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

BÀI TẬP MÔN HỌC
PHÂN TÍCH THIẾT KẾ THUẬT TOÁN

Sinh viên: Đỗ Phương Duy - 23520362

Sinh viên: Nguyễn Nguyên Khang - 22520623

Ngày 6 tháng 10 năm 2024

Mục lục

1 Bài 1: Huffman Coding:

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

- **Vòng lặp init:**
  - Gọi α là danh sách các cây ban đầu với mỗi cây chỉ có một node gốc.
  - Vòng lặp này lần lượt lặp qua từng cây này (lặp tổng cộng n lần, với n là tổng số cây trong α)

$$\Rightarrow \text{Độ phức tạp của vòng lặp này là } O(n)$$

- **Vòng lặp chính:**
  - Gọi n là số lượng cây ban đầu, chúng ta cần tổng cộng n-1 vòng lặp để hợp nhất tất cả thành một cây.

$$\Rightarrow \text{Độ phức tạp của vòng lặp này là } O(n)$$

  - Tại mỗi vòng lặp, ta cần tìm cây có tần suất nhỏ nhất, chi phí là $O(m)$ cho mỗi vòng lặp với m là số lượng cây tại vòng lặp hiện tại.
  - Tại mỗi vòng lặp, ta cũng cần tìm cây có tần suất nhỏ thứ hai cũng với chi phí $O(m)$ cho mỗi vòng lặp.
  - Ta có thể nhận thấy rằng, ở lượt lặp đầu tiên ta có m=n, giá trị của m giảm đi sau mỗi vòng lặp. Độ phức tạp của các vòng lặp tìm cây có tần suất nhỏ nhất và nhỏ thứ hai là:

$$O(m) + O(m) = O(2m) = O(m) = O(n)$$

$$\Rightarrow \text{Độ phức tạp của vòng lặp này là } O(n \cdot n) = O(n^2)$$

$$\Rightarrow \text{Độ phức tạp tổng thể của thuật toán là là } O(n + n^2) = O(n^2)$$

1.2 Tối ưu thuật toán

Một trong những phương án để tối ưu thuật toán này là sử dụng cấu trúc dữ liệu min-heap, min-heap cho phép chúng ta tìm và xóa các phần tử có giá trị nhỏ nhất một cách hiệu quả hơn.

- **Vòng lặp thứ nhất:**
  - Chúng ta ánh xạ danh sách các node thành 1 cây nhị phân (node thứ i sẽ nhận node con trái ở vị trí 2i và node con phải ở vị trí 2i+1)
  - Chúng ta tiến hành heapify cây nhị phân thu được, bắt đầu từ node không lá đầu tiên ở vị trí thứ n/2 (n là tổng số node).
  - Khi tiến hành heapify tại một node, chúng ta so sánh giá trị tại node này với các node con của chúng. Chi phí để heapify cho mỗi node là $O(H)$ với H là chiều cao của cây.
  - Đối với cây nhị phân hoàn chỉnh chiều cao H của một cây có n node có thể được tính bằng cách:

$$H = \lfloor \log_2(n) \rfloor$$

- Do đó, độ phức tạp cho việc heapify tại mỗi node là:

$$O(log(n))$$

- Ta có số lượng node cần heapify sẽ giảm dần khi đi từ dưới lên, ta có thể tính tổng chi phí như sau:

$$T(n) = 1 \cdot O(h) + 2 \cdot O(h-1) + 4 \cdot O(h-2) + 8 \cdot O(h-3) + \ldots + n \cdot O(0)$$

- Hay:

$$T(n) = \sum_{i=0}^{h} 2^i \cdot O(h-i)$$

- Khi thay O bằng một hằng số C:

$$T(n) = \sum_{k=0}^{h} C \cdot 2^k \cdot (h-k)$$

- Ta có thể tách tổng này thành 2 phần:

$$T(n) = C \cdot h \cdot \sum_{k=0}^{h} 2^k - C \cdot \sum_{k=0}^{h} k \cdot 2^k$$

- Tính tổng phần đầu tiên: Tổng $\sum_{k=0}^{h} 2^k$ là tổng cấp số nhân:

$$\sum_{k=0}^{h} 2^k = 2^{h+1} - 1$$

- Tính tổng phần thứ hai:

$$S = \sum_{k=0}^{h} k \cdot 2^k$$

- Nhân cả hai bên với 2:

$$2S = \sum_{k=0}^{h} k \cdot 2^{k+1} = \sum_{k=1}^{h+1} (k-1) \cdot 2^k$$

$$2S = \sum_{k=1}^{h+1} (k-1) \cdot 2^k = \sum_{k=1}^{h+1} k \cdot 2^k - \sum_{k=1}^{h+1} 2^k$$

- Ta có thể viết lại tổng đầu tiên như sau:

$$\sum_{k=1}^{h+1} k \cdot 2^k = S + (h+1) \cdot 2^{h+1}$$

- Tính vế sau:

$$\sum_{k=1}^{h+1} 2^k = 2 \cdot (2^{h+1} - 1) = 2^{h+2} - 2$$

- Thay vào ta được:

$$2S = S + (h+1)\cdot 2^{h+1} - (2^{h+2} - 2)$$
$$2S = S + (h+1)\cdot 2^{h+1} - 2^{h+2} + 2$$

- Tính $2S$-S:

$$2S - S = S = (h+1)\cdot 2^{h+1} - 2^{h+2} + 2$$

- Tiến hành rút gọn:

$$S = 2^{h+1}(h+1-2) + 2$$
$$S = 2^{h+1}(h-1) + 2$$

- Quay lại tổng ban đầu:

$$T(n) = C\cdot h\cdot (2^{h+1} - 1) - C\cdot \Big((h-1)\cdot 2^{h+1} + 2\Big)$$

- Mà ta có $h = O(\log n)$ nên :

$$2^{h+1} = O(n)$$

- Do đó:

$$T(n) = O(h\cdot n) - O((h-1)\cdot n) = O(n)$$

- **Vòng lặp thứ hai:**
  - Chúng ta tiến hành lặp $n-1$ lần với n là số cây ban đầu: (mỗi cây chỉ có node gốc)

    $$\Rightarrow \text{Chi phí là } O(n)$$

  - Chúng ta lặp đến khi chỉ còn 1 cây trong min-heap:
    * Lấy ra 2 cây có tần suất nhỏ nhất là T_1 và T_2 (chi phí là $O(\log n) + O(\log n) = O(\log n)$)
    * Tạo cây T_3 từ T_1 và T_2
    * Chèn T_3 vào min-heap (chi phí là $O(\log n)$)

    $$\Rightarrow \text{Chi phí tổng là } O(n\log n)$$

  $\Rightarrow$ Tổng chi phí của cả 2 vòng lặp là là $O(n) + O(n\log n) = O(n\log n)$

2 Bài 2: Thuật toán Minimum Spanning Tree

2.1 Prim

- **Pseudocode**

```
function PRIM(Graph)
    Initialize an empty set MST
    Initialize a priority queue PQ
    Initialize a set of visited nodes, Visited = {}
    StartNode ← 0
    Visited.add(StartNode)
     for each edge (StartNode, v, weight) in Graph[StartNode] do
     end
     PQ.enqueue(v, weight)

     while PQ is not empty do
     end
     (u, weight) ← PQ.dequeue()
     if u ∉ Visited then
     end
     MST.add((StartNode, u, weight))
Visited.add(u)
     for each edge (u, v, weight) in Graph[u] do
     end
     v ∉ Visited
PQ.enqueue(v, weight)


StartNode ← u


return MST
end function
```

Algorithm 1: Prim's Algorithm

- **Phân tích độ phức tạp:**
  Vì mỗi lần ta thêm 1 đỉnh vào tập X, nên vòng lặp while sẽ lặp |V| lần. Ở bước tìm cạnh ta dùng ý tưởng của priority-queue ở bài 1, khi đó chúng ta chỉ cần tìm cạnh đó trong độ phức tạp O(log|E|). Vậy độ phức tạp của thuật toán sẽ là O(|V|log(|E|)).

2.2 Kruskal:

- **Pseudocode:**

```
function Kruskal(Graph)
    Initialize an empty set MST
    Initialize a disjoint set structure
    Sort edges in Graph by weight in ascending order
     for each edge (u, v, weight) in sorted edges do
     end
     Find(u) ≠ Find(v)
    MST.add((u, v, weight))
    Union(u, v)


    return MST
end function
```

Algorithm 2: Kruskal's Algorithm

- **Phân tích độ phức tạp:**
  Sắp xếp các cạnh theo trọng số lớn nhất mất O(|E|log(|E|)). Mỗi lần kiểm tra và hợp nhất hai tập hợp (Find and Union) mất O(ß(V))) với ß hàm đảo ngược với hàm Ackerman với hằng số rất nhỏ, nhỏ hơn cả log(|E|)
  Vậy tổng các vòng lặp là O(|E|ß(V))
  Tổng cộng độ phức tạp O(|E|log(|E|))

Tài liệu