ĐẠI HỌC quốc gia TP.HCM
Trường đại học CÔNG NGHỆ THÔNG TIN

NgÀNH ...

MÔN HỌC: ...

Tiêu đề Bài tập CS112 - Nhóm 6

Nhóm 8:
Tên SV1: Hoàng Minh Thái – MSSV: 23521414, Tên SV2: Nguyễn Trọng Tất Thành –MSSV: 23521455,

Giảng viên:
Tên Giảng Viên: Nguyên Thanh Sơn

1 Mục thfí nhất

Thuật toán Huffman Coding

1, Phân tích và xác định độ phfíc tạp của thuật toán

Phân tích thuật toán:

Bước khởi tạo (init):

Tạo một cây chứa chỉ một nút cho mỗi ký tự a trong bảng chữ cái α .Mỗi cây này được gán một xác suất $P_{T_\alpha} = T_\alpha$ là xác suất của kí tự α.

Tất cả các cây này được thêm vào một hàng đợi ưu tiên F, trong đó mỗi cây được sắp xếp dựa trên xác suất xuất hiện của nó.

Vòng lặp chính:

Trong khi số lượng cây $|F| \geq 2$, *thchin* :

- Lấy ra hai cây T_1 và T_2 có xác suất nhỏ nhất từ F.

- Tạo một cây mới T_3 bằng cách hợp nhất hai cây này. Gốc của cây mới sẽ có xác suất bằng tổng xác suất của hai cây con:

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

Cây con trái là T_1, cây con phải là T_2.

- Thêm cây T_3 trở lại hàng đợi ưu tiên F.

Khi hàng đợi chỉ còn một cây, đó là cây Huffman cuối cùng.

2. Phân tích độ phfíc tạp thời gian:

Khởi Tạo

Tạo một cây cho mỗi ký tự và thêm nó vào hàng đợi ưu tiên. Với n ký tự, mỗi thao tác thêm vào hàng đợi ưu tiên tốn $O(\log n)$, do đó bước này có độ phức tạp là $O(n \log n)$.

Vòng Lặp Chính

Mỗi lần lặp, chúng ta lấy ra hai cây có xác suất nhỏ nhất từ hàng đợi ưu tiên, thao tác này mất $O(\log n)$ thời gian mỗi lần. Sau đó, chúng ta hợp nhất chúng và thêm cây mới vào lại hàng đợi, cũng mất $O(\log n)$.

Vòng lặp này chạy $n-1$ lần, vì sau mỗi lần hợp nhất số lượng cây giảm đi 1, từ n cây xuống còn 1 cây cuối cùng. Do đó, tổng thời gian của vòng lặp là $O(n \log n)$.

Tổng Quát

Kết hợp lại, độ phức tạp thời gian tổng thể của thuật toán là $O(n \log n)$.

2. Giải Pháp Tối Ưu Thuật Toán Huffman Coding

Tối ưu hóa bằng việc sử dụng mảng thay vì cấu trúc dữ liệu như hàng đợi ưu tiên (priority queue).

Giải Thích

Thay vì sử dụng một hàng đợi ưu tiên để tìm hai cây có xác suất nhỏ nhất, chúng ta có thể sử dụng một mảng đã được sắp xếp để lưu trữ các cây này. Bằng cách sử dụng mảng sắp xếp, quá trình trích xuất hai phần tử nhỏ nhất sẽ trở nên dễ dàng và việc chèn cây mới vào mảng cũng được tối ưu hóa.

Các Bước Thực Hiện

1. **Bước Chuẩn Bị:**
    - Đầu tiên, sắp xếp danh sách các ký tự theo xác suất xuất hiện theo thứ tự tăng dần.
    - Lưu trữ các ký tự và xác suất của chúng trong một mảng sắp xếp. Điều này thay thế hàng đợi ưu tiên thông thường.
2. **Xây Dựng Cây Huffman:**

- Trong quá trình xây dựng cây, thay vì sử dụng hàng đợi ưu tiên, bạn sẽ lấy hai phần tử đầu tiên từ mảng sắp xếp (phần tử nhỏ nhất và phần tử thứ hai nhỏ nhất).
- Sau khi hợp nhất hai cây con nhỏ nhất, cây mới sẽ được thêm vào vị trí thích hợp trong mảng, đảm bảo mảng luôn được sắp xếp.
- Quá trình này tiếp tục cho đến khi còn lại chỉ một cây trong mảng, đó chính là cây Huffman.

Ưu Điểm

- **Tối ưu không gian bộ nhớ:** Việc sử dụng mảng sắp xếp sẽ giúp loại bỏ sự phức tạp của việc quản lý hàng đợi ưu tiên. Mảng sắp xếp sử dụng ít không gian hơn, vì chúng ta không cần phải duy trì các con trỏ hoặc cấu trúc dữ liệu bổ sung như trong hàng đợi ưu tiên hoặc heap nhị phân.
- **Thực thi nhanh cho tập dữ liệu nhỏ hoặc trung bình:** Đối với các tập dữ liệu có kích thước nhỏ hoặc trung bình (ví dụ như khi số lượng ký tự là vừa phải), việc sử dụng mảng sắp xếp có thể mang lại hiệu suất tốt hơn so với heap nhị phân hoặc Fibonacci heap. Với mảng, việc tìm hai phần tử nhỏ nhất luôn chỉ cần duyệt qua đầu mảng, không cần thao tác với các cấu trúc dữ liệu phức tạp.

Độ Phfíc Tạp

- **Sắp xếp ban đầu:** Bước sắp xếp ban đầu mất

$$O(n \log n)$$

thời gian.

- **Trích xuất hai phần tử nhỏ nhất:** Việc tìm hai phần tử nhỏ nhất luôn có thể thực hiện trong

$$O(1)$$

vì chúng nằm ở đầu mảng.

- **Chèn cây mới vào mảng:** Chèn cây mới vào mảng sắp xếp yêu cầu duyệt qua mảng và chèn vào vị trí phù hợp. Điều này mất

$$O(n)$$

cho mỗi lần chèn.

Tuy nhiên, khi kết hợp lại, tổng độ phức tạp thời gian vẫn là

$$O(n \log n)$$

cho toàn bộ thuật toán, tương đương với việc sử dụng hàng đợi ưu tiên. Nhưng việc sử dụng mảng sắp xếp có thể cải thiện hiệu suất thực tế trong nhiều trường hợp, nhờ tiết kiệm bộ nhớ và thao tác đơn giản hơn.

2 Mục thfí hai

Bài toán cây khung nhỏ nhất:

Thuật toán Prim:

Mã giả:

Dữ liệu:

n: số đỉnh của đồ thị

adj: danh sách kề, lưu các cạnh và trọng số giữa các đỉnh

dist: mảng lưu khoảng cách nhỏ nhất từ đỉnh bất kỳ đến cây bao trùm

parent: mảng lưu đỉnh cha của mỗi đỉnh trong cây bao trùm

visited: mảng lưu trạng thái đã thăm của mỗi đỉnh

Hàm khởi tạo PRIM(*n*):

Khởi tạo đồ thị có *n* đỉnh

Khởi tạo mảng *dist* với giá trị vô hạn, *parent* với giá trị -1, và *visited* với giá trị false

Hàm ADDEdgE(*u, v, w*):

Thêm cạnh nối giữa hai đỉnh *u* và *v* với trọng số *w* vào danh sách kề *adj*

Hàm fINDMST(*start*):

Đặt khoảng cách từ đỉnh bắt đầu *start* đến cây bao trùm bằng 0

Khởi tạo hàng đợi ưu tiên *pq*, thêm đỉnh *start* vào *pq*

mst $\leftarrow 0$ Biến lưu tổng trọng số của cây bao trùm nhỏ nhất

While hàng đợi *pq* không rỗng:

Lấy đỉnh *u* với khoảng cách nhỏ nhất từ *pq*

Đánh dấu *u* là đã thăm

Cộng khoảng cách *dist*[*u*] vào *mst*

For mỗi cạnh (*v, w*) kề với *u*:

If *v* chưa được thăm và trọng số *w* nhỏ hơn *dist*[*v*]:

Cập nhật *dist*[*v*] $\leftarrow$ *w*

Cập nhật *parent*[*v*] $\leftarrow$ *u*

Thêm *v* vào hàng đợi ưu tiên *pq*

End While

return *mst* Trả về tổng trọng số của cây bao trùm nhỏ nhất

Giải thích độ phfíc tạp:

Thuật toán trên có độ phức tạp là: $O((n + m) \log n)$.

Bới độ phức tạp của nó được tính bằng cách nó duyệt qua n đinh mỗi đỉnh 1 lần và m cạnh mỗi cạnh 1 lần, cộng với độ phức tạp của hàng đợi ưu tiên nên nó đạt được độ phức tạp như trên.

Thuật toán Kruskal:

Mã giả:

Dữ liệu:

n: số đỉnh của đồ thị

parent: mảng đại diện cha của mỗi đỉnh trong cây bao trùm

rank: mảng lưu thứ bậc của mỗi đỉnh

edges: danh sách các cạnh trong đồ thị

Hàm khởi tạo KRUSKAL(*n*):

Khởi tạo đồ thị có *n* đỉnh

For mỗi đỉnh *i* = 1 đến *n*:

parent[*i*] ← *i*

rank[*i*] ← 0

Hàm ADDEDGE(*u, v, w*):

Thêm cạnh nối giữa hai đỉnh *u* và *v* với trọng số *w* vào danh sách cạnh *edges*

Hàm FINDSET(*u*):

If *parent*[*u*] = *u*:

return *u*

Else:

return *parent*[*u*] ← FINDSET(*parent*[*u*]) **Hàm** UNIONSET(*u, v*):

u ← FINDSET(*u*)

v ← FINDSET(*v*)

If *u* = *v*:

return False

If *rank*[*u*] < *rank*[*v*]:

```
        Đổi chỗ u và v
    parent[v] ← u
    If rank[u] = rank[v]:
        rank[u] ← rank[u] + 1
    return True
Hàm mst():
    Sắp xếp các cạnh trong edges theo trọng số tăng dần
    res ← 0
    For mỗi cạnh (w, (u, v)) trong edges:
        If unionSet(u, v):
            res ← res + w
    return res
```

Giải thích độ phíc tạp:

Thuật toán trên có độ phức tạp là: $O(n \log n + m \log m)$.

Bới độ phức tạp của nó được quyết định bởi 2 phần chính, đó là: Độ phức tạp của thuật toán sort các cạnh tối ưu là $O(m \log m)$

Độ phức tạp của disjoint set để kiểm tra 2 đỉnh hiện tại hợp vào có cùng thành phần liên thông hay không tốn $O(n \log n)$.