ĐẠI HỌC QUỐC GIA VIỆT NAM, THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH

DISCRETE STRUCTURE (CO1007)

Assignment

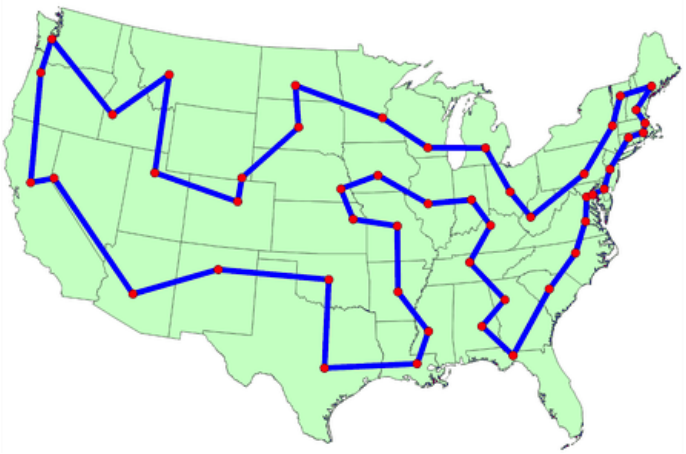
TRAVELING SALESMAN PROBLEM

Giảng viên hướng dẫn: Mai Xuân Toàn
Sinh viên: Trần Ngọc Khánh Huy - 2252265.

Mục lục

1 Giới thiệu về Traveling Salesman Problem

Bài toán Người du lịch (TSP) là một trong những bài toán nổi tiếng và khó khăn nhất trong lĩnh vực tối ưu hóa tổ hợp. Bài toán là về việc tìm đường đi ngắn nhất cho người thương nhân (salesman), hay còn gọi là người chào hàng xuất phát từ một thành phố, đi qua lần lượt tất cả các thành phố duy nhất một lần và quay về thành phố ban đầu với chi phí rẻ nhất, được phát biểu vào thế kỷ 17 bởi hai nhà toán học vương quốc Anh là Sir William Rowan Hamilton và Thomas Penyngton Kirkman, và được ghi trong cuốn giáo trình Lý thuyết đồ thị nổi tiếng của Oxford. Nó nhanh chóng trở thành bài toán khó thách thức toàn thế giới bởi độ phức tạp thuật toán tăng theo hàm số mũ (trong chuyên ngành thuật toán người ta còn gọi chúng là những bài toán NP-khó). Người ta bắt đầu thử và công bố các kết quả giải bài toán này trên máy tính từ năm 1954 (49 đỉnh), cho đến năm 2004 bài toán giải được với số đỉnh lên tới 24.978, và dự báo sẽ còn tiếp tục tăng cao nữa. Bài toán có thể phát biểu dưới ngôn ngữ đồ thị như sau : Cho đồ thị n đỉnh đầy đủ và có trọng số G=(V-tập đỉnh,E-tập cạnh) có hoặc vô hướng.



Hình 1: Sơ đồ minh họa TSP

1.1 Mô tả bài toán TSP

- Cho một danh sách các thành phố và khoảng cách giữa mỗi cặp thành phố.
- Tìm một đường đi (một chu trình) đi qua tất cả các thành phố, sao cho tổng khoảng cách di chuyển là ngắn nhất.

Ví dụ:

Giả sử có 4 thành phố A, B, C và D. Khoảng cách giữa các thành phố được cho bởi bảng sau:

	A	B	C	D
A	0	2	5	3
B	2	0	4	1
C	5	4	0	6
D	3	1	6	0

Một giải pháp khả thi cho bài toán này có thể là đường đi A -> B -> D -> C -> A, với tổng khoảng cách là $2 + 1 + 6 + 5 = 14$. Tuy nhiên, đây có thể không phải là đường đi ngắn nhất.

1.2 Độ phức tạp

TSP là một bài toán NP-khó, nghĩa là không có thuật toán nào có thể giải quyết bài toán này một cách chính xác trong thời gian đa thức (tức là thời gian chạy của thuật toán tăng theo hàm mũ với kích thước của bài toán). Điều này có nghĩa là khi số lượng thành phố tăng lên, thời gian cần thiết để tìm ra giải pháp tối ưu sẽ tăng lên rất nhanh.

1.3 Ứng dụng

TSP có nhiều ứng dụng thực tế, bao gồm:

- **Lập kế hoạch lộ trình:** Tìm đường đi tối ưu cho các phương tiện giao thông, xe buýt, xe tải,...thực thi tiến trình đó cho đến khi hoàn thành.
- **Thiết kế mạch in:** Tối ưu hóa việc đặt các linh kiện trên mạch in để giảm thiểu độ dài dây dẫn.
- **Lập lịch trình:** Lập lịch trình cho các công việc, máy móc,... để tối ưu hóa thời gian và tài nguyên.
- **Phân tích DNA:** Xác định thứ tự của các đoạn DNA.

2 Hiện thực

Thuật toán: Để giải bài toán Traveling Salesman Problem này chúng ta sẽ dùng thuật toán Held-Karp.

Held-Karp Algorithm:

Thuật toán **Held-Karp** (còn gọi là Dynamic Programming Algorithm for TSP) sử dụng lập trình động và bit masking để giải quyết TSP. Thuật toán này hoạt động bằng cách duyệt qua tất cả các tập hợp con của các đỉnh và cập nhật chi phí tối thiểu để di chuyển giữa các đỉnh. Ý tưởng chính là xây dựng một bảng (table) lưu trữ chi phí tối thiểu cho tất cả các tập hợp con của các đỉnh, với mỗi tập hợp kết thúc tại một đỉnh cụ thể.

2.1 Hàm TSP: string TSP(int G[20][20], int n, int start)

Hàm chính TSP được định nghĩa để giải quyết TSP bằng cách sử dụng lập trình động và bit masking. Các bước chính trong Thuật toán **Held-Karp**:

1. **Khởi Tạo Bảng DP**

```
vector<vector<int>> dp(1 << n, vector<int>(n, INF));
vector<vector<int>> parent(1 << n, vector<int>(n, -1));
dp[1 << start][start] = 0;
```

- **dp**: Một vector 2D, trong đó **dp[mask][i]** đại diện cho chi phí tối thiểu để thăm tập hợp các thành phố được đánh dấu bởi **mask** và kết thúc tại thành phố **i**.
- **parent**: Một vector 2D để theo dõi đường đi. **parent[mask][i]** lưu trữ thành phố trước đó được thăm trước thành phố **i**.
- **1 « n**: Đại diện cho tất cả các tập hợp con của các thành phố (sử dụng bit masking).
- **dp[1 « start][start] = 0**: Khởi tạo thành phố bắt đầu với chi phí 0.

2. **Chuyển Trạng Thái**

```
for (int mask = 0; mask < (1 << n); mask++) {
for (int u = 0; u < n; u++) {
    if ((mask & (1 << u)) == 0) {
        continue;
    }
    for (int v = 0; v < n; v++) {
        if ((mask & (1 << v)) != 0) {
            continue;
        }
        if (G[u][v] != 0 && dp[mask][u] != INF) {
            int newMask = mask | (1 << v);
            if (dp[newMask][v] > dp[mask][u] + G[u][v]) {
                dp[newMask][v] = dp[mask][u] + G[u][v];
                parent[newMask][v] = u;
            }
        }
    }
}
}
```

- Vòng lặp ngoài duyệt qua tất cả các tập hợp con của các thành phố **(mask)**.
- Vòng lặp giữa duyệt qua mỗi thành phố **u** trong tập hợp con hiện tại.
- Vòng lặp trong xem xét việc thêm một thành phố **v** vào tập hợp con hiện tại **mask**.
- Nếu thành phố **v** không có trong tập hợp (**mask & (1 « v) == 0**), nó sẽ cập nhật các mảng **dp** và **parent** nếu tìm thấy đường đi ngắn hơn.

3. **Tìm Đường Đi Ngắn Nhất (có chi phí tối thiểu)**

```
int mask = (1 << n) - 1;
int last = start;
for (int i = 0; i < n; i++) {
    if (dp[mask][i] + G[i][start] < dp[mask][last] + G[last][start]) {
        last = i;
    }
}
```

- **Mask** cuối cùng đại diện cho tất cả các thành phố đã được thăm **((1 « n) - 1)**.
- Tìm thành phố **last** mà cho chi phí tối thiểu để quay lại thành phố bắt đầu.

4. **Truy Vết Lại Đường Đi**

```
vector<int> path;
while (last != -1) {
    path.push_back(last);
    int next = parent[mask][last];
    mask = mask ^ (1 << last);
    last = next;
}
reverse(path.begin(), path.end());
```

- Đường đi được tái tạo bằng cách theo dõi mảng **parent** từ thành phố cuối cùng đến thành phố bắt đầu.
- Đường đi sau đó được đảo ngược để có thứ tự chính xác.

Bonus bước Định Dạng Kết Quả:

```
string result;
for (int i : path) {
    result += static_cast<char>(i + 'A');
    result += " ";
}
result += static_cast<char>(start + 'A');
return result;
```

- Chuyển đổi các chỉ số thành phố thành các ký tự (**A**, **B**, v.v.) và thêm chúng vào chuỗi kết quả.
- Trả về chuỗi kết quả đại diện cho đường đi.

2.2 Hàm Traveling: string Traveling(int G[20][20], int n, char start)

```
int startIndex = start - 'A';
string result = TSP(G, n, startIndex);
return result;
```

- Chuyển đổi ký tự thành phố bắt đầu thành chỉ số.
- Gọi hàm **TSP** với các tham số phù hợp.
- Trả về chuỗi kết quả.

Đặc điểm:

- Thuật toán **Held-Karp** có độ phức tạp thời gian là $O(n^2 \cdot 2^n)$, trong đó n là số lượng thành phố. Đây là một cải tiến lớn so với giải pháp duyệt toàn bộ (**brute-force**) với độ phức tạp là $O(n!)$.
- **Phù hợp cho đầu vào trung bình**: Phương pháp này có thể xử lý kích thước đầu vào lên tới khoảng 20 nút một cách hiệu quả. Mặc dù vậy, thuật toán này vẫn không phù hợp cho các bài toán có số lượng thành phố rất lớn do sự tăng trưởng theo hàm mũ của thời gian chạy.

2.3 Input-Output

Với **input** như sau:

```
int num_vertices = 12;
int GraphTVL[20][20] = {
    {0, 85, 0, 0, 0, 0, 92, 0, 50, 83, 21, 0},
    {85, 0, 0, 0, 0, 40, 0, 0, 0, 35, 89, 0},
    {0, 0, 0, 42, 97, 0, 55, 56, 0, 0, 0, 28},
    {0, 0, 42, 0, 0, 22, 0, 0, 0, 0, 0, 69},
    {0, 0, 97, 0, 0, 78, 0, 0, 40, 0, 0, 0},
    {0, 40, 0, 22, 78, 0, 49, 0, 0, 0, 63, 14},
    {92, 0, 55, 0, 0, 49, 0, 53, 0, 72, 0, 25},
    {0, 0, 56, 0, 0, 0, 53, 0, 31, 0, 78, 0},
    {50, 0, 0, 0, 40, 0, 0, 31, 0, 93, 0, 61},
    {83, 35, 0, 0, 0, 0, 72, 0, 93, 0, 77, 0},
    {21, 89, 0, 0, 0, 63, 0, 78, 0, 77, 0, 84},
    {0, 0, 28, 69, 0, 14, 25, 0, 61, 0, 84, 0}};
```

Áp dụng giải thuật **Held-Karp** như trên, ta được kết quả:

```
TSP Functions: A K J B F D C L G H I E A
-----------------------------------
```

2.4 Kết luận

Cách làm này giải quyết TSP một cách hiệu quả bằng cách sử dụng lập trình động và bit masking, cung cấp một cách tiếp cận chi tiết từng bước để tìm đường đi ngắn nhất thăm tất cả các thành phố một lần và quay lại điểm bắt đầu. Hàm Traveling cung cấp một cách tiện lợi để tương tác với thuật toán bằng cách sử dụng tên thành phố dưới dạng ký tự.

3 Toàn bộ Source Code

Link source code trên GitHub

4 Nguồn tham khảo

(i) Held-Karp Algorithm
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm

https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/

https://viblo.asia/p/giai-bai-toan-nguoi-du-lich-noi-tieng-bang-mo-phong-hanh-vi-cua-dan-kien-