ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

BÀI TẬP NHÓM 15
BỞI NHÓM 12

Giáo viên hướng dẫn: ***Nguyễn Thanh Sơn***

Nhóm 12:

- ***Hoàng Minh Thái*** **- 23521414**
- ***Nguyễn Trọng Tất Thành*** **- 2352**

Mục lục

GIẢI BÀI TẬP BỞI NHÓM 12

Bài 1

Liên minh Hansa (Hanseatic League) là một mạng lưới các thành phố thương mại ven biển ở Bắc Âu, hình thành và phát triển mạnh mẽ trong suốt thế kỷ 12 đến 17. Bạn là một thương nhân đang trên hành trình từ London đến Novgorod.

Sử dụng đồ thị dưới đây (trọng số trên cạnh là chi phí di chuyển thực tế, trọng số tại node là khoảng cách Euclide đến Novgorod). Chúng ta sẽ sử dụng hai thuật toán để tìm kiếm đường đi từ London đến Novgorod:

1. **Thuật toán Greedy**
2. **Thuật toán UCS (Uniform Cost Search)**

Giải thuật

Thuật toán Greedy

Thuật toán Greedy tìm kiếm đường đi bằng cách chọn thành phố kế tiếp có khoảng cách Euclide gần nhất đến Novgorod mà không quan tâm đến chi phí di chuyển thực tế.

Cách thức làm việc của thuật toán Greedy:

- Mỗi bước, thuật toán chọn thành phố có khoảng cách Euclide đến Novgorod nhỏ nhất.
- Không xét đến chi phí di chuyển thực tế.
- Thuật toán dừng lại khi đến Novgorod.

Ví dụ: Giả sử chúng ta có đồ thị sau:

$$\text{London} \xrightarrow{8} \text{Amsterdam} \xrightarrow{6} \text{Novgorod}$$

$$\text{London} \xrightarrow{10} \text{Berlin} \xrightarrow{4} \text{Novgorod}$$

- Bước 1: Bắt đầu từ London, chọn thành phố gần nhất (Amsterdam với khoảng cách Euclide ngắn nhất).
- Bước 2: Từ Amsterdam, đi đến Novgorod.

Kết quả:

- Tuyến đường: London - Amsterdam - Novgorod.
- Chi phí tổng: 8 (London đến Amsterdam) + 6 (Amsterdam đến Novgorod) = 14.

Thuật toán UCS

Thuật toán UCS tìm kiếm đường đi tối ưu bằng cách lựa chọn thành phố có chi phí di chuyển nhỏ nhất từ điểm xuất phát.

Cách thức làm việc của thuật toán UCS:

- Mỗi bước, thuật toán chọn thành phố có chi phí di chuyển nhỏ nhất.
- Thuật toán dừng lại khi đến Novgorod hoặc không còn thành phố nào có thể di chuyển đến.

Ví dụ: Giả sử chúng ta có đồ thị sau:

$$\text{London} \xrightarrow{8} \text{Amsterdam} \xrightarrow{6} \text{Novgorod}$$
$$\text{London} \xrightarrow{10} \text{Berlin} \xrightarrow{4} \text{Novgorod}$$

- Bước 1: Bắt đầu từ London, chọn thành phố có chi phí thấp nhất (Amsterdam với chi phí 8).
- Bước 2: Từ Amsterdam, đi đến Novgorod với chi phí 6.

Kết quả:

- Tuyến đường: London - Amsterdam - Novgorod.
- Chi phí tổng: 8 (London đến Amsterdam) + 6 (Amsterdam đến Novgorod) = 14.

So sánh kết quả và tính tối ưu

Thuật toán Greedy:

- Mặc dù Greedy chọn thành phố gần nhất (theo khoảng cách Euclide), nhưng không tối ưu về chi phí di chuyển thực tế.
- Tuy nhiên, trong ví dụ trên, Greedy đã chọn đúng tuyến đường tối ưu.

Thuật toán UCS:

- UCS tối ưu hóa chi phí di chuyển thực tế, vì vậy đảm bảo tìm ra tuyến đường có chi phí thấp nhất.
- Trong ví dụ trên, UCS cũng chọn đúng tuyến đường tối ưu.

Kết luận: Cả hai thuật toán Greedy và UCS tìm ra cùng một tuyến đường với chi phí tổng là 14, nhưng UCS đảm bảo tối ưu về chi phí di chuyển thực tế trong mọi trường hợp, trong khi Greedy không phải lúc nào cũng tối ưu về chi phí.

Bài 2

Kaiser, một cảnh sát kỳ cựu, đang phải đối mặt với một thử thách khắc nghiệt để cứu vợ mình, Kayra, khỏi tay tên tội phạm thời gian. Để tìm ra nơi vợ anh ấy bị giam giữ, Kaiser phải đi qua một số thành phố được nối với nhau bằng các con đường một chiều. Tuy nhiên, một số con đường có trọng số âm (do tên tội phạm tạo ra bẫy thời gian).

Bài toán yêu cầu xác định xem có chu trình âm nào trong đồ thị hay không, nếu có thì in ra chu trình đó.

Ý tưởng và phương pháp thiết kế thuật toán

Thuật toán Bellman-Ford sẽ được sử dụng để phát hiện chu trình âm trong đồ thị có trọng số âm. Các bước thực hiện bao gồm:

- Bắt đầu từ một đỉnh nguồn và thực hiện $N - 1$ lần lặp qua tất cả các cạnh của đồ thị để cập nhật khoảng cách tối thiểu.
- Sau $N - 1$ lần lặp, nếu có sự thay đổi trong lần lặp thứ N, thì chắc chắn đồ thị chứa chu trình âm.
- Lần theo các đỉnh để tìm ra chu trình âm.

Độ phức tạp: Thuật toán Bellman-Ford có độ phức tạp thời gian là $O(N \times M)$, với N là số đỉnh và M là số cạnh trong đồ thị.

Mã giả của thuật toán

```
1. Khởi tạo khoảng cách của tất cả các đỉnh là vô cùng lớn, ngoại trừ đỉnh nguồn là 0
2. Lặp M lần, mỗi lần:
   a. Duyệt tất cả các cạnh a -> b với trọng số c.
   b. Nếu khoảng cách[a] + c < khoảng cách[b], cập nhật khoảng cách[b].
3. Sau M lần lặp, kiểm tra xem có thay đổi gì không. Nếu có, ta phát hiện chu trình âm
4. Nếu có chu trình âm, lần theo các đỉnh để tìm chu trình và in ra chu trình.
5. Nếu không có chu trình âm, in ra "NO".
```

Cài đặt thuật toán

Python code:

```
def find_negative_cycle(N, M, edges):
    # Khởi tạo khoảng cách từ nguồn tới tất cả các đỉnh là vô cùng lớn
    INF = float('inf')
    distance = [INF] * (N + 1)
    parent = [-1] * (N + 1)
    distance[1] = 0  # Giả sử đỉnh nguồn là 1

    # Bellman-Ford: Lặp qua tất cả các cạnh M lần
    for i in range(N - 1):
        for u, v, w in edges:
            if distance[u] != INF and distance[u] + w < distance[v]:
                distance[v] = distance[u] + w
                parent[v] = u

    # Kiểm tra chu trình âm: Lặp qua một lần nữa để tìm sự thay đổi
    for u, v, w in edges:
        if distance[u] != INF and distance[u] + w < distance[v]:
            # Phát hiện chu trình âm, tìm chu trình
```

```
            cycle = []
            visited = [False] * (N + 1)
            x = v
            # Lấy chu trình bằng cách lần theo các đỉnh trong chu trình
            for _ in range(N):
                x = parent[x]
            cycle_start = x
            cycle.append(cycle_start)
            current = parent[cycle_start]
            while current != cycle_start:
                cycle.append(current)
                current = parent[current]
            cycle.append(cycle_start)
            cycle.reverse()

            # In ra chu trình âm
            print("YES")
            print(" ".join(map(str, cycle)))
            return

    # Nếu không có chu trình âm
    print("NO")

# Đọc đầu vào
N, M = map(int, input().split())
edges = []
for _ in range(M):
    a, b, c = map(int, input().split())
    edges.append((a, b, c))

# Gọi hàm kiểm tra chu trình âm
find_negative_cycle(N, M, edges)
```

Giải thích mã

1. **Khởi tạo:** Mảng ‘distance‘ lưu khoảng cách từ đỉnh nguồn đến các đỉnh còn lại. Mảng ‘parent‘ lưu đỉnh cha của mỗi đỉnh để có thể truy vết chu trình nếu phát hiện. 2. **Thuật toán Bellman-Ford:** Lặp qua tất cả các cạnh M lần. Nếu tìm thấy một khoảng cách nhỏ hơn giá trị hiện tại, cập nhật khoảng cách và lưu lại đỉnh cha. 3. **Kiểm tra chu trình âm:** Sau khi lặp $N-1$ lần, nếu có thay đổi trong lần lặp thứ N, ta bắt đầu lần theo các đỉnh cha để tìm chu trình âm và in ra. 4. **Đầu ra:** Nếu phát hiện chu trình âm, in "YES"và chu trình. Nếu không có chu trình âm, in "NO".

Ví dụ

Input:

4 5

1 2 1

2 4 1

3 1 1

4 1 −3

4 3 −2

Output:

YES

1 2 4 1

Tính toán độ phức tạp

Thuật toán Bellman-Ford có độ phức tạp thời gian là $O(N \times M)$, vì phải lặp qua tất cả các cạnh M lần, và mỗi lần lặp phải kiểm tra tất cả các cạnh.

Không gian: Độ phức tạp không gian là $O(N + M)$, vì cần lưu trữ khoảng cách và các cạnh của đồ thị.

Kết luận

Thuật toán Bellman-Ford là một lựa chọn hợp lý để phát hiện chu trình âm trong đồ thị có trọng số âm. Thuật toán này có thể đảm bảo tìm ra chu trình âm nếu có, và có độ phức tạp chấp nhận được với các đồ thị có số đỉnh và cạnh lớn.