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

Cấu trúc rời rạc cho KHMT : CO1007

BÁO CÁO BÀI TẬP LỚN

Họ và tên: Nguyễn Duy Mạnh
MSSV: 2412024
Giảng viên: Mai Xuân Toàn

TP. Hồ Chí Minh, ngày 21 tháng 6 năm 2025

Mục lục

1. Sơ lược về bài toán

Bài toán Travelling Salesman Problem (TSP) là một bài toán quan trọng trong lĩnh vực tối ưu hóa, yêu cầu tìm lộ trình ngắn nhất để đi qua tất cả các thành phố một lần và quay trở lại điểm xuất phát. Mặc dù đơn giản về định nghĩa, nhưng việc tìm lời giải tối ưu cho TSP đòi hỏi sự phức tạp và là một thách thức lớn trong tính toán. Bài toán này có ứng dụng rộng rãi, từ quản lý vận tải, lập kế hoạch giao hàng đến thiết kế mạng, thu hút sự quan tâm của nhiều nhà nghiên cứu và kỹ sư.

2. Xác định yêu cầu của bài toán

2.1. Yêu cầu

Viết hàm Traveling() tính toán và xuất ra lộ trình ngắn nhất để đi qua tất cả các đỉnh trong đồ thị và quay trở lại đỉnh xuất phát.

2.2. Input

Đồ Thị (Graph): Đồ thị biểu diễn dang sách các cạnh và trọng số. Trong đó, đường đi giữa các đỉnh được liên kết sẽ có trọng số dương, mỗi cạnh được sẽ bao gồm 3 số nguyên đại diện cho đỉnh bắt đầu, đỉnh kết thúc và trọng số.
Các đỉnh bắt đầu và kết thúc cần được đọc dưới dạng kí tự mã ASCII (0-255).
Đồ thị đầu vào sẽ được xử lý như một đồ thị có hướng (Dạng dữ liệu : int EdgeList[MAX][3]).

2.3. Ouput

Chu trình ngắn nhất để đi qua tất cả các đỉnh trong đồ thị, bắt đầu từ đỉnh xuất phát và quay trở lại đỉnh xuất phát (Hamilton cycle).

3. Thuật toán Held-Karp(Quy hoạch động)

3.1. Giới thiệu

Thuật toán Held-Karp là một quy hoạch động (Dynamic Programming) hiệu quả cho bài toán Người bán hàng (TSP) với số lượng đỉnh tương đối nhỏ (thường ≤ 21). Thuật toán này tận dụng kỹ thuật bitmasking để biểu diễn trạng thái của các đỉnh đã thăm, kết hợp với lưu trữ con đường ngắn nhất tại mỗi trạng thái, giúp giảm đáng kể số lượng phép tính so với phương pháp vét cạn.

3.2. Ý tưởng chính

Giả sử có **n** đỉnh, với index từ **0** tới **n-1** và kí hiệu từ **0** tới **n** (theo bảng mã ASCII). Ta muốn tìm đường đi ngắn nhất ví dụ từ đỉnh A tới các đỉnh còn lại đúng một lần và quay lại đỉnh ban đầu.
Thuật toán lưu trữ một mảng **dp[mask][current]** :

- **mask** biểu diễn các đỉnh đã được đi tới dưới dạng bit.
- **current** : đỉnh hiện tại.
- **dp[mask][current]** : chi phí nhỏ nhất để đi qua các đỉnh trong mask.

Công thức chuyển trạng thái:

$$\mathbf{dp[mask][u] = min(dp[mask][u], dp[premask][v] + cost[v][u])}$$

(u - đỉnh hiện tại, v - đỉnh liền trước u, premask = mask $\oplus$ (1 « u))

Ta tạo một mảng 2 chiều dp[mask][u] chứa chi phí nhỏ nhất để đi từ điểm bắt đầu đi qua các đỉnh được đánh dấu trong mask và kết thúc ở u. Và một mảng 2 chiều khác để chứa các đỉnh đã được đi qua để truy hồi.

```
// INF đại diện cho một số cực kì lớn
   std::vector<std::vector<int>>dp(1 << n, std::vector<int>(n,INF));
   std::vector<std::vector<int>>trace(1<<n, std::vector<int>(n, -1));
```

Ta có công thức quy hoạch động như sau :

```
   int premask = mask ^ (1 << u);
   dp[mask][u] = std::min(dp[mask][u], dp[premask][v]+cost[v][u]);
```

Đây là một cách giải vấn đề bằng quy hoạch động từ dưới lên (dp bottom-up). Như vậy sau khi xử lý xong ta phải duyệt lại từ cuối lên để lấy lại các đỉnh đã duyệt qua.

```
    std::vector<int> path;
    int current_mask = final_mask;
    int current_vert = last_vert;
    while (current_vert != -1)
    {
       path.push_back(Vertices_List[current_vert]);
       int prev_vert = trace[current_mask][current_vert];
       current_mask ^= (1 << current_vert);
       current_vert = prev_vert;
     }
```

Do quá trình quá trình truy hồi ngược lại với đường đã đi nên ta phải đảo ngược lại path. `std::reverse(path.begin(),path.end());`
Sau đó ta sẽ chuyển lại các đỉnh được lưu trữ dưới dạng `int` thành `char` vào trong một biến `std::string output`

```
std::string output{};
 for(int i = 0 ; i < path.size(); i++){
    if(i > 0){
      output.push_back(' ');
    }
    output.push_back((char)path[i]);
}
output.push_back(' ');
output.push_back(start_vert);
```

3.3. Kết luận

Độ chính xác của thuật toán Held-Karp là tuyệt đối, nghĩa là sẽ luôn cho ra kết quả đúng với mọi đồ thị.
Độ phức tạp của thuật toán Held-Karp là : $O(n^2 \cdot 2^n)$. Nhanh hơn nhiều so với duyệt tất cả các tổ hợp O(n!). Nhưng có thể nhận thấy rằng thời gian chạy và bộ nhớ của thuật toán sẽ tăng theo cấp số mũ so với số lượng đỉnh, nên sẽ không khả thi với số lượng định lớn (> 21 đỉnh).

4. Phương án tính xấp xỉ đối với đồ thị có nhiều đỉnh

4.1. Giới thiệu

Giải thuật "tổ kiến" (Ant Colony Optimization-ACO), là một giải thuật sử dụng phương pháp suy nghiệm (heuristic) mô phỏng hành vi tìm đường của của kiến trong tự nhiên để giải các bài toán tổ hợp, đặc biệt là TSP.

4.2. Ý tưởng chính

Trong tự nhiên, kiến tìm đường từ tổ đến nguồn thức ăn bằng cách rải pheromone. Đường nào nhiều pheromone thì càng có khả năng được kiến chọn. Sau nhiều lần lặp, đường ngắn nhất sẽ có nhiều pheromone nhất, vì kiến di chuyển nhanh hơn, rải nhiều hơn.
Những phương án tối ưu sẽ được thêm pheromone còn những phương án còn lại sẽ mất bớt heromone theo một số thông số được định nghĩa sẵn. Giải thuật sẽ được chạy nhiều lần để tránh việc đàn kiến nhận diện đường đi tối ưu quá nhanh.
Các thông số được định nghĩa sẵn bao gồm :

```
const int numLoop = 200; // số lần lặp mỗi lần chạy
const int numAnts = n; // số lượng đỉnh, luồng , hay kiến
const double alpha = 1.0; // hệ số ảnh hưởng của pheromone.
const double beta = 2; // hệ số heuristic.
const double rate = 0.1; // tốc độ bay hơi pheromone.
const double Q = 100.0; // lượng pheromone thêm vào mỗi cạnh.
const double initialPheromone = 1e-3; // lượng pheromone ban đầu.
```

Input sẽ được ưu tiên đưa về danh sách kề.

4.3. Kết luận

ACO là một thuật toán mạnh mẽ, tuy là một thuật toán heuristic nhưng ACO đảm bảo sẽ ra được những đường đi gần như tối ưu nên sẽ được ưu tiên sử dụng với số lượng luồng(đỉnh) lớn. Thông thường trong thực tế người ta chỉ quan tâm tới những phương pháp gần tối ưu, hoặc đủ tốt do các vấn đề trong thực tế quá phức tạp, hoặc có nhiều biến số ngẫu nhiên ảnh hưởng nên những thuật toán như ACO.

5. Các nguồn tham khảo

[1] *Tìm hiểu đôi chút về giải thuật đàn kiến (Ant Colony Optimization)*, Viblo, 2020.
https://viblo.asia/p/tim-hieu-doi-chut-ve-giai-thuat-dan-kien

[2] *Bitmasking and Dynamic Programming | Travelling Salesman Problem*, GeeksForGeeks, 2024
https://www.geeksforgeeks.org/dsa/bitmasking-dynamic-programming

[3] *The Traveling Salesman Problem: When Good Enough Beats Perfect*, Reducible
https://www.youtube.com/watch?v=GiDsjIBOVoA