ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Khoa Khoa học máy tính

Môn học Phân Tích Và Thiết Kế Thuật Toán

Bài tập Đồ thị

Cao Lê Công Thành
MSSV: 23521437
Đặng Quang Vinh
MSSV: 23521786

Mục lục

1 Bài tập 1

Thuật toán Greedy Best-First Search

Quy tắc: Mở rộng node có giá trị heuristic nhỏ nhất (khoảng cách Euclide đến Novgorod).

Quá trình:

- Bắt đầu từ London: chọn Hamburg (giá trị heuristic nhỏ hơn so với Amsterdam).
- Từ Hamburg: chọn Falsterbo (giá trị heuristic nhỏ nhất).
- Từ Falsterbo: chọn Danzig (giá trị heuristic nhỏ nhất).
- Từ Danzig: chọn Visby (giá trị heuristic nhỏ nhất).
- Từ Visby: chọn Tallinn (giá trị heuristic nhỏ nhất).
- Từ Tallinn: chọn Novgorod (đích đến, giá trị heuristic là 0).

Đường đi:

$$\text{London} \to \text{Hamburg} \to \text{Falsterbo} \to \text{Danzig} \to \text{Visby} \to \text{Tallinn} \to \text{Novgorod}$$

Tổng chi phí thực tế:

$$801 + 324 + 498 + 606 + 590 + 474 = 3293$$

Đánh giá: Đường đi không tối ưu vì chỉ dựa trên giá trị heuristic mà bỏ qua chi phí thực tế.

Thuật toán Uniform Cost Search (UCS)

Quy tắc: Mở rộng node có tổng chi phí thực tế nhỏ nhất.

Quá trình:

- Bắt đầu từ London: chọn Amsterdam (chi phí thực tế nhỏ nhất: 395).
- Từ Amsterdam: chọn Hamburg (chi phí thực tế nhỏ nhất: 411).
- Từ Hamburg: chọn Lubeck (chi phí thực tế nhỏ nhất: 64).
- Từ Lubeck: chọn Danzig (chi phí thực tế nhỏ nhất: 262).
- Từ Danzig: chọn Visby (chi phí thực tế nhỏ nhất: 738).
- Từ Visby: chọn Riga (chi phí thực tế nhỏ nhất: 201).
- Từ Riga: chọn Tallinn (chi phí thực tế nhỏ nhất: 305).
- Từ Tallinn: chọn Novgorod (đích đến, chi phí thực tế là 474).

Đường đi:

London → Amsterdam → Hamburg → Lubeck → Danzig → Visby → Riga → Tallinn → Novgorod

Tổng chi phí thực tế:

$$395 + 411 + 64 + 262 + 738 + 201 + 305 + 474 = 2850$$

Đánh giá: Đường đi là tối ưu vì thuật toán luôn chọn chi phí thực tế nhỏ nhất.

So sánh và đánh giá

Greedy Best-First Search

- **Ưu điểm:** Nhanh và có thể tìm được giải pháp hợp lý một cách nhanh chóng.
- **Nhược điểm:** Không đảm bảo tìm được đường đi tối ưu vì chỉ xét giá trị heuristic, bỏ qua chi phí thực tế.
- **Kết luận:** Đường đi không tối ưu do phụ thuộc vào giá trị heuristic.

Uniform Cost Search (UCS)

- **Ưu điểm:** Đảm bảo tìm được đường đi tối ưu vì xét chi phí thực tế nhỏ nhất từ điểm bắt đầu.
- **Nhược điểm:** Có thể chậm hơn Greedy do phải mở rộng nhiều node hơn.
- **Kết luận:** Đường đi tối ưu vì xét chi phí thực tế từ gốc.

Như vậy, Uniform Cost Search (UCS) là thuật toán cung cấp giải pháp tối ưu bằng cách xem xét chi phí thực tế, trong khi Greedy Best-First Search có thể không tìm được đường đi hiệu quả nhất do chỉ dựa vào giá trị heuristic.

2 Bài tập 2

2.1 Ý tưởng và phương pháp thiết kế thuật toán

Bài toán này yêu cầu kiểm tra sự tồn tại của chu trình âm trong đồ thị có trọng số, đồng thời in ra chu trình nếu có. Để giải quyết, ta sử dụng thuật toán Bellman-Ford với một số điều chỉnh:

- Bellman-Ford được sử dụng để tìm khoảng cách ngắn nhất từ một đỉnh nguồn đến các đỉnh khác trong đồ thị trọng số, kể cả khi có trọng số âm.
- Nếu sau $N - 1$ lần lặp, khoảng cách tiếp tục giảm, thì tồn tại chu trình âm.

2.1.1 Các bước chính:

1. **Khởi tạo mảng khoảng cách `dist` và mảng truy vết `parent`:**
   - Đặt tất cả các giá trị `dist[i]` ban đầu là $+\infty$, trừ đỉnh bắt đầu.
   - Mảng `parent` lưu đỉnh trước đó trên đường đi để dựng lại chu trình.
2. **Thực hiện $N-1$ lần cập nhật cạnh:**
   - Với mỗi cạnh (u, v, w), nếu `dist[u]` $+ w <$ `dist[v]`, cập nhật `dist[v]` và lưu truy vết `parent[v] = u`.
3. **Kiểm tra chu trình âm:**
   - Lặp qua tất cả các cạnh, nếu `dist[u]` $+ w <$ `dist[v]`, tức là tồn tại chu trình âm. Dùng mảng truy vết để tìm chu trình.

2.1.2 Tìm chu trình:

- Từ đỉnh bị ảnh hưởng bởi chu trình âm, sử dụng mảng `parent` để truy ngược lại các đỉnh trong chu trình.
- Đảm bảo đi đúng thứ tự bằng cách lặp thêm một chu trình từ đỉnh đó.

2.2 Mã giả

```
Input: N, M, edges (các cạnh đồ thị)
Output: YES + chu trình âm hoặc NO

1. Khởi tạo:
   dist[i] = + với mọi i  [1, N], parent[i] = -1
   Chọn đỉnh bắt đầu (giả sử là 1), dist[1] = 0

2. Thực hiện Bellman-Ford:
   For i từ 1 đến N-1:
       For mỗi cạnh (u, v, w) trong edges:
           If dist[u] + w < dist[v]:
               dist[v] = dist[u] + w
               parent[v] = u

3. Kiểm tra chu trình âm:
   For mỗi cạnh (u, v, w) trong edges:
       If dist[u] + w < dist[v]:
           Tồn tại chu trình âm:
           a. Đặt x = v
           b. Lặp N lần: x = parent[x] (để chắc chắn nằm trong chu trình)
           c. Khởi tạo chu trình: cycle = []
              Bắt đầu từ x, truy ngược lại qua parent để xây chu trình
           d. Đảo ngược chu trình để đúng thứ tự
           e. In "YES" và chu trình, kết thúc chương trình
```

4. Nếu không tìm thấy chu trình âm: In "NO"

2.3 Độ phức tạp

Thời gian:

- Thuật toán Bellman-Ford: $O(N \times M)$, với N là số đỉnh và M là số cạnh.
- Tìm chu trình: $O(N)$, khi cần lặp ngược qua chu trình.
- Tổng: $O(N \times M)$.

Không gian:

- Mảng `dist` và `parent`: $O(N)$.
- Tổng: $O(N + M)$.

2.4 Triển khai mã Python

```python
def detect_negative_cycle():
    # Input the number of vertices N and edges M
    N, M = map(int, input("Enter the number of vertices and
       edges (N M): ").split())

    # Input the list of edges
    edges = []
    print("Enter the edges in the format: u v w")
    for _ in range(M):
        u, v, w = map(int, input().split())
        edges.append((u, v, w))

    # Initialize distance array and parent array
    dist = [float('inf')] * (N + 1)
    parent = [-1] * (N + 1)
    dist[1] = 0

    # Bellman-Ford algorithm
    for _ in range(N - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                parent[v] = u

    # Check for negative weight cycle
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            # Negative weight cycle detected
            x = v
            for _ in range(N):
                x = parent[x]  # Ensure x is within the cycle

```

```
            # Retrieve the cycle
            cycle = []
            start = x
            while True:
                cycle.append(x)
                x = parent[x]
                if x == start and len(cycle) > 1:
                    break
            cycle.append(start)  # Add the starting vertex to
               complete the cycle
            cycle.reverse()

            print("YES")
            print(" ".join(map(str, cycle)))
            return

    print("NO")

# Execute the function
detect_negative_cycle()
```

2.5 Ví dụ chạy

Input:

```
4 5
1 2 1
2 4 1
3 1 1
4 1 -3
4 3 -2
```

Output:

```
YES
1 2 4 1
```

Input khác (không có chu trình âm):

```
3 3
1 2 3
2 3 4
3 1 5
```

Output:

```
NO
```