Lớp: CS112.P11.CTTN
Nhóm: 14
Sinh viên: Lê Nguyễn Anh Khoa. MSSV: 23520742
Sinh viên: Cáp Kim Hải Anh. MSSV: 23520036

BÀI TẬP
Phân tích độ phức tạp thuật toán không đệ quy

A. Bài 1: Huffman Coding:

Huffman coding là một thuật toán nén nổi tiếng và trong thực tế, thường được sử dụng rộng rãi trong các công cụ nén như Gzip, Winzip

1. Phân tích và xác định độ phức tạp của thuật toán trên:

Độ Phức Tạp của thuật toán tạo Huffman Tree có thể được phân tích như sau:

- Khởi tạo: Với mỗi ký tự trong bảng ký tự α, ta tạo một cây chỉ chứa một nút, và thêm vào tập hợp F. Thao tác này có độ phức tạp O(n), với n là số lượng ký tự.

- Vòng lặp chính:

+ Tìm hai cây có xác suất nhỏ nhất: Nếu sử dụng danh sách đơn giản để lưu trữ F, mỗi lần tìm hai cây có xác suất nhỏ nhất sẽ tốn O(n) do phải duyệt toàn bộ danh sách.

+ Ghép hai cây và thêm cây mới vào F: Thao tác này tốn O(1).

+ Lặp lại thao tác trên cho đến khi trong F chỉ còn một cây.

- Số lần lặp: Vòng lặp chạy n - 1 lần vì mỗi lần lặp giảm số lượng cây trong F đi một.

- Tổng độ phức tạp: $O(n^2)$ nếu sử dụng danh sách đơn giản để tìm cây có xác suất nhỏ nhất.

2. Một giải pháp để tối ưu thuật toán vừa được nêu trên:

- Để tối ưu thuật toán, ta có thể sử dụng một hàng đợi ưu tiên (priority queue), thường được triển khai bằng heap nhị phân:

+ Khởi tạo hàng đợi ưu tiên: Thêm tất cả cây ban đầu vào hàng đợi ưu tiên. Mỗi thao tác thêm có độ phức tạp O(log n), tổng cộng O(n.log n).

+ Vòng lặp chính: Thao tác lấy ra hai cây có xác suất nhỏ nhất và thêm cây mới trở lại vào hàng đợi đều có độ phức tạp O(log n). Vòng lặp chạy n - 1 lần.

- Tổng độ phức tạp: O(n.log n) khi sử dụng hàng đợi ưu tiên.

Bằng cách sử dụng hàng đợi ưu tiên, chúng ta có thể giảm độ phức tạp của thuật toán từ $O(n^2)$ xuống O(n log n), làm cho thuật toán hiệu quả hơn đặc biệt khi n lớn.

***Mã giải khi sử dụng hàng đợi ưu tiên:**

// Khởi tạo hàng đợi ưu tiên

priorityQueue = new PriorityQueue()

// Khởi tạo

For each a in α do:

T_a = tree containing only one node, labeled "a"

$P(T_a) = p_a$ // Insert T_a into priorityQueue with priority $P(T_a)$

// Vòng lặp chính

While priorityQueue.size() > 1 do:

T_1 = priorityQueue.removeMin() // Lấy cây có xác suất nhỏ nhất

T_2 = priorityQueue.removeMin() // Lấy cây có xác suất nhỏ thứ hai

T_3 = merger of T_1 and T_2

// Gốc của T_1 và T_2 là con trái, phải của T_3

// $P(T_3) = P(T_1) + P(T_2)$

// Insert T_3 into priorityQueue with priority $P(T_3)$

// Kết quả

Return priorityQueue.removeMin() // Cây Huffman cuối cùng

B. Bài 2: Thuật toán Minimum Spanning Tree

Câu 1:

1. Mã giải chi tiết cho thuật toán và phân tích độ phức tạp của thuật toán:

*** Mã giải chi tiết cho thuật toán Prim (G = (V, E)):**

Đầu vào: Một đồ thị vô hướng liên thông G với tập đỉnh V và tập cạnh E.

Đầu ra: Cây khung nhỏ nhất (MST).

1. Khởi tạo:

- X = {s}, trong đó s là một đỉnh bất kỳ.
- T = {} (tập rỗng dành cho các cạnh của cây khung nhỏ nhất.
- Hàng đợi ưu tiên Q để lưu các cạnh với trọng số nhỏ nhất.

2. Với mỗi cạnh (s, v) mà v kề với s:

- Thêm cạnh (s, v) vào Q với chi phí c(s, v).

3. Trong khi X ≠ V:

a. Lấy ra cạnh (u, v) có chi phí nhỏ nhất từ Q.

b. Nếu v ∉ X:

i. Thêm v vào X.

ii. Thêm cạnh (u, v) vào T.

iii. Với mỗi cạnh (v, w) mà w ∉ X: Thêm (v, w) vào Q với chi phí c(v, w).

4. Kết thúc: Trả về T (tập các cạnh của cây khung nhỏ nhất).

***Phân tích độ phức tạp của thuật toán:**

- Khởi tạo: Việc thêm tất cả các cạnh từ đỉnh ban đầu vào hàng đợi ưu tiên mất O(log m), với m là số cạnh.

- Vòng lặp chính:

+ Mỗi lần thêm một cạnh vào X, mất O(log m) để lấy cạnh nhỏ nhất từ hàng đợi.

+ Việc thêm các cạnh mới vào hàng đợi cũng mất O(log m)

Tổng số lần lấy ra và thêm cạnh vào hàng đợi bị giới hạn bởi m, vì vậy độ phức tạp là O(m.log m)

Vì vậy, tổng độ phức tạp cho cách tiếp cận này là **O(m.log m)**

2. Thuật toán này chưa cho độ phức tạp là O((n+m)log(n)) với n là số đỉnh, m là số cạnh. Đề xuất một phương pháp khác cho độ phức tạp như trên:

Sử dụng **hàng đợi ưu tiên** được cài đặt bằng **Min-Heap**

***Cấu trúc chính của Min-Heap:**

Min-Heap là một cây nhị phân mà trong đó giá trị tại mỗi nút cha luôn nhỏ hơn hoặc bằng giá trị tại các nút con của nó. Điều này giúp việc lấy ra phần tử nhỏ nhất (ở đỉnh heap) rất hiệu quả với độ phức tạp O(log n)

***Thuật toán Prim sử dụng Min-Heap:**

Đầu vào: Đồ thị liên thông G = (V, E)

Đầu ra: Cây khung nhỏ nhất (MST)

1. Khởi tạo:

- Chọn một đỉnh ban đầu s, thêm nó vào tập đỉnh đã thăm **X** = {s}

- Tạo một min-heap Q rỗng.

- Đẩy tất cả các cạnh nối từ s đến các đỉnh khác vào **min-heap**, với trọng số của cạnh là giá trị ưu tiên.

2. Vòng lặp chính:

Trong khi có đỉnh chưa được thăm X ≠ V:

- Lấy cạnh có trọng số nhỏ nhất (u, v) từ **min-heap** Q

- Nếu đỉnh v tương ứng với cạnh đó chưa được thăm:

+ Thêm đỉnh v này vào **X**.

+ Thêm cạnh (u, v) này vào cây khung nhỏ nhất **T**.

+ Đẩy tất cả các cạnh của đỉnh v vừa thêm mà kết nối đến các đỉnh chưa được thăm vào **min-heap** Q

3. Kết thúc: Khi tất cả các đỉnh đã được thăm, trả về cây khung nhỏ nhất T.

***Độ phức tạp khi sử dụng Min-Heap:**

- Với min-heap, tổng thời gian cần để thêm các cạnh vào hàng đợi và lấy ra cạnh có trọng số nhỏ nhất sẽ là O(log n) cho mỗi thao tác.

- Có tổng cộng n đỉnh và m cạnh, nên số lần thao tác với heap là m lần. Độ phức tạp tổng cộng là:

+ Khi thêm **cạnh vào heap** và **lấy cạnh ra từ heap**: O(m.log n), vì heap sẽ có kích thước không vượt quá số đỉnh n, do đó các thao tác với heap tốn O(log n)

+ Việc duyệt qua mỗi cạnh và đỉnh không chiếm thêm thời gian ngoài các thao tác liên quan đến heap.

Vì vậy, độ phức tạp của thuật toán Prim khi sử dụng **min-heap** là O((n+m)log n), trong đó n là số đỉnh và m là số cạnh.

Câu 2:

1. Mã giải chi tiết cho thuật toán và phân tích độ phức tạp của thuật toán:

*** Mã giải chi tiết cho thuật toán Kruskal (G = (V, E))**

Đầu vào: Đồ thị liên thông G = (V, E)

Đầu ra: Cây khung nhỏ nhất (MST)

1. Khởi tạo:

- T = {} (tập rỗng cho MST)

- Sắp xếp các cạnh trong E theo chi phí (thí dụ, sử dụng MergeSort)

2. Vòng lặp chính:

Đối với mỗi cạnh e ∈ E, theo thứ tự không giảm của chi phí:

a. Nếu T ∪ {e} không tạo thành chu trình: Thêm e vào T.

3. Kết thúc: Trả về T (các cạnh của MST).

***Phân tích độ phức tạp của thuật toán:**

- **Sắp xếp các cạnh**: Việc sắp xếp mmm cạnh sẽ tốn O(m.log m) thời gian (sử dụng MergeSort hoặc QuickSort).

- **Kiểm tra chu trình**:

+ Để kiểm tra xem T ∪ {e} có tạo thành chu trình hay không, ta thường sử dụng cấu trúc dữ liệu Union-Find (hay Disjoint Set Union - DSU).

+ Phép hợp và kiểm tra chu trình của Union-Find có độ phức tạp gần tuyến tính, khoảng O(α(n)) cho mỗi phép toán, với α(n) là hàm rất chậm.

- **Vòng lặp chính**: Trong vòng lặp chính, ta sẽ thực hiện phép kiểm tra chu trình cho mỗi cạnh, tổng cộng là m lần.

Vì vậy, tổng độ phức tạp của thuật toán Kruskal sẽ là: O(m.log m) + O(α(n)) = O(m.log m)

2. Thuật toán này chưa cho độ phức tạp là O((n+m)log(n)) với n là số đỉnh, m là số cạnh. Đề xuất một phương pháp khác cho độ phức tạp như trên:

Thuật toán Kruskal thường không phải là phương pháp tối ưu để đạt được độ phức tạp như trên do quá trình sắp xếp các cạnh. Để đạt được độ phức tạp này, có thể sử dụng **thuật toán Kruskal** kết hợp với phương pháp **sắp xếp nhanh** và cấu trúc dữ liệu Union-Find với tối ưu hóa.

***Cách thực hiện:**

1. Khởi tạo:

Tạo một cấu trúc Union-Find để quản lý các tập hợp đỉnh, bao gồm hai phép toán: Find và Union với tối ưu hóa Path Compression và Union by Rank.

2. Sắp xếp các cạnh: Nhận danh sách các cạnh và sắp xếp chúng theo chi phí tăng dần.

3. Duyệt qua các cạnh:

- Khởi tạo danh sách rỗng cho MST và tổng chi phí bằng 0.
- Với mỗi cạnh (u,v,cost):
  + Nếu Find(u) khác Find(v) (tức là không thuộc cùng tập):
    i. Gọi Union(u, v) để hợp nhất các tập.
    ii. Thêm cạnh (u,v) vào MST.
    iii. Cộng chi phí vào tổng chi phí.

4. Kết thúc:

- Dừng lại khi MST chứa đủ n − 1 cạnh (n là số đỉnh).
- Trả về MST và tổng chi phí.

***Độ phức tạp:**

- **Sắp xếp các cạnh**: O(m.log m), trong đó m là số cạnh của đồ thị.

- **Thao tác Union-Find**: Với việc tối ưu Path Compression và Union by Rank, mỗi phép toán find và union có độ phức tạp gần như O(1). Tổng cộng cho m phép toán là O(mα(n)), trong đó α(n) là hàm đảo Ackermann (rất nhỏ, gần như hằng số).