng đang đứng.

Ví dụ: trong một đồ thị 4 đỉnh, dp[7][2] mang ý nghĩa:

- **7**: chuyển sang hệ nhị phân là 0111, mang ý nghĩa rằng người giao hàng đã đi qua đỉnh 0, đỉnh 1 và đỉnh 2. Tuy nhiên cần phải đi qua thêm đỉnh 3 và trở về lại đỉnh bắt đầu.
- **2**: hiện tại người giao hàng đang đứng ở đỉnh 2.

Bằng cách lưu lại các giá trị trạng thái, ta sẽ không cần phải sử dụng đệ quy để tính lại các trạng thái đã tính trước đó, giúp tiết kiệm thời gian, tài nguyên và tăng hiệu suất tính toán với công thức truy hồi:

$$dp[mask][i] = \min_{j \notin mask} (c[i][j] + dp[mask \cup \{j\}][j])$$

Cuối cùng, $min\_cost = dp[1 << start][start]$.

Để truy xuất lại chu trình có chi phí ít nhất, ta sẽ khai báo một mảng $path[2^{num\_vertices}][num\_vertices]$ nhằm lưu trữ đỉnh cần phải đi tiếp trong chu trình ngắn nhất sau trạng thái [mask][i] hiện tại. Ta sẽ thực hiện quá trình này song song với việc gán các giá trị vào mảng dp.

3. Giải thích thuật toán Bellman-Held-Karp chi tiết

*** Mục tiêu 1: Tìm min_cost**

Bước 1: Chuyển input edge list thành adjacency matrix để dễ dàng trong quá trình tính toán:

```
    int num_vertices = 0;

    int **edge1 = new int *[numberOfEdges];
    for (int i = 0; i < numberOfEdges; ++i)
    {
        edge1[i] = new int[3];
        edge1[i][0] = edge[i][0];
        edge1[i][1] = edge[i][1];
        edge1[i][2] = edge[i][2];
    }

    int countarr[256];
    for (int i = 0; i < 256; ++i)
    {
        countarr[i] = -1;
    }
    for (int i = 0; i < numberOfEdges; ++i)
    {
        ++countarr[edge1[i][0]];
        ++countarr[edge1[i][1]];
    }

    int mark = 0;

    for (int i = 0; i < 256; ++i)
    {
        if (countarr[i] != -1)
        {
            ++num_vertices;
            countarr[i] = mark++;
```

```
        }
    }
    for (int i = 0; i < numberOfEdges; ++i)
    {
        edge1[i][0] = countarr[edge1[i][0]];
        edge1[i][1] = countarr[edge1[i][1]];
    }
    int **G = new int *[num_vertices];
    for (int i = 0; i < num_vertices; ++i)
    {
        G[i] = new int[num_vertices]{};
    }
    for (int j = 0; j < numberOfEdges; ++j)
    {
        G[edge1[j][0]][edge1[j][1]] = edge1[j][2];
    }
```

Bước 2: Khởi tạo mảng dp với kích thước $2^{num\_vertices} \times num\_vertices$ và gán giá trị khởi đầu của mảng bằng -1:

```
    int masknum = pow(2, num_vertices);
        int **dp = new int *[masknum];
        for (int i = 0; i < masknum; ++i)
        {
            dp[i] = new int[num_vertices];
            path[i] = new int[num_vertices];
            for (int j = 0; j < num_vertices; ++j)
            {
                dp[i][j] = -1;
            }
        }
```

Bước 3: Với các dp[1 $\ll$ start][i], ta gán vào các giá trị G[i][start] để làm cơ sở (lưu ý: nếu G[i][start] == 0 hoặc i == start) thì sẽ không gán giá trị:

```
    for (int i = 0; i < num_vertices; ++i)
    {
        if (i == start || G[i][start] == 0)
            continue;
        dp[masknum - 1][i] = G[i][start];
    }
```

Bước 4: Chạy vòng lặp từ lá của cây đệ quy lên gốc để xác định giá trị từng trạng thái, lưu lại vào mảng dp bằng công thức truy hồi :

```
for (int mask = (1 << num_vertices) - 2; mask > 0; --mask){
    if (!(mask & (1 << start)))
        continue;
    for (int i = 0; i < num_vertices; ++i){
        if (!(mask & (1 << i)) || (i == start && mask != (1 <<
         ↪ start))) continue;
        for (int j = 0; j < num_vertices; ++j){
            if (mask & (1 << j) || G[i][j] == 0 || (dp[mask | (1 <<
             ↪ j)][j] == -1))
                continue;
            if (dp[mask][i] < dp[mask | (1 << j)][j] + G[i][j] &&
             ↪ dp[mask][i] != -1){
                continue;
            }
            else{
                dp[mask][i] = dp[mask | (1 << j)][j] + G[i][j];
            }
        }
        if (mask == (1 << start) && dp[1 << start][start] != -1){
            int maskcopy = mask;
        }
    }
}
```

***Lưu ý:** Khi chạy các vòng lặp, các trạng thái dp[mask][i] mà:

- Tại mask, đỉnh bắt đầu chưa được thăm
- Với mọi dp[mask][i], mà tại mask, đỉnh i chưa được thăm
- G[i][j] == 0 (không tồn tại đường đi từ đỉnh i đến đỉnh j)
- Mặc dù dp[mask][i] < dp[mask | (1 ≪ j)][j] + G[i][j] nhưng dp[mask][i] == -1

Thì ta skip và tới vòng lặp tiếp theo để tối ưu thời gian và tài nguyên.

Bước 5: Xác định kết quả:

```
min_cost = dp[1 << start][start];
```

*** Mục tiêu 2: Tìm shortest circuit**

Bước 1: Khởi tạo mảng path với kích thước $2^{num\_vertices} \times num\_vertices$ nhằm lưu trữ đỉnh cần phải đi tiếp trong chu trình ngắn nhất sau trạng thái [mask][i] hiện tại và gán giá trị khởi đầu của mảng bằng -1:

```
int **path = new int *[masknum];
for (int i = 0; i < masknum; ++i)
{
    path[i] = new int[num_vertices];
    for (int j = 0; j < num_vertices; ++j)
    {
        path[i][j] = -1;
    }
}
```

Bước 2: Khai báo một biến string route để lưu lại chu trình cuối cùng, sau đó gán đỉnh bắt đầu vào route:

```
string route;
for (int i = 0; i < 256; ++i)
{
    if (countarr[i] == start)
    {
        char temp = i;
        route.push_back(temp);
        route.push_back(' ');
        break;
    }
}
```

Bước 3: Với mỗi lần cập nhật giá trị của dp[mask][i], ta cũng sẽ cập nhật giá trị của path[mask][i]:

```
dp[mask][i] = dp[mask | (1 << j)][j] + G[i][j];
path[mask][i] = j;
```

Bước 4: Khi đã hoàn thành tính min_cost và min_cost khác -1, ta sẽ truy ngược lại path để lưu vào route:

```
    if (mask == (1 << start) && dp[1 << start][start] != -1)
    {
        int maskcopy = mask;
        int nextvertex = path[maskcopy][i];
        while (maskcopy != masknum - 1)
        {
            for (int k = 0; k < 256; ++k)
            {
                if (countarr[k] == nextvertex)
                {
                    char temp = k;
                    route.push_back(temp);
                    route.push_back(' ');
                    maskcopy = maskcopy | (1 << nextvertex);
                    nextvertex = path[maskcopy][nextvertex];
                    break;
                }
            }
        }
        for (int i = 0; i < 256; ++i)
        {
            if (countarr[i] == start)
            {
                char temp = i;
                route.push_back(temp);
                break;
            }
        }
```

Bước 5: Xác định kết quả:

```
     return route;
```

Sau cùng, delete các vùng nhớ động nằm trên heap để tránh việc rò rỉ bộ nhớ:

```
    for (int i = 0; i < numberOfEdges; ++i)
    {
        delete[] edge1[i];
    }
    delete[] edge1;
```

```
    for (int i = 0; i < num_vertices; ++i)
    {
        delete[] G[i];
    }
    delete [] G;

    for (int i = 0; i < masknum; ++i)
    {
        delete [] dp[i];
        delete [] path[i] ;
    }
    delete [] dp;
    delete [] path;
```

Có thể thấy rằng, đối với những đồ thị ≥ 25 đỉnh, việc khai báo các mảng dp và path với kích thước $2^{num\_vertices} \times num\_vertices$ chiếm vô cùng nhiều không gian và với độ phức tạp của thuật toán Bellman-Held-Karp là $O(n^2 2^n)$, việc xử lí những đồ thị lớn không hề khả thi. Chính vì thế, thay vì tìm một kết quả hoàn toàn tối ưu, ta sẽ hướng đến việc tìm những kết quả xấp xỉ tối ưu nhất có thể bằng những thuật toán heuristic hay meta-heuristic. Sau đây là phần trình bày của em về thuật toán Ant Colony Optimization.

Chương III

HƯỚNG TIẾP CẬN VÀ THUẬT TOÁN ANT COLONY OPTIMIZATION CHI TIẾT

1. Giới thiệu thuật toán Tối ưu hóa Đàn kiến (Ant Colony Optimization)

Thuật toán Tối ưu hóa Đàn kiến (ACO) là một thuật toán metaheuristic lấy cảm hứng từ hành vi của đàn kiến trong tự nhiên, đặc biệt là cách chúng tìm đường ngắn nhất từ tổ đến nguồn thức ăn bằng cách sử dụng pheromone (chất dẫn dụ).

1.1 Ý tưởng chính

1.1.1 Mô phỏng đàn kiến:

- Mỗi "con kiến" là một tác nhân (agent) đại diện cho một giải pháp khả thi trong không gian tìm kiếm.
- Kiến di chuyển trên một đồ thị, nơi các đỉnh đại diện cho các trạng thái (ví dụ: thành phố trong TSP) và các cạnh có trọng số (khoảng cách, chi phí).

1.1.2 Pheromone và xác suất di chuyển:

- Mỗi cạnh của đồ thị có một lượng pheromone, ban đầu được khởi tạo ngẫu nhiên hoặc đồng đều.
- Khi kiến di chuyển, chúng chọn cạnh tiếp theo dựa trên:
  - **Pheromone**: Cạnh có lượng pheromone cao sẽ hấp dẫn hơn.

- **Thông tin heuristic**: Thường dựa trên chi phí hoặc khoảng cách (ví dụ: trong TSP, ưu tiên cạnh ngắn hơn).
- Công thức xác suất chọn cạnh thường là:

$$P_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{k \in \text{allowed}} \tau_{ik}^{\alpha} \cdot \eta_{ik}^{\beta}}$$

Trong đó:

* P_{ij}: Xác suất chọn cạnh từ i đến j.
* τ_{ij}: Lượng pheromone trên cạnh (i, j).
* η_{ij}: Giá trị heuristic (thường là $1/d_{ij}$, với d_{ij} là khoảng cách).
* α, β: Tham số điều chỉnh mức độ ảnh hưởng của pheromone và heuristic.

1.1.3 Cập nhật pheromone:

- Sau mỗi vòng lặp (iteration), pheromone trên các cạnh được cập nhật:
  - **Bay hơi pheromone:** Giảm lượng pheromone trên tất cả các cạnh để tránh hội tụ quá sớm:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij}$$

Với ρ là tỷ lệ bay hơi (thường $0 < \rho < 1$).

  - **Cộng pheromone:** Các cạnh được kiến đi qua được tăng pheromone:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}$$

$\Delta\tau_{ij}$ thường tỷ lệ nghịch với chi phí của giải pháp (ví dụ: $1/L$, với L là độ dài đường đi).

1.1.4 Lặp lại:

- Quá trình được lặp lại với nhiều thế hệ kiến cho đến khi đạt tiêu chí dừng (số vòng lặp tối đa, thời gian chạy, hoặc giải pháp đủ tốt).
- Giải pháp tốt nhất được lưu lại và trả về.

1.2 Quy trình thuật toán

1.2.1 Khởi tạo:

- Đặt giá trị ban đầu cho pheromone trên các cạnh.
- Khởi tạo các tham số α, β, ρ, số lượng kiến, tiêu chí dừng.

1.2.2 Vòng lặp chính:

- Mỗi kiến xây dựng một giải pháp bằng cách di chuyển trên đồ thị, chọn cạnh theo xác suất.
- Tính chi phí của mỗi giải pháp.
- Cập nhật pheromone:
  - Giảm pheromone trên tất cả các cạnh (bay hơi).
  - Tăng pheromone trên các cạnh kiến đi qua.

1.2.3 Kết thúc:

Trả về giải pháp tốt nhất tìm được sau khi đạt tiêu chí dừng.

Ưu điểm

- Hiệu quả trong việc tìm lời giải gần tối ưu cho các bài toán NP-Hard.
- Linh hoạt, dễ áp dụng cho nhiều bài toán tổ hợp.
- Khả năng khám phá không gian tìm kiếm lớn nhờ tính ngẫu nhiên và pheromone.

Nhược điểm

- Dễ bị kẹt ở cực trị cục bộ nếu tham số không được điều chỉnh tốt.
- Tốn thời gian tính toán với bài toán quy mô lớn.
- Phụ thuộc vào việc chọn tham số (α, β, ρ)..

2. Áp dụng thuật toán Tối ưu hóa đàn kiến vào bài toán chi tiết

Bước 1: Tạo ra một class Ant, nhằm lưu lại trạng thái các thành phố đã ghé thăm, chưa ghé thăm; một vector chứa các đỉnh mà kiến đi qua trên đường đi; tổng cost trên chu trình mà kiến đang đi:

```
class Ant
{
public:
    int citiesStatus;
    vector<int> pos;
    float currentcost;
};
```

Bước 2: Khai báo các parameters, đồng thời, khởi tạo mảng chứa các giá trị thể hiện lượng pheromone trên từng cạnh, và mảng chứa giá trị nghịch đảo của cost trên từng cạnh:

```
int Q = 4;
float rho = 0.1;
int bestCost = -1;
vector<int> bestTour;
int currentCost = 0;
int alpha = 1;
int beta = 4;
Ant **ants = new Ant *[num_vertices];
float **pheromone = new float *[num_vertices];
for (int i = 0; i < num_vertices; ++i){
    ants[i] = new Ant;
    ants[i]->citiesStatus = 1 << i;
    ants[i]->pos.push_back(i);
    ants[i]->currentcost = 0;
    pheromone[i] = new float[num_vertices];
    for (int j = 0; j < num_vertices; ++j){
        if (G[i][j] != 0){
            pheromone[i][j] = 100;
            G[i][j] = 100.0 / G[i][j];
        }
    }
}
```

Ở đây, em lựa chọn số lượng kiến bằng với số đỉnh mà đồ thị có, đồng thời mỗi con kiến sẽ bắt đầu tại mỗi đỉnh trong đồ thị trùng với số thứ tự của nó.

Bước 3: Với mỗi con kiến, với mỗi vị trí hiện tại của kiến, dựa vào đồ thị pheromone và đồ thị G, chúng ta sẽ tính xác suất của những đỉnh mà con kiến chưa ghé thăm:

```
for (int antNum = 0; antNum < num_vertices; ++antNum)
{
    bool roadExist = false;
    for (int steps = 0; steps < num_vertices - 1; ++steps)
    {
        roadExist = false;
        vector<float> probability(num_vertices, 0);
        float sumprob = 0;
        for (int j = 0; j < num_vertices; ++j)
        {
            if ((ants[antNum]->citiesStatus & 1 << j) == 0)
            {
                sumprob += (pow(pheromone[ants[antNum]->pos.back()][j],
                 ↪  alpha) * pow(G[ants[antNum]->pos.back()][j], beta));
            }
        }
        if (sumprob != 0)
        {
            roadExist = true;
            for (int j = 0; j < num_vertices; ++j)
            {
                if ((ants[antNum]->citiesStatus & 1 << j) == 0)
                {
                    probability[j] =
                     ↪  (pow(pheromone[ants[antNum]->pos.back()][j],
                     ↪  alpha) * pow(G[ants[antNum]->pos.back()][j],
                     ↪  beta)) / sumprob;
                }
            }
        }
        else
        {
            break;
        }
```

Bước 4: Từ mảng probability, ta tạo ra mảng culmulative, và tạo một số thực ngẫu nhiên trong khoảng từ 0 tới 1 để lựa chọn thành phố con kiến sẽ ghé thăm tiếp theo:

```
vector<float> culmulative(num_vertices, 0);
for (int j = 0; j < num_vertices; ++j){
    if (probability[j] == 0){
        continue;
        }
    for (int k = j; k < num_vertices; ++k){
    culmulative[j] += probability[k];
    }
}

random_device rd;
mt19937 gen(rd());
uniform_real_distribution<float> dist(0.0, 1.0);
float randNum = dist(gen);
float firsthead = 0;
int firstindex;
for (int search = num_vertices - 1; search >= 0; --search){
    if (culmulative[search] != 0){
        firsthead = culmulative[search];
        firstindex = search;
        break;
    }
}
if (0 <= randNum && randNum <= firsthead){
    ants[antNum]->citiesStatus = ants[antNum]->citiesStatus | 1 <<
     ↪  (firstindex);
    ants[antNum]->currentcost += 100.0 /
     ↪  G[ants[antNum]->pos.back()][firstindex];
    ants[antNum]->pos.push_back(firstindex);
    continue;
}
float back = 0;
for (int i = 0; i < num_vertices - 1; ++i){
    if (culmulative[num_vertices - 1 - i] == 0)
    {
        continue;
    }
    back = culmulative[num_vertices - 1 - i];
    if (back == 1)
    {
        ants[antNum]->citiesStatus = ants[antNum]->citiesStatus | 1 <<
         ↪  (num_vertices - 1 - i);
        ants[antNum]->currentcost += 100.0 /
         ↪  G[ants[antNum]->pos.back()][num_vertices - 1 - i];
        ants[antNum]->pos.push_back(num_vertices - 1 - i);
        break;
```

```
    }
    float head = 0;
    int index;
    for (int l = num_vertices - 2 - i; l >= 0; --l)
    {
        if (culmulative[l] == 0)
        {
            continue;
        }
        head = culmulative[l];
        index = l;
        break;
    }
    if (back != 0 && head != 0 && back <= randNum && randNum <= head)
    {
        ants[antNum]->citiesStatus = ants[antNum]->citiesStatus | 1 <<
         ↪ index;
        ants[antNum]->currentcost += 100.0 /
         ↪ G[ants[antNum]->pos.back()][index];
        ants[antNum]->pos.push_back(index);
        break;
    }
}
```

Chú ý: nếu tìm được thành phố tiếp theo cần đến, ta sẽ cập nhật các thuộc tính của kiến để tiếp tục step đi tiếp theo.

Bước 5: Nếu con kiến đó tìm được chu trình về lại đỉnh bắt đầu, ta sẽ so sánh chi phí chu trình kiến đã đi với bestCost và lưu đường đi vào bestTour:

```
if (roadExist == true &&
 ↪ G[ants[antNum]->pos.back()][ants[antNum]->pos.front()] != 0){
    ants[antNum]->currentcost += 100.0 /
     ↪ G[ants[antNum]->pos.back()][ants[antNum]->pos.front()];
    ants[antNum]->pos.push_back(ants[antNum]->pos.front());
    if (ants[antNum]->currentcost < bestCost || bestCost == -1){
        bestCost = ants[antNum]->currentcost;
        bestTour = ants[antNum]->pos;
    }
}
else{
    ants[antNum]->currentcost = 0;
}
```

Bước 6: Sau khi toàn bộ tất cả kiến hoàn thành chu trình ta sẽ áp dụng công thức tính lượng pheromone trên từng cạnh mà kiến đã đi qua dựa trên sự bay hơi và tích tụ:

```
int **deltaPheromone = new int *[num_vertices];
for (int i = 0; i < num_vertices; ++i)
{
    deltaPheromone[i] = new int[num_vertices]();
}
for (int antNum = 0; antNum < num_vertices; ++antNum){
    for (int i = 0; i < ants[antNum]->pos.size() - 1; ++i)
    {
        if (ants[antNum]->currentcost != 0)
        {
            deltaPheromone[ants[antNum]->pos[i]][ants[antNum]->pos[i +
             ↪  1]] += Q / (float)ants[antNum]->currentcost;
        }
    }
}
for (int i = 0; i < num_vertices; ++i)
{
    for (int j = 0; j < num_vertices; ++j)
    {
        pheromone[i][j] = (1 - rho) * pheromone[i][j] +
         ↪  deltaPheromone[i][j];
    }
}
for (int i = 0; i < num_vertices; ++i){
    delete[] deltaPheromone[i];
}
delete[] deltaPheromone;
for (int i = 0; i < num_vertices; ++i){
    ants[i]->citiesStatus = 1 << i;
    ants[i]->pos.clear();
    ants[i]->pos.push_back(i);
    ants[i]->currentcost = 0;
}
```

Bước 7: Lặp lại các bước đã tính toán sau số lần tự chọn để tìm kiếm chu trình xấp xỉ tốt nhất.

Bước 8: Cuối cùng, khôi phục lại chu trình có cost xấp xỉ nhỏ nhất dựa trên start vertex và delete các mảng động đã được khai báo trước đó để tránh rò rỉ bộ nhớ:

```
string route;
int startindex;
for (int i = 0; i < bestTour.size(); ++i)
{
    if (bestTour[i] == countarr[start])
    {
        startindex = i;
        break;
    }
}
for (int i = startindex; i < bestTour.size(); ++i)
{
    for (int j = 0; j < 256; ++j)
    {
        if (countarr[j] == bestTour[i])
        {
            char temp = j;
            route.push_back(temp);
            break;
        }
    }
    route.push_back(' ');
}
for (int i = 1; i < startindex + 1; ++i)
{
    for (int j = 0; j < 256; ++j)
    {
        if (countarr[j] == bestTour[i])
        {
            char temp = j;
            route.push_back(temp);
            break;
        }
    }
    if (i != startindex)
    {
        route.push_back(' ');
    }
}

for (int i = 0; i < numEdge; ++i)
{
    delete[] edge1[i];
}
delete[] edge1;

```

```
for (int i = 0; i < num_vertices; ++i)
{
    delete[] G[i];
    delete ants[i];
    delete[] pheromone[i];
}
delete[] ants;
delete[] G;
delete[] pheromone;

return route;
```

Chương IV

TỔNG KẾT

1. Kiến thức

- Hiểu rõ tính chất NP-Hard của bài toán Người du lịch (TSP), nhận biết đây là bài toán tối ưu hóa đòi hỏi nỗ lực lớn cả về lý thuyết lẫn thực nghiệm.
- Nắm vững cách xây dựng đồ thị đầy đủ với ma trận trọng số và nguyên tắc tìm chu trình quay về đỉnh xuất phát sao cho tổng chi phí nhỏ nhất.
- Thành thạo triển khai và tối ưu thuật toán quy hoạch động Bellman–Held–Karp cho TSP với đồ thị ≤ 20 đỉnh, bao gồm cách biểu diễn trạng thái (mask, đỉnh cuối) và công thức truy hồi.
- Hiểu và áp dụng thuật toán Ant Colony Optimization (ACO) cho các đồ thị lớn hơn (> 20 đỉnh), tận dụng tính chất heuristic để tìm lời giải gần tối ưu trong thời gian hợp lý.

2. Kết luận

Ưu điểm của thuật toán Bellman–Held–Karp

- **Chính xác tuyệt đối:** Thuật toán Bellman–Held–Karp đảm bảo trả về chi phí tối ưu cho TSP với đồ thị ≤ 20 đỉnh.
- **Cấu trúc quy hoạch động rõ ràng:** Mỗi trạng thái chỉ phụ thuộc vào tập con các đỉnh đã thăm và đỉnh hiện tại, dễ dàng thiết lập công thức truy hồi.
- **Tái sử dụng kết quả:** Nhờ memoization, thuật toán tránh tính toán lặp lại, tiết kiệm thời gian so với liệt kê hoán vị toàn phần.

- **Phù hợp cho đồ thị nhỏ:** Với đồ thị ≤ 20 đỉnh, Bellman–Held–Karp thực thi trong thời gian chấp nhận được, phù hợp cho các bài toán quy mô vừa.

Ưu điểm của thuật toán Ant Colony Optimization (ACO)

- **Hiệu quả cho đồ thị lớn:** Với đồ thị > 20 đỉnh, ACO cung cấp lời giải gần tối ưu trong thời gian hợp lý, phù hợp cho các bài toán thực tế với hàng trăm hoặc hàng nghìn đỉnh.
- **Tính linh hoạt:** ACO dễ dàng thích nghi với các biến thể của TSP, tận dụng cơ chế học tập dựa trên pheromone để cải thiện chất lượng lời giải.

Hạn chế

- **Hạn chế của Bellman–Held–Karp:** Không phù hợp cho đồ thị lớn (> 20 đỉnh) do độ phức tạp tính toán cao, đòi hỏi sử dụng các phương pháp heuristic như ACO.
- **Hạn chế của ACO:** Lời giải không đảm bảo tối ưu tuyệt đối, phụ thuộc vào tham số và cấu hình của thuật toán.
- **Không tận dụng cấu trúc đặc biệt:** Cả hai thuật toán không tận dụng triệt để các tính chất đặc biệt của đồ thị (như tính chất metric hay ma trận tam giác), đòi hỏi kết hợp các kỹ thuật cắt nhánh hoặc xấp xỉ chuyên biệt.

Dựa trên các kết quả đạt được, em đã hoàn thành bài tập lớn bằng cách triển khai thuật toán Bellman–Held–Karp cho đồ thị ≤ 20 đỉnh và thuật toán ACO cho đồ thị > 20 đỉnh trên môi trường lập trình C++. Bài tập giúp em hiểu sâu hơn về việc áp dụng quy hoạch động và các phương pháp heuristic vào thực tiễn, đồng thời nhận thấy tiềm năng của các thuật toán này trong việc giải quyết TSP ở các quy mô khác nhau. Ngoài kiến thức về kỹ thuật thuật toán và phân tích độ phức tạp, em còn rèn luyện kỹ năng chia nhỏ công việc, kiểm thử và tối ưu code. Qua quá trình này, em nâng cao đam mê với các bài toán tối ưu hóa đồ thị và có thêm động lực tìm hiểu sâu hơn về các phương pháp xấp xỉ cũng như ứng dụng thực tế của TSP trong logistics, lập lịch và quản lý mạng lưới.

Lời cảm ơn

Để hoàn thành báo cáo này, em xin gửi lời cảm ơn chân thành đến thầy Lê Hồng Trang, người đã tận tình giảng dạy và truyền đạt những kiến thức quý báu về bài toán Người giao hàng (TSP). Sự nhiệt huyết và tận tâm của thầy là nguồn động lực lớn giúp nhóm em có đủ hành trang để hoàn thành đề tài, đặc biệt trong việc áp dụng thuật toán Bellman–Held–Karp cho đồ thị ≤ 20 đỉnh và thuật toán Ant Colony Optimization (ACO) cho đồ thị > 20 đỉnh.

Em cũng xin bày tỏ lòng biết ơn sâu sắc đến thầy Trần Hồng Tài, người đã đưa chủ đề TSP vào chương trình giảng dạy, trang bị cho em nền tảng vững chắc về quy hoạch động, lý thuyết đồ thị và các phương pháp heuristic. Nhờ vậy, em có thể triển khai và tối ưu các thuật toán giải TSP một cách hiệu quả.

Trong quá trình thực hiện, em đã tích lũy nhiều kiến thức mới, từ cách xây dựng trạng thái, thiết lập hàm truy hồi, đến việc tối ưu hóa thời gian và bộ nhớ khi sử dụng Bellman–Held–Karp, cũng như điều chỉnh các tham số của ACO để đạt hiệu suất tốt hơn. Tuy nhiên, vẫn còn những sai sót và hạn chế do thiếu kinh nghiệm. Em rất mong nhận được những nhận xét và góp ý quý giá từ các thầy để hoàn thiện hơn về nội dung và phương pháp thực hiện.

Một lần nữa, em xin gửi lời cảm ơn sâu sắc đến thầy Lê Hồng Trang và thầy Trần Hồng Tài vì sự tận tâm và những kiến thức quý báu đã truyền đạt. Sự hướng dẫn của hai thầy như ngọn hải đăng, tiếp thêm động lực để em không ngừng nỗ lực trong môn học này và các nghiên cứu sau này.

Cuối cùng, em xin kính chúc hai thầy luôn dồi dào sức khỏe, hạnh phúc và thành công. Em hy vọng sẽ tiếp tục nhận được sự hỗ trợ và hướng dẫn từ các thầy trong những hành trình học tập và nghiên cứu tương lai.

Em xin chân thành cảm ơn!

Tài liệu tham khảo

[1] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," J. ACM, vol. 9, no. 1, pp. 61–63, Jan. 1962. Available: https://dl.acm.org/doi/pdf/10.1145/321105.321111. Accessed: June 22, 2025.

[2] Q. N. Nguyen, "Travelling salesman problem and Bellman-Held-Karp algorithm," YouTube, May 10, 2020. Available: https://www.math.nagoya-u.ac.jp/ richard/teaching/s2020/Quang1.pdf. Accessed: June 22, 2025.

[3] C. Blum, "Ant colony optimization: Introduction and recent trends," Phys. Life Rev., vol. 2, no. 4, pp. 353–373, Dec. 2005. Available: PDF_Link. Accessed: June 22, 2025.

[4] Reducible, “The Traveling Salesman Problem: When Good Enough Beats Perfect”, YouTube, [Online]. Available: https://youtu.be/GiDsjIBOVoA. Accessed: June 22, 2025.