CS112.P11.KHTN - Bài tập chủ đề 1

Nhóm 3
Nguyễn Hữu Đặng Nguyên - 23521045
Trần Vạn Tấn - 23521407

Ngày 7 tháng 10 năm 2024

Mục lục

1 Bài tập 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.

Dưới đây là một đoạn mã giả về cách tạo Huffman Tree:

```
//init
For each a in α do:
        T_a = tree containing only one node, labeled "a"
        P(T_a) = p_a
        F = {T_a} //invariant: for all T in F, P(T) = ∑_{a∈T} p_α
//main loop
While length(F) >= 2 do
        T_1 = tree with smallest P(T)
        T_2 = tree with second smallest P(T)
        remove T1 and T2 from F
        T_3 = merger of T1 and T2
        // root of T1 and T2 is left, right children of T3
        P(T_3) = P(T_1) + P(T_2)
        add T_3 to F
Return F[0]
```

1.1 Hãy phân tích và xác định độ phức tạp của thuật toán.

Giả sử độ dài của *alpha* là n. Số lượng ký tự khác nhau là k.

Bước khởi tạo:

-Với mỗi giá trị khaác nhau a trong *alpha*, ta sẽ tạo một cây chỉ chứa a. Các cây này sẽ được lưu trong một mảng F.

- Đồng thời ta cũng tạo một mảng P đánh dấu tần suất xuất hiện của mỗi ký tự a bằng cách duyệt qua cả mảng *alpha*.

-Bước này có độ phức tạp là $O(n + k)$.

Bước thực thi:

-Mỗi bước ta chọn 2 ký tự có số lần xuất hiện ít nhất trong a (gọi là x, y) . Sau đó ta thực hiện xoá 2 cây x, y và ghép 2 cây này lại thành một cây mới rồi đưa cây mới này vào lại trong F.

-Vì sử dụng F như một mảng, ta chỉ có thể chọn x,y bằng cách duyệt qua tất cả thành phần bên trong F. Mỗi bước này có độ phức tạp là $O(k)$.

-Vì mỗi bước, số lượng cây trong F giảm đi 1 nên ta thực hiện k-1 bước, tổng độ phức tạp xấp xỉ $O(k^2)$

Tổng độ phức tạp: $O(n + k^2)$

1.2 Một giải pháp để tối ưu thuật toán trên.

Nhận xét: Nhận thấy bước thực thi sử dụng phương án duyệt qua cả mảng khá mất thời gian, ta sử dụng cấu trúc dữ liệu min-heap để tối ưu.

Tối ưu bằng min-heap:

- Nếu sử dụng min-heap để lưu các cây, khi đó bước khởi tạo ta phải đưa từng cây vào min-heap khiến độ phức tạp là $O(n + k * log_2 k)$.
- Tại bước thực thi, việc lấy ra phần từ nhỏ nhất của min-heap có độ phức tạp $O(log_2 k)$. Vì thực hiện k-1 bước nên độ phức tạp lúc này là $O(k * log_2 k)$.
- Tổng độ phức tạp lúc này là $O(n + k * log_2 k)$

2 Bài tập 2: Thuật toán Minimum Spanning Tree

2.1 Thuật toán Prim:

2.1.1 Mã giả chi tiết và phân tích độ phức tạp của thuật toán.

Mã giả:

```
// Input:
// G = (V, E) - Đồ thị với tập đỉnh V và tập cạnh E
// w(u, v) - Trọng số của cạnh nối giữa đỉnh u và v
// Output:
// MST - Cây khung nhỏ nhất (dưới dạng danh sách các cạnh)

// Pseudocode Prim:

function Prim(G, w)
    Select an arbitrary vertex s from V
    Initialize X as an empty set     // X là tập các đỉnh trong cây khung
    Initialize T as an empty set     // T là tập các cạnh trong cây khung (MST)
    Initialize a priority queue PQ   // PQ quản lý các cạnh dựa trên trọng số

    // Bước 1: Thêm tất cả các cạnh xuất phát từ đỉnh s vào hàng đợi
    for each vertex v adjacent to s do
        Insert edge (s, v) into PQ with weight w(s, v)

    // Bước 2: Bắt đầu tìm cây khung nhỏ nhất
```

```
    while |X| < |V| do
        (u, v) := PQ.extractMin()    // Lấy cạnh có trọng số nhỏ nhất (u, v)
        if v not in X then           // Nếu đỉnh v chưa có trong cây khung
             Add vertex v to X       // Thêm đỉnh v vào cây khung
             Add edge (u, v) to T    // Thêm cạnh (u, v) vào tập T

            // Bước 3: Thêm các cạnh nối từ đỉnh v vào hàng đợi PQ
            for each vertex w adjacent to v do
                if w not in X then   // Nếu đỉnh w chưa trong cây khung
                    Insert edge (v, w) into PQ with weight w(v, w)

    return T                         // Trả về tập T là cây khung nhỏ nhất (MST)

```

Phân tích độ phức tạp:

- **Input**
  - Dòng đầu tiên chứa số nguyên n, số đỉnh.
  - Dòng thứ hai chứa số nguyên m, số cạnh.
- **Cài đặt**
  - **Khởi tạo:**
    - Tập X quản lý các đỉnh nằm bên trong cây khung.
    - Tập Y quản lý các đỉnh nằm bên ngoài cây khung
    - Tập T quản lý các cạnh theo trọng số.
    - Chọn s là một đỉnh bất kì, sau đó thêm tất cả các cạnh (s, v) cùng trọng số vào T.
  - **Chương trình chính:**
    - Ta lấy ra cạnh có trọng số nhỏ nhất trong tập T, gọi nó là (u, v). Trong đó: u thuộc tập X(cây khung), v thuộc tập Y(ngoài cây khung). Việc này có thể được thực hiện bằng nhiều cách tiếp cận khác nhau. Ở đây ta sử dụng danh sách liên kết, hoặc ma trận trọng số.

* Tìm cạnh có trọng số nhỏ nhất bằng cách duyệt toàn bộ đỉnh u thuộc tập X, sau đó duyệt các đỉnh v thuộc tập Y. Với mỗi lần thêm cạnh, đỉnh, việc tìm cạnh có trọng số nhỏ nhất có thể mất độ phức tạp O(m), và tổng thì có tổng cộng n – 1 cạnh cần thêm.
* **Độ phức tạp tổng quát:** O(n*m).

2.1.2 Phương pháp mà bạn đã đề ra có cho độ phức tạp là O((n+m)log(n)) (với n là số đỉnh còn m là số cạnh) không? Nếu không thì đề xuất một phương pháp khác cho độ phức tạp như trên.

Phương pháp mới dùng Min-Heap:
Sử dụng Min-Heap cho tập T(Tập quản lý các cạnh theo trọng số) để lưu trữ các cạnh với trọng số của chúng, ta có thể thực hiện lấy ra cạnh có trọng số nhỏ nhất, có thể xóa cạnh đó và thêm cạnh mới vào T. Min-Heap ở đây ta có thể sử dụng priority queue.

- **Khởi tạo:**
  - Tập X, Y, Z lần lượt quản lý đỉnh bên trong cây khung, bên ngoài cây khung, và các cạnh theo trọng số nhỏ nhất đến lớn nhất.
  - Chọn s là đỉnh bất kì và là đỉnh khởi tạo, thêm tất cả các cạnh cùng trọng số kề với s vào min-heap.
- **Chương trình chính**
  - Với mỗi vòng lặp, chọn cạnh có trọng số nhỏ nhất từ priority queue(Min-Heap), nếu cạnh này liên kết với đỉnh nằm bên ngoài cây khung(thuộc tập Y) thì ta chọn cạnh này, và cho rằng ta đã có thêm 1 cạnh mới. Xóa cạnh này ra khỏi X.
  - Với cạnh này, ta thêm đỉnh nằm bên ngoài cây khung(thuôc tập Y) vào tập X.
  - Duyệt hết các cạnh kết nối đỉnh mới thêm vào tập T(priority queue) với đỉnh còn lại nối đến chưa thuộc X.
- **Độ phức tạp**
  - Với priority queue(Min-Heap), việc lấy ra cạnh nhỏ nhất, xóa cạnh nhỏ nhất, thêm cạnh đều mất độ phức tạp O(log(n)) cho mỗi thao tác.
  - Ta duyệt n – 1 lần tương ứng với việc thêm n – 1 cạnh và thêm, xóa khỏi Min-Heap tối đa m lần(m cạnh). Do đó độ phức tạp tổng quát là $O((n+m)*log_2 n)$.

2.2 Thuật toán Kruskal:

2.2.1 Mã giả chi tiết và phân tích độ phức tạp của thuật toán.

Mã giả:

```
// Input:
// G = (V, E) - Đồ thị vô hướng liên thông với danh sách kề và trọng số của từng cạnh e | E

// Output:
// T - Các cạnh của cây khung nhỏ nhất của G

// Pseudocode Kruskal:

function Kruskal(G):
    T := {}                         // Tập rỗng cho các cạnh của MST
    sort edges of E by cost         // Sắp xếp các cạnh của E theo trọng số (Ex: Merge Sort)

    // Khởi tạo cấu trúc dữ liệu để quản lý tập hợp (Union-Find)
    initialize a disjoint set DS for vertices in V

    // Chương trình chính:
    for each edge e | E, in nondecreasing order of cost do:
        (u, v) = e                  // Lấy đỉnh u và v từ cạnh e
        if DS.find(u) | DS.find(v): // Kiểm tra xem u và v có thuộc cùng một tập hợp không
            T := T | {e}            // Thêm cạnh e vào T
            DS.union(u, v)          // Hợp nhất tập hợp chứa u và v

    return T                        // Trả về tập T là cây khung nhỏ nhất (MST)


```

Phân tích độ phức tạp:

- **Khởi tạo:**

- Khởi tạo tập rỗng lưu các cạnh của cây khung nhỏ nhất là T
- Các cạnh trong E được sắp xếp theo trọng số để xử lý từ nhỏ đến lớn.

- **Cấu trúc dữ liệu trong DS(Union-Find):** For trâu để tìm tập mà đỉnh thuộc về.
- **Chương trình chính:**
  - Duyệt qua từng cạnh trong danh sách đã sắp xếp.
  - Nếu hai đỉnh của cạnh không thuộc cùng một tập hợp (tức là không tạo thành chu trình), thì thêm cạnh vào cây khung và hợp nhất hai tập hợp
- **Độ phức tạp cuối cùng:** Việc kiểm tra hai đỉnh có thuộc cùng tập hợp hay không, vì ta for trâu nên độ phức tạp tệ nhất là O(n). Ta thêm tổng cộng n – 1 cạnh để tạo thành cây khung nhỏ nhất. Độ phức tạp cuối cùng sẽ là O(n^2).

2.2.2 Phương pháp mà bạn đã đề ra có cho độ phức tạp là O((n+m)log(n)) (với n là số đỉnh còn m là số cạnh) không? Nếu không thì đề xuất một phương pháp khác cho độ phức tạp như trên.

Phương pháp khác dùng thuật toán DSU(Disjoint Sets Union)
Ta chỉ cần tối ưu cấu trúc dữ liệu trong DS, kiểm tra 2 đỉnh của cạnh có thuộc cùng 1 tập hợp trong O(log) bằng thuật toán DSU

```
void make_set(int v) {
    lab[u] = -1;
}

int find_set(int v) {
    return lab[v] < 0 ? v : lab[v] = find_set(lab[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);

    if (a != b) {
        if (lab[a] > lab[b]) swap(a, b);
        lab[a] += lab[b];
        lab[b] = a;
    }
}
```

Hình 1: DSU - Disjoint Set Union

Độ phức tạp:

- Sắp xếp tập cạnh có độ phức tạp $O(m * log_2 m)$
- DSU(Disjoint Sets Union), Union-Find với chi phí $O(log_2 n)$ nhưng $log_2 n$ ở đây nhỏ đến mức ta có thể xem là O(1). Tổng thời gian của bước duyệt cạnh và sử dụng thuật toán DSU là O$O(m * log_2 n)$ nhưng có thể coi là O(m).

Tổng độ phức tạp:

- Sắp xếp: $O(m * log_2 m)$
- Union-Find m lần: $O(m * log_2 n) \approx O(m)$

Vậy, ta có thể kết luận độ phức tạp cuối cùng của thuật toán kruskal là
O((n + m)* log(n))