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

Cấu trúc Rời rạc cho Khoa học Máy tính (CO1007)

Bài tập lớn

The Travelling salesman problem

Giáo viên hướng dẫn: Mai Xuân Toàn
Sinh viên thực hiện: Đoàn Thị Hương Lan - 2311806.

Thành phố Hồ Chí Minh, Tháng 6 2025

Mục lục

1 Vấn đề đặt ra:

"Bài toán nhân viên bán hàng du lịch, còn được gọi là vấn đề nhân viên bán hàng du lịch (TSP), đặt câu hỏi sau: "Đưa ra danh sách các thành phố và khoảng cách giữa mỗi cặp thành phố, tuyến đường ngắn nhất có thể đến thăm mỗi thành phố chính xác một lần và trở về thành phố xuất phát là gì?" Đây là một bài toán khó NP trong tối ưu hóa tổ hợp, quan trọng trong khoa học máy tính lý thuyết và nghiên cứu hoạt động"

Trong lĩnh vực tối ưu hóa tổ hợp, vấn đề của người du lịch (TSP) là một trong những bài toán được quan tâm nhiều nhất. TSP đặt ra câu hỏi quan trọng: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" Bài toán này không chỉ có ứng dụng trong thực tế mà còn là một bài toán NP-khó, thu hút sự quan tâm của cả lý thuyết và thực tiễn.

2 Giải quyết vấn đề

2.1 Phân tích bài toán

2.1.1 Dữ liệu

- Danh sách các thành phố (gồm số lượng và có thể đi lại giữa các thành phố hay không?)
- Khoảng cách giữa mỗi cặp thành phố
- Thành phố xuất phát

2.1.2 Mục tiêu

Tìm tuyến đường cao cho tổng khoảng cách di qua tất cả các thành phố là nhỏ nhất, mỗi thành phố chỉ đi qua một lần, và trờ về thành phố xuất phát.

2.2 Hiện thực bằng thuật toán

2.2.1 Yêu cầu

Viết hàm Traveling() để tính toán quãng đường ngắn nhất để đi qua tất cả các đỉnh trong đồ thị và quay trở lại đỉnh bắt đầu, với hình dung:

- Đồ thị là bản đồ các thành phố, có thể xét sự kết nối giữa các thành phố, được xem như đỉnh.
- Trọng số các cạnh trong đồ thị là quãng đường di chuyển giữa các cặp thành phố; như vậy, trọng số sẽ luôn dương.

2.2.2 Đầu ra của hàm:

Hàm sẽ in ra một chuỗi là đường dẫn ngắn nhất đi qua tất cả các đỉnh, không lặp lại, và quay trở lại đỉnh bắt đầu.

2.3 Ý tưởng

Có nhiều ý tưởng được đưa ra để giải quyết vấn đề nhân viên bán hàng du lịch, có thể sơ lược như:

- Thuật toán Greedy: Ý tưởng cơ bản của thuật toán Greedy là chọn lựa lựa chọn tốt nhất tại mỗi bước của quá trình giải quyết vấn đề mà không quan tâm đến các lựa chọn tương lai, hy vọng rằng loạt các lựa chọn tốt nhất địa phương này sẽ dẫn đến lời giải tốt nhất cho vấn đề.

  Nhược điểm là không đảm bảo tìm ra lời giải tối ưu toàn cục và có thể rơi vào các bế tắc cục bộ. Ngoài ra, kết quả của thuật toán có thể phụ thuộc mạnh vào điểm xuất phát ban đầu. Nếu điểm xuất phát được chọn không tốt, có thể dẫn đến một đường đi không tối ưu hoặc không tạo ra một lời giải tối ưu.

- Thuật toán Dynamic Programming (Lập trình động): Dùng trong trường hợp bài toán có cấu trúc con tối ưu. Ý tưởng chính là lưu trữ các kết quả đã tính toán trước đó để tránh tính toán lại.

  Trong bài toán TSP, thuật toán Dynamic Programming có thể được sử dụng để giảm bớt số lần tính toán lại các đường đi, nhưng nó yêu cầu một lượng lớn bộ nhớ và có thể không hiệu quả cho các bài toán có số lượng thành phố lớn.

- Thuật toán Backtracking là một phương pháp tìm kiếm hệ thống trong đó chúng ta thử từng lựa chọn một và "quay lui" nếu không tìm được giải pháp. Ý tưởng chính là sử dụng đệ quy để thử tất cả các khả năng có thể của vấn đề.

  Mặc dù hiệu quả trong một số trường hợp, nhưng nhược điểm của backtracking là nó có thể dễ dàng rơi vào tối ưu cục bộ và đòi hỏi một lượng lớn bộ nhớ đệm.

2.4 Khởi tạo đồ thị

Mục đích

- Danh sách đỉnh `vertexList`: chứa các đỉnh khác nhau có trong danh sách cạnh edgeList.
- Ma trận trọng số `weight[u][v]`: thể hiện chi phí đi từ đỉnh `u` đến `v`. Ban đầu khởi tạo là ∞ (vô cực), sau đó cập nhật theo dữ liệu đầu vào.
- Danh sách cạnh edges: mỗi cạnh là một `struct Edge` gồm `u, v, w` tương ứng chỉ số đỉnh `u, v` và trọng số `w`.
- Chuẩn bị dữ liệu đầu vào cho 3 thuật toán TSP: Backtracking, Dynamic Programming, và Heuristic (Greedy + 2-opt).

- Đảm bảo biểu diễn đồ thị đúng cấu trúc với trọng số chính xác. Giữ sự nhất quán giữa chỉ số đỉnh và ký tự đại diện bằng vertexList.

2.4.1 Khởi tạo `weight[u][v]`

```
// Initial matrix weight
for (int i = 0; i < 25; ++i)
    for (int j = 0; j < 25; ++j)
        weight[i][j] = INF;
```

- Khởi tạo tất cả giá trị trong ma trận trọng số là $INF = 10^9$, tượng trưng cho "không có đường đi".
- Kích thước 25×25 là đủ vì số lượng đỉnh tối đa là 25 ký tự khác nhau.

2.4.2 Tạo danh sách đỉnh `vertexList`

```
// Traverse each edge in edgeList
for (int i = 0; i < numEdges; ++i) {
    char from = (char)edgeList[i][0];
    char to = (char)edgeList[i][1];

    // If 'from' vertex is not in vertexList, add it
    if (std::find(vertexList.begin(), vertexList.end(), from) == vertexList.end())
        vertexList.push_back(from);

    // If 'to' vertex is not in vertexList, add it
    if (std::find(vertexList.begin(), vertexList.end(), to) == vertexList.end())
        vertexList.push_back(to);
}
```

- Đọc từng cạnh trong `edgeList`, trích xuất ký tự `from` và `to`.
- Nếu ký tự chưa có trong `vertexList`, thì thêm vào.
- Mục tiêu: lập danh sách đầy đủ các đỉnh xuất hiện trong input.

2.4.3 Sắp xếp `vertexList`

```
// Sort vertexList alphabetically
std::sort(vertexList.begin(), vertexList.end());
```

- Giúp đồng bộ thứ tự xử lý đỉnh, dễ debug và nhất quán khi in kết quả.

2.4.4 Gán trọng số các cạnh

```
// Assign weights and store edges
for (int i = 0; i < numEdges; ++i) {
    int u = getIndex((char)edgeList[i][0], vertexList); // Get index of 'from' vertex
    int v = getIndex((char)edgeList[i][1], vertexList); // Get index of 'to' vertex
    int w = edgeList[i][2];                             // Get the edge weight

    weight[u][v] = w;                                   // Set weight from u to v
    edges.push_back({u, v, w});                         // Store edge for later use
}
```

- `getIndex` chuyển từ ký tự đỉnh sang chỉ số trong `vertexList`.
- Gán trọng số từ đỉnh u đến v là w.
- Đồng thời lưu vào danh sách `edges` để sử dụng cho các thuật toán khác.

2.5 Thuật toán Brute-force TSP

2.5.1 Mô tả

- Tên: Brute-force TSP using Backtracking
- Mô tả: Duyệt qua mọi hoán vị của các đỉnh để tìm chu trnh có tổng trọng số nhỏ nhất.
- Độ phức tạp: O(n!) nên phù hợp với n nhỏ ($n \leq 15$)
- **Ý tưởng:**
  1. Từng bước, thử đi tới các đỉnh chưa thăm.
  2. Dừng sớm (cắt tỉa) nếu chi phí hiện tại đã vượt quá chi phí tốt nhất hiện tại (minCost).
  3. Nếu đã đi qua đủ n đỉnh, cộng thêm chi phí trở về đỉnh ban đầu nếu có đường đi.

2.5.2 Cấu trúc dữ liệu

- `weight[i][j]`: Ma trận trọng số giữa các đỉnh `i` và `j`.
- `visited[i]`:Mảng đánh dấu các đỉnh đã đi qua.
- `curPath`: Đường đi hiện tại.
- `bestPath`: Đường đi tối ưu tìm được.
- `minCost`: Chi phí nhỏ nhất hiện tại.

2.5.3 Thực hiện hàm `Traveling_Backtrack`

```
void TSP_Backtrack(int current, int start, int n, int depth, int curCost){...}
```

Tham số:

- `current`: Đỉnh hiện tại đang được xét hoặc đang đứng.
- `start`: Đỉnh xuất phát ban đầu trong quá trình tìm kiếm hoặc duyệt.
- `n`: Tổng số đỉnh trong đồ thị.
- `depth`: Số lượng đỉnh đã đi qua hoặc chiều sâu hiện tại của hành trình.
- `curCost`: Tổng chi phí tích lũy đến thời điểm hiện tại.

Logic hoạt động:

- Nếu `curCost` $\geq$ `minCost`: dừng nhánh hiện tại (cắt tỉa).
- Nếu đã đi qua đủ n đỉnh:
  - Kiểm tra xem có đường quay về `start` không.
  - Nếu có, cập nhật `minCost` và `bestPath` nếu tổng chi phí tốt hơn.
- Duyệt tất cả các đỉnh `nxt` sao cho:
  - `visited[nxt] == false` và
  - `weight[current][nxt] < INF`
- Với mỗi `nxt` thỏa điều kiện:
  - Đánh dấu `visited[nxt]` = `true`
  - Thêm `nxt` vào `curPath`
  - Gọi đệ quy
  - Quay lui:
    - Xóa `nxt` khỏi `curPath`
    - Bỏ đánh dấu `visited[nxt]`

Các bước thực hiện

1. Xác định số lượng đỉnh và đỉnh xuất phát

   ```
   int n = vertexList.size();
   int start = getIndex(start_char, vertexList);
   ```

   Lấy số lượng đỉnh từ danh sách `vertexList` và xác định chỉ số tương ứng với ký tự đỉnh xuất phát ban đầu. Khởi tạo giá trị ban đầu

   ```
   minCost = INF;
   bestPath.clear();
   curPath.clear();
   ```

   Gán chi phí ban đầu là `INF`, xóa đường đi tốt nhất và đường đi hiện tại trước khi bắt đầu quay lui.

2. Thiết lập trạng thái visited

```
for (int i = 0; i < n; ++i)
    visited[i] = false;

visited[start] = true;
curPath.push_back(start);
```

Đặt tất cả các đỉnh là chưa thăm, đánh dấu đỉnh bắt đầu và thêm vào hành trình ban đầu.

3. Gọi hàm đệ quuy **Hàm** `TSP_Backtrack`

```
void TSP_Backtrack(int current, int start, int n, int depth, int curCost) {
    if (curCost >= minCost) return;

    if (depth == n) {
        if (weight[current][start] < INF) {
            int totalCost = curCost + weight[current][start];
            if (totalCost < minCost) {
                minCost = totalCost;
                bestPath = curPath;
                bestPath.push_back(start);
            }
        }
        return;
    }

    for (int nxt = 0; nxt < n; ++nxt) {
        if (!visited[nxt] && weight[current][nxt] < INF) {
            visited[nxt] = true;
            curPath.push_back(nxt);
            TSP_Backtrack(nxt, start, n, depth + 1, curCost + weight[current][nxt
                ]);
            curPath.pop_back();       // Quay lui
            visited[nxt] = false;
        }
    }
}
```

- Nếu `curCost` đã vượt quá `minCost` hiện tại, dừng nhánh này để tiết kiệm thời gian (cắt tỉa).
- Nếu đã đi đủ n đỉnh, kiểm tra có thể quay về đỉnh xuất phát hay không:

- Nếu có, cộng thêm chi phí quay về và cập nhật `minCost`, đồng thời lưu `bestPath` nếu tốt hơn.

- Nếu chưa đi hết:
  - Duyệt tất cả các đỉnh `nxt` chưa thăm, có đường đi từ `current`.
  - Đánh dấu đã thăm, thêm vào hành trình, gọi đệ quy.
  - Sau khi quay lui, xóa đỉnh khỏi hành trình và bỏ đánh dấu.

```
TSP_Backtrack(start, start, n, 1, 0);
```

Đặt tất cả các đỉnh là chưa thăm, đánh dấu đỉnh bắt đầu và thêm vào hành trình ban đầu.

4. Xử lý kết quả sau khi đệ quy

```
std::string result;
if (bestPath.empty())
    result = "Error: No valid cycle found";
else {
    for (int i = 0; i < bestPath.size(); ++i) {
        if (i) result += " ";
        result += vertexList[bestPath[i]];
    }
}
```

Nếu không tìm thấy chu trình hợp lệ, trả về thông báo lỗi. Ngược lại, dựng chuỗi kết quả từ danh sách chỉ số trong `bestPath`.

5. Trả về kết quả

```
return result;
```

Hoàn tất hàm bằng cách trả lại kết quả đường đi tốt nhất dưới dạng chuỗi.

2.6 Thuật toán Held-Karp (Dynamic Programming) (dùng quy hoạch động)

2.6.1 Mô tả

- Tên: Held-Karp Algorithm
- Mô tả: Quy hoạch động với tập con (subset DP)

  Trạng thái: dp[mask][u] là chi phí nhỏ nhất để đi qua tập mask và kết thúc tại đỉnh u
- Độ phức tạp: $O(n^2 \cdot 2^n)$ nên phù hợp cho n $\leq$ 20.
- **Ý tưởng:**

  1. **Biểu diễn tập các đỉnh đã đi qua bằng Bitmask:**
     Sử dụng một số nguyên để biểu diễn tập các đỉnh đã thăm.
     Ví dụ: nếu có 4 đỉnh và đã thăm đỉnh 0 và 2, thì bitmask tương ứng là 0101 (giá trị thập phân là 5).
  2. **Khởi tạo bảng quy hoạch động** `dp[mask][u]`:
     `dp[mask][u]` lưu chi phí nhỏ nhất để đi từ đỉnh xuất phát, qua tất cả các đỉnh trong `mask`, và kết thúc tại đỉnh u.
     Gán giá trị khởi tạo:

     $$dp[1 \ll \text{start}][\text{start}] = 0$$

  3. **Duyệt qua tất cả các trạng thái** `mask` **theo số lượng bit bật:**
     - Với mỗi `mask`, duyệt qua tất cả đỉnh u có trong `mask`, xem như đỉnh cuối cùng đã đi qua.
     - Với mỗi đỉnh $v \neq u$ cũng có trong `mask`, cập nhật:

     $$dp[\text{mask}][u] = \min\left(dp[\text{mask}][u],\ dp[\text{mask} \setminus \{u\}][v] + cost[v][u]\right)$$

  4. **Tính kết quả cuối cùng:**
     Sau khi đã đi qua tất cả các đỉnh (tức là `mask = 2`$^{\blacksquare}$`n - 1`), thử quay về đỉnh xuất phát:

     $$\text{answer} = \min_{u \neq \text{start}} \left(dp[2^n - 1][u] + cost[u][\text{start}]\right)$$

5. **Truy vết đường đi tối ưu (nếu cần):**
   Khi cập nhật `dp[mask][u]`, lưu lại đỉnh trước đó (hoặc đỉnh kế tiếp) để có thể truy ngược hành trình tối ưu sau khi tính toán xong.

2.6.2 Cấu trúc dữ liệu

- `std::vector<Edge> edges`: Lưu danh sách các cạnh trong đồ thị. Mỗi phần tử `Edge` gồm 3 trường: đỉnh đầu, đỉnh cuối, và trọng số.
- `std::vector<char> vertexList`: Danh sách các đỉnh của đồ thị, được biểu diễn dưới dạng ký tự `char`.
- `int weight[20][20]`: Ma trận trọng số, trong đó `weight[i][j]` là trọng số cạnh từ đỉnh i đến j. Nếu không có cạnh nối giữa hai đỉnh, ta gán `weight[i][j] = INF`.
- `int dp[1 << MAXN][MAXN]`: Bảng quy hoạch động lưu chi phí nhỏ nhất để đi qua một tập hợp các đỉnh (biểu diễn bởi `mask`) và kết thúc tại đỉnh `current`. `dp[mask][current]` chính là chi phí nhỏ nhất để đến `current` khi đã đi qua toàn bộ các đỉnh có bit bật trong `mask`.
- `int next_city[1 << MAXN][MAXN]`: Mảng hỗ trợ truy vết đường đi tối ưu. Với mỗi trạng thái `mask` và vị trí hiện tại `current`, giá trị lưu tại đây là đỉnh kế tiếp cần đi để đạt được chi phí tối ưu.

2.6.3 Thực hiện hàm `TravelingDP`

```
int TSP(const std::vector<Edge>& edges, const std::vector<char>& vertexList, int mask,
    int current, int start){...}
```

Tham số:

- `mask`: bitmask thể hiện tập các đỉnh đã đi qua.
- `current`: vị trí đỉnh hiện tại.
- `start`: đỉnh xuất phát ban đầu.

Ý tưởng thuật toán:

1. **Điều kiện dừng:** Nếu đã đi qua tất cả các đỉnh (`mask` $+ 1 = 2^n$), kiểm tra xem có cạnh từ `current` quay về `start` hay không:

- Nếu có: trả về tổng chi phí.
- Nếu không có đường về: trả về `INF`.

2. **Hồi quy có nhớ:** Nếu trạng thái `dp[mask][current]` đã được tính trước đó, trả về ngay để tránh tính lại.

3. **Duyệt các khả năng mở rộng:**
   - Với mỗi đỉnh `nxt` chưa được thăm (bit chưa bật trong `mask`).
   - Nếu có cạnh nối từ `current` đến `nxt`:
     - Tính chi phí đi từ `current` đến `nxt`.
     - Gọi đệ quy cho trạng thái mới: thêm `nxt` vào `mask`, cập nhật vị trí hiện tại là `nxt`.
     - Lưu lại kết quả tốt nhất và cập nhật `dp[mask][current]` nếu tìm được chi phí nhỏ hơn.

4. **Truy vết đường đi:** Khi cập nhật trạng thái tốt nhất, ghi nhận `next_city[mask][current]` để xây dựng đường đi tối ưu về sau.

Các bước thực hiện:

1. **Xác định số lượng đỉnh và đỉnh bắt đầu:**

```
int n = vertexList.size();
int start = getIndex(start_char, vertexList);
```

   Lấy tổng số đỉnh và chỉ số của đỉnh xuất phát từ ký tự.

2. **Khởi tạo bảng quy hoạch động:**

```
for (int mask = 0; mask < (1 << n); ++mask)
    for (int i = 0; i < n; ++i)
        dp[mask][i] = next_city[mask][i] = -1;
```

   Đặt toàn bộ giá trị trong `dp` và `next_city` về `-1` để chuẩn bị cho thuật toán TSP.

3. **Gọi hàm `TSP(...)` để tính chi phí tối ưu:**

```
int optimalCost = TSP(edges, vertexList, 1 << start, start, start);
```

   Bắt đầu từ trạng thái chỉ có đỉnh `start` đã được thăm.

4. **Truy vết hành trình tối ưu từ bảng `next_city`:**

```
std::string resultPath(1, vertexList[start]);
int visited = 1 << start;
int current = start;

while (true) {
    int nxt = next_city[visited][current];
    if (nxt == -1)
        break;
    resultPath += " ";
    resultPath += vertexList[nxt];
    visited |= (1 << nxt);
    current = nxt;
}
```

   Duyệt các trạng thái tiếp theo được lưu sẵn trong bảng `next_city` để xây dựng lại hành trình đi qua tất cả các đỉnh.

5. **Kiểm tra quay về đỉnh xuất phát (nếu có):**

```
if (weight[current][start] < INF)
    resultPath += " " + std::string(1, vertexList[start]);
```

   Nếu tồn tại cạnh nối từ đỉnh cuối về đỉnh xuất phát, thêm đỉnh này vào chuỗi kết quả.

6. **Trả về kết quả cuối cùng:**

```
return resultPath;
```

   Trả về chuỗi đường đi tối ưu dạng danh sách các đỉnh cách nhau bằng khoảng trắng.

2.7 Thuật toán Nearest Neighbor Heuristic + 2-Opt (tham lam đỉnh gần nhất + cải tiến 2-Opt)

2.7.1 Mô tả

- Tên: Nearest Neighbor Heuristic + 2-Opt Local Search
- Mô tả: Nearest Neighbor: bắt đầu từ một đỉnh, mỗi bước chọn đỉnh chưa thăm gần nhất. 2-Opt: cải thiện tour bằng cách hoán đổi 2 cạnh để giảm chi phí.
- Nearest Neighbor: $O(n)$

  2-Opt: thường $O(n)$ mỗi lần cải tiến, lặp lại đến khi không còn cải thiện.
- **Ý tưởng:**

  1. **Tạo đường đi ban đầu** – `createInitialTour(int n, int start)`
     - **Mục đích:** Tạo một tour ban đầu bắt đầu từ đỉnh `start`, đi qua tất cả các đỉnh còn lại và quay về điểm xuất phát.
     - **Cách hoạt động:**
       * Thêm `start` vào đầu danh sách tour.
       * Duyệt các đỉnh còn lại (khác `start`) theo thứ tự và thêm vào tour.
       * Thêm lại `start` ở cuối tour để tạo chu trình.
  2. **Tối ưu hóa bằng thuật toán 2-OPT** – `twoOpt(std::vector<int> tour)`
     - **Mục đích:** Cải thiện tour hiện tại bằng cách hoán đổi hai cạnh (a, b) và (c, d) thành (a, c) và (b, d) nếu giúp giảm tổng chi phí.
     - **Thuật toán:**

       (a) Lặp đến khi không còn cải tiến:
       * Duyệt tất cả cặp cạnh (a, b) và (c, d) trong tour (không trùng nhau).
       * Tính chi phí:

$$\texttt{costBefore} = \texttt{dist[a][b]} + \texttt{dist[c][d]}$$

$$\texttt{costAfter} = \texttt{dist[a][c]} + \texttt{dist[b][d]}$$

* Nếu `costAfter` < `costBefore`, thực hiện hoán đổi (2-opt move): đảo ngược đoạn giữa `b` và `c`.

(b) Kết thúc vòng lặp khi không còn cải thiện được nữa. Trả về tour tối ưu hiện tại.

2.7.2 Thực hiện hàm `TravelingGreedy`

Tham số các hàm

- `std::vector<int> createInitialTour(int n, int start)`
  - `n`: Số lượng đỉnh trong đồ thị.
  - `start`: Chỉ số của đỉnh bắt đầu trong tour.
  - **Kết quả:** Trả về một tour ban đầu theo thứ tự tuần tự các đỉnh khác (trừ `start`) và kết thúc bằng chính `start`.
- `std::vector<int> twoOpt(std::vector<int> tour)`
  - `tour`: Đường đi ban đầu (danh sách chỉ số các đỉnh), kết thúc bằng đỉnh xuất phát.
  - **Kết quả:** Trả về tour sau khi đã được tối ưu bằng thuật toán 2-OPT.
- `std::string TravelingGreedy(...)`

```
std::string TravelingGreedy(int edgeList[][3], int numEdges, char start_char,
    const std::vector<char>& vertexList, const std::vector<Edge>& edges) {...}
```

  - `n`: Số lượng đỉnh trong đồ thị. Dùng để xác định kích thước của tour.
  - `start`: Chỉ số của đỉnh xuất phát trong tour (ví dụ: nếu bắt đầu từ đỉnh thứ 2 thì `start = 1`).
  - `tour`: Danh sách các chỉ số đỉnh, đại diện cho thứ tự các đỉnh trong hành trình. Đỉnh đầu tiên và cuối cùng thường giống nhau (chu trình).
  - `edgeList[][3]`: Ma trận gồm các cạnh của đồ thị. Mỗi hàng chứa bộ ba (u, v, w) tương ứng: đỉnh đầu, đỉnh cuối và trọng số.
  - `numEdges`: Tổng số cạnh trong `edgeList`.
  - `start_char`: Đỉnh bắt đầu, truyền vào dưới dạng ký tự (ví dụ: 'A', 'B', 'C',...).

- `vertexList`: Danh sách đỉnh biểu diễn bằng ký tự, dùng để chuyển đổi giữa chỉ số và tên đỉnh.
- `edges`: Danh sách cạnh được đóng gói dưới dạng cấu trúc `Edge` (thường gồm `u, v, w`).
- **Mục tiêu:** Trả về tour (chuỗi ký tự của các đỉnh) được xây dựng từ `edgeList`, bắt đầu từ `start_char`, sử dụng thuật toán tham lam (greedy) kết hợp tối ưu hóa 2-OPT.
- **Kết quả:** Chuỗi các đỉnh cách nhau bằng khoảng trắng nếu thành công, hoặc chuỗi `"Error: No valid cycle found"` nếu không tồn tại chu trình hợp lệ.

Các bước thực hiện:

1. **Xác định số lượng đỉnh và đỉnh bắt đầu**

```
int n = vertexList.size();
int start = getIndex(start_char, vertexList);
```

Tìm chỉ số tương ứng với ký tự `start_char` trong danh sách đỉnh `vertexList`.

2. **Tạo tour ban đầu**

```
std::vector<int> tour = createInitialTour(n, start);
```

Tạo tour ban đầu: bắt đầu từ `start`, đi qua các đỉnh còn lại theo thứ tự và quay lại `start`.

3. **Tối ưu tour với thuật toán 2-OPT**

```
tour = twoOpt(tour);
```

Duyệt tất cả các cặp cạnh trong tour. Nếu hoán đổi hai cạnh (a, b) và (c, d) thành (a, c) và (b, d) giúp giảm chi phí và các đoạn này có tồn tại đường đi (trọng số $<$ `INF`), thì thực hiện phép đảo ngược đoạn (2-opt move). Lặp lại cho đến khi không còn cải tiến nào nữa.

4. **Kiểm tra tính hợp lệ của tour**

```
bool validCycle = true;
for (int i = 0; i + 1 < tour.size(); ++i) {
    if (weight[tour[i]][tour[i + 1]] >= INF) {
        validCycle = false;
        break;
    }
}
```

Nếu tồn tại cặp đỉnh liên tiếp không có cạnh nối (trọng số là `INF`), xem như tour không hợp lệ.

5. **Tạo chuỗi kết quả trả về**

```
if (validCycle) {
    for (int i = 0; i < tour.size(); ++i) {
        if (i) result += " ";
        result += vertexList[tour[i]];
    }
} else {
    result = "Error: No valid cycle found";
}
```

Nếu tour hợp lệ, ánh xạ các chỉ số đỉnh sang ký tự tương ứng trong `vertexList` và nối thành chuỗi kết quả. Nếu không, trả về thông báo lỗi.

```
std::string Traveling(int edgeList[][3], int numEdges, char start_char)
```

Tham số đầu vào:

- `edgeList`: Ma trận 2 chiều, mỗi dòng gồm 3 phần tử `[from, to, cost]` — trong đó `from` và `to` là mã ASCII của ký tự đỉnh.
- `numEdges`: Số lượng cạnh trong `edgeList`.
- `start_char`: Ký tự đại diện cho đỉnh bắt đầu của hành trình TSP.

Các bước thực hiện chính:

1. **Khởi tạo lại ma trận trọng số `weight`**

```
for (int i = 0; i < 25; ++i)
    for (int j = 0; j < 25; ++j)
        weight[i][j] = INF;
```

Gán toàn bộ trọng số các cặp đỉnh là `INF` (không có đường đi) để đảm bảo không bị ảnh hưởng từ lần chạy trước.

2. **Xây dựng danh sách đỉnh `vertexList` từ `edgeList`**

```
for (int i = 0; i < numEdges; ++i) {
    char from = (char)edgeList[i][0];
    char to = (char)edgeList[i][1];

    if (std::find(vertexList.begin(), vertexList.end(), from) == vertexList.end()
        )
    vertexList.push_back(from);
    if (std::find(vertexList.begin(), vertexList.end(), to) == vertexList.end())
    vertexList.push_back(to);
}
```

Trích các đỉnh từ cạnh và đưa vào `vertexList`, tránh trùng lặp.

3. **Sắp xếp danh sách đỉnh**

```
std::sort(vertexList.begin(), vertexList.end());
```

Đảm bảo thứ tự ổn định để việc tra cứu chỉ số bằng `getIndex()` được chính xác.

4. **Gán trọng số vào `weight` và tạo danh sách cạnh `edges`**

```
for (int i = 0; i < numEdges; ++i) {
    int u = getIndex(...);
    int v = getIndex(...);
    int w = edgeList[i][2];

    weight[u][v] = w;
    edges.push_back({u, v, w});
}
```

Ánh xạ từ ký tự sang chỉ số, gán trọng số vào ma trận và lưu cạnh vào danh sách `edges` để dùng với các thuật toán khác.

5. **Lựa chọn thuật toán TSP phù hợp với số đỉnh**

```
int nodeCount = vertexList.size();

if (nodeCount <= 15)
    return TravelingBacktrack(...);
else if (nodeCount <= 20)
    return TravelingDP(...);
else
    return TravelingGreedy(...);
```

Dựa vào số lượng đỉnh trong đồ thị, lựa chọn thuật toán tương ứng để giải bài toán TSP hiệu quả.

2.8 Hiện thực với các testcase

2.8.1 Traveling 15_1

Traveling15_1.txt

```
int main() {
    int edgeList[1000][3];
    int numEdges = 0;

    ifstream fin("Traveling15_1.txt");
    while (fin >> edgeList[numEdges][0] >> edgeList[numEdges][1] >> edgeList[numEdges
        ][2])
        numEdges++;
    fin.close();

    char start = 'K';
```

```
    cout << Traveling(edgeList, numEdges, start) << endl;

    return 0;
}
```

Output: K F d s ; - A T u H r ‘ X ! ^ K

2.8.2 Traveling15_2

Traveling15_2.txt

```
int main() {
    int edgeList[1000][3];
    int numEdges = 0;

    ifstream fin("Traveling15_2.txt");
    while (fin >> edgeList[numEdges][0] >> edgeList[numEdges][1] >> edgeList[numEdges
        ][2])
        numEdges++;
    fin.close();

    char start = 'l';
    cout << Traveling(edgeList, numEdges, start) << endl;

    return 0;
}
```

Output: l e G p 6 ~ D 3 A) , i (# E l

2.8.3 Traveling20_1

Traveling20_1.txt

```
int main() {
    int edgeList[1000][3];
    int numEdges = 0;

    ifstream fin("Traveling20_1.txt");
    while (fin >> edgeList[numEdges][0] >> edgeList[numEdges][1] >> edgeList[numEdges
        ][2])
        numEdges++;
    fin.close();
```

```
    char start = 'm';
    cout << Traveling(edgeList, numEdges, start) << endl;

    return 0;
}
```

Output: m v ! . # S D X @ H O R \ Z t T A 6 C z m

2.8.4 Traveling20_2

Traveling20_2.txt

```
int main() {
    int edgeList[1000][3];
    int numEdges = 0;

    ifstream fin("Traveling20_2.txt");
    while (fin >> edgeList[numEdges][0] >> edgeList[numEdges][1] >> edgeList[numEdges
        ][2])
        numEdges++;
    fin.close();

    char start = '9';
    cout << Traveling(edgeList, numEdges, start) << endl;

    return 0;
}
```

Output: 9 s % [& / R) l V q $ (+ P * E Y N j 9

2.8.5 Traveling25

Traveling25.txt

```
int main() {
    int edgeList[1000][3];
    int numEdges = 0;

    ifstream fin("Traveling25.txt");
    while (fin >> edgeList[numEdges][0] >> edgeList[numEdges][1] >> edgeList[numEdges
        ][2])
        numEdges++;
    fin.close();
```

```
    char start = 'v';
    cout << Traveling(edgeList, numEdges, start) << endl;

    return 0;
}
```

Output: v { w I M 6 ? # q S (9 W x G c [J y ‘ ; % B F] v

2.9 Kết luận

Bài toán Traveling Salesman Problem (TSP) là một trong những bài toán kinh điển thuộc lĩnh vực tối ưu tổ hợp, với độ phức tạp thuộc lớp NP-Hard. Trong đồ án này, chúng tôi đã tiếp cận bài toán thông qua ba phương pháp giải khác nhau tùy theo số lượng đỉnh trong đồ thị: Backtracking (cho trường hợp nhỏ), Lập trình động (DP) (cho số lượng đỉnh trung bình), và heuristic với Two-Opt (cho trường hợp lớn).

Cụ thể, thuật toán Backtracking được áp dụng cho các trường hợp có số lượng đỉnh nhỏ ($\leq$ 15), cho phép tìm ra chu trình ngắn nhất một cách chính xác bằng cách liệt kê và kiểm tra toàn bộ các hoán vị có thể. Đối với đồ thị có số đỉnh từ 16 đến 20, thuật toán lập trình động với bitmask giúp giảm đáng kể thời gian xử lý nhờ lưu trữ kết quả trung gian, đảm bảo vẫn tìm được lời giải tối ưu. Cuối cùng, với đồ thị có số lượng đỉnh lớn hơn, thuật toán heuristic Two-Opt được sử dụng để tìm lời giải gần tối ưu trong thời gian hợp lý.

Kết quả thu được từ chương trình cho thấy sự hiệu quả và phù hợp của từng thuật toán với quy mô dữ liệu khác nhau. Tuy nhiên, cũng cần lưu ý rằng mỗi phương pháp đều có giới hạn về hiệu suất và độ chính xác. Trong tương lai, có thể xem xét cải tiến bằng cách áp dụng các thuật toán metaheuristic như Genetic Algorithm, Ant Colony Optimization hoặc Simulated Annealing để xử lý các trường hợp dữ liệu lớn hơn với độ chính xác tốt hơn.

Tiêu chí	**Backtracking**	**Lập trình động (Bitmask DP)**	**Heuristic** (NN, 2-Opt...)
Độ chính xác lời giải	Tối ưu tuyệt đối	Tối ưu tuyệt đối	Gần tối ưu, sai lệch vài %
Độ phức tạp thời gian	$O(n!)$	$O(n^2 \times 2^n)$	$O(n^2)$ (tùy kỹ thuật)
Mức tiêu thụ bộ nhớ	Thấp	Cao (mảng DP 2 chiều)	Thấp đến Trung bình
Tính mở rộng (scalability)	Rất kém ($n \leq 10$)	Giới hạn ($n \leq 20$)	Tốt (n hàng trăm vẫn dùng được)
Tính linh hoạt	Ít linh hoạt	Ít linh hoạt	Dễ áp dụng cho biến thể TSP
Dễ cài đặt	Dễ, cần đệ quy	Phức tạp, dùng bitmask	Rất dễ (NN, 2-Opt)
Tốc độ chạy thực tế	Chậm	Trung bình	Nhanh
Tối ưu thêm được không?	Không cần (đã tối ưu)	Không cần	Có thể kết hợp thuật toán khác
Phù hợp cho bài toán học thuật	Tốt để học thuật toán	Hiểu sâu về tối ưu	Không phản ánh đầy đủ lý thuyết
Phù hợp cho ứng dụng thực tế	Không	Có thể dùng nếu n nhỏ	dùng trong AI,logistics, vận tải,...

3 Ứng dụng thực tế

Bài toán người du lịch (TSP) không chỉ là một vấn đề trừu tượng trong toán học, mà còn có ứng dụng sâu rộng trong nhiều lĩnh vực thực tiễn. TSP đóng vai trò quan trọng trong:

- Quản lý tuyến đường và tối ưu hoá lộ trình giao hàng trong logistics.
- Thiết kế vi mạch điện tử để tối ưu hoá chiều dài dây kết nối.
- Lập kế hoạch lịch trình trong sản xuất và quản lý thời gian.
- Tối ưu hóa lộ trình của robot trong các hệ thống tự động hóa.

Việc áp dụng TSP giúp giảm chi phí, tăng hiệu suất và sử dụng tài nguyên hiệu quả hơn trong thực tế.

4 Thách thức và cơ hội

TSP là một trong những bài toán NP-hard nổi bật nhất, đặc biệt trở nên thách thức khi số lượng đỉnh tăng lên. Việc tìm lời giải tối ưu trong thời gian hợp lý là điều không khả thi với các phương pháp vét cạn truyền thống khi bài toán mở rộng.

Tuy nhiên, chính tính chất này cũng mở ra nhiều cơ hội nghiên cứu trong lĩnh vực:

- Tối ưu hóa tổ hợp.
- Thuật toán gần đúng (approximation).
- Các phương pháp heuristic và metaheuristic như di truyền, kiến, mô phỏng tôi luyện.

Từ đó, thúc đẩy sự phát triển các giải pháp hiệu quả hơn cho nhiều bài toán trong thực tế.

5 Tài Liệu Tham Khảo

1. David L. Applegate, Robert E. Bixby, Vasek Chvatal, William J. Cook, The Traveling Salesman Problem: A Computational Study.
2. Marco Dorigo, Thomas Stützle, Ant Colony Optimization.
3. David E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning.
4. GeeksforGeeks – TSP using Backtracking
5. Brute Force TSP Explanation – Brilliant.org
6. CP-Algorithms – TSP Bitmask DP
7. GeeksforGeeks – TSP using DP and Bitmasking
8. Nearest Neighbor Heuristic – GeeksforGeeks
9. 2-Opt and Greedy Improvements – Visual Guide