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

NGÀNH KHOA HỌC MÁY TÍNH

MÔN HỌC: PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

Phân tích thuật toán không đệ quy

Nhóm 7:
Hoàng Đức Dũng
Nguyễn Văn Hồng Thái

Mục lục

1 Bài 1

1.1 Phân tích và xác định độ phức tạp của thuật toán Huffman Coding

1.1.1 Khởi tạo

Đối với mỗi ký tự a trong tập ký tự α, chúng ta tạo một cây T_a chứa một nút duy nhất, được gán nhãn là a. Việc này mất $O(N)$ thời gian, với N là số lượng ký tự duy nhất.

1.1.2 Xây dựng cây Huffman

- **Khởi tạo danh sách**: Chúng ta khởi tạo một danh sách chứa tất cả các cây T_a. Việc này mất $O(N)$ thời gian.
- **Vòng lặp chính**: Trong vòng lặp chính, chúng ta thực hiện các bước sau cho đến khi chỉ còn một cây duy nhất trong danh sách:
  - Tìm cây có trọng số nhỏ nhất và cây có trọng số nhỏ thứ hai trong danh sách. Mỗi lần tìm kiếm mất $O(N)$ thời gian.
  - Hợp nhất hai cây này thành một cây mới T_3. Việc này mất $O(1)$ thời gian.
  - Thêm cây mới vào danh sách. Việc này mất $O(1)$ thời gian.
  - Xóa hai cây đã được hợp nhất khỏi danh sách. Việc này mất $O(N)$ thời gian.

Vì vòng lặp chính thực hiện $N-1$ lần, tổng thời gian cho vòng lặp chính là $O(N^2)$.

1.1.3 Tổng hợp độ phức tạp

- Khởi tạo: $O(N)$
- Xây dựng cây Huffman: $O(N^2)$

Do đó, độ phức tạp tổng thể của thuật toán Huffman Coding là $O(N^2)$.

1.2 Giải pháp tối ưu thuật toán Huffman Coding

Ta có thể sử dụng cấu trúc Min-Heap để việc tìm kiếm và thêm cây mới mất $O(\log N)$. Sau khi duyệt $N-1$ lần thì tổng độ phức tạp là $O(N \log N)$.

2 Bài 2

2.1 Phân tích thuật toán Prim

Hãy đưa ra mã giả chi tiết cho thuật toán trên và phân tích độ phức tạp của thuật toán

```
Prim(graph):
    MST = ∅  // Tập hợp các cạnh của cây khung nhỏ nhất
    key = [∞, ∞, ..., ∞]  // Mảng lưu trữ trọng số nhỏ nhất để kết nối các đỉnh
    parent = [None, None, ..., None]  // Mảng lưu trữ đỉnh cha của mỗi đỉnh trong MST
    key[0] = 0  // Bắt đầu từ đỉnh đầu tiên
    Q = {0, 1, 2, ..., n-1}  // Tập hợp các đỉnh chưa được xử lý

    while Q is not empty:
        u = extract_min(Q, key)  // Lấy đỉnh u có key nhỏ nhất
        MST = MST ∪ {u}  // Thêm đỉnh u vào MST

        for each v adjacent to u:
            if v in Q and weight(u, v) < key[v]:
                key[v] = weight(u, v)
                parent[v] = u

    return parent

```

Hình 1: Mã giá thuật toán Prim

- Sử dụng ma trận kề (Adjacency Matrix):
  - Trong trường hợp này, việc tìm đỉnh có giá trị key nhỏ nhất trong tập hợp Q mất $O(V^2)$ thời gian, với V là số đỉnh của đồ thị.
  - Do đó, độ phức tạp tổng thể của thuật toán là $O(V^2)$.
- Sử dụng danh sách kề (Adjacency List) và hàng đợi ưu tiên (Priority Queue):

- Việc tìm đỉnh có giá trị key nhỏ nhất trong tập hợp Q có thể được thực hiện trong $O(\log V)$ thời gian bằng cách sử dụng hàng đợi ưu tiên.
- Cập nhật các giá trị key của các đỉnh kề mất $O(\log V)$ thời gian cho mỗi cạnh.
- Do đó, độ phức tạp tổng thể của thuật toán là $O(E \log V)$, với E là số cạnh của đồ thị.

- Tóm lại, độ phức tạp của thuật toán Prim là $O(V^2)$ khi sử dụng ma trận kề và $O(E \log V)$ ($\approx$O$((m + n) \log n)$) khi sử dụng danh sách kề và hàng đợi ưu tiên.

2.2 Phân tích thuật toán Kruskal

Hãy đưa ra mã giả chi tiết cho thuật toán trên và phân tích độ phức tạp của thuật toán.

```
1.  Kruskal(graph):
2.      MST = Ø  // Tập hợp các cạnh của cây khung nhỏ nhất
3.      sort all edges in non-decreasing order of their weights
4.      for each vertex v in graph:
5.          make_set(v)  // Tạo tập hợp riêng cho mỗi đỉnh
6.      for each edge (u, v) in sorted edges:
7.          if find_set(u) != find_set(v):  // Kiểm tra nếu u và v thuộc các tập hợp khác nhau
8.              MST = MST U {(u, v)}  // Thêm cạnh (u, v) vào MST
9.              union(u, v)  // Hợp nhất hai tập hợp chứa u và v
10.     return MST
```

Hình 2: Mã giả thuật toán Kruskal

- Sắp xếp các cạnh: Sắp xếp các cạnh theo trọng số mất $O(E \log E)$ thời gian, với E là số cạnh của đồ thị.
- Thao tác trên tập hợp rời rạc (Disjoint Set Operations): Sử dụng cấu trúc dữ liệu Union-Find với kỹ thuật nén đường đi (path compression) và hợp nhất theo thứ hạng (union by rank), mỗi thao tác `find_set` và `union` mất $O(\log V)$ thời gian, với V là số đỉnh của đồ thị.

- Do đó, độ phức tạp tổng thể của thuật toán Kruskal là $O(E \log E + E \log V)$. Vì $E \log V$ thường nhỏ hơn hoặc bằng $E \log E$, độ phức tạp có thể được viết gọn là $O(E \log E)$ ($\approx$O($(m + n) \log n$)).