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

Khoa Khoa học máy tính

Môn học Phân tích và thiết kế thuật toán

Bài tập chủ đề 1

Nhóm 11:
Đặng Quang Vinh
MSSV: 23521786
Cao Lê Công Thành
MSSV: 23521437

1 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.
- 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 trên

- **Khởi tạo:** Với mỗi phần tử a trong tập A, tạo một cây chỉ chứa một nút, gán nhãn là "a". Độ phức tạp là $O(n)$, với n là số phần tử trong A.
- **Vòng lặp chính:** Trong mỗi lần lặp, chọn hai cây có xác suất nhỏ nhất, hợp nhất chúng và thêm cây mới vào tập F. Độ phức tạp của việc này là $O(n^2)$ vì phải duyệt toàn bộ mảng . Vòng lặp này chạy $n-1$ lần, do đó tổng độ phức tạp là $O(n^2)$.
- **Tổng kết:** Độ phức tạp của thuật toán là $O(n^2)$.

1.2 Hãy đưa ra một giải pháp để tối ưu thuật toán vừa được nêu trên

Thay vì sử dụng mảng để lưu toàn bộ cây rồi duyệt toàn bộ mảng để tìm ra 2 cây có tần suất nhỏ nhất nó sẽ có độ phức tạp là O(n) thì chúng ta có thể sử dụng hàng đợi ưu tiên (priority queue). Hàng đợi ưu tiên cho phép chúng ta thực hiện các thao tác chèn và lấy phần tử có giá trị nhỏ nhất với độ phức tạp ($O(\log n)$). Vậy độ phức tạp của thuật toán trên sẽ giảm xuống còn $O(n \log(n))$.

- Dưới đây là mã giả cho thuật toán trên

```
// Khởi tạo hàng đợi ưu tiên
priority_queue F

// Thêm tất cả các cây vào hàng đợi ưu tiên
for each a in A:
    T_a = tree containing only one node, labeled "a"
    F.push(T_a, P(T_a))

// Vòng lặp chính
while F.size() > 1:
    T1 = F.pop() // Lấy cây có xác suất nhỏ nhất
    T2 = F.pop() // Lấy cây có xác suất nhỏ thứ hai
    T3 = merge(T1, T2) // Hợp nhất hai cây
    P(T3) = P(T1) + P(T2)
    F.push(T3, P(T3)) // Thêm cây mới vào hàng đợi ưu tiên

return F.pop() // Trả về cây cuối cùng
```

2 Bài 2: Thuật toán Minimum Spanning Tree.

2.1 Câu 1

Prim

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.
Output: the edges of a minimum spanning tree of G.

// Initialization
$X := \{s\}$ // s is an arbitrarily chosen vertex
$T := \emptyset$ // invariant: the edges in T span X
// Main loop
while there is an edge (v, w) with $v \in X, w \notin X$ **do**
    $(v^*, w^*) :=$ a minimum-cost such edge
    add vertex w^* to X
    add edge (v^*, w^*) to T
return T

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

- Input:
  - Đồ thị vô hướng liên thông G=(V,E) với tập đỉnh V và tập cạnh E
  - Trọng số W=(u,v) của mỗi cạnh (u,v)
- Output: Cây khung nhỏ nhất (MST) chứa danh sách các cạnh của đồ thị.

```
1 // Prim(G, w): Tim cay khung nho nhat cua do thi G
2
3 Prim(G, w):
4     Chon dinh ban dau s bat ky tu tap V     // Khoi tao mot
          dinh bat dau
```

```
    X := {s}                                    // Tap dinh da them vao cay
khung
    T := rong                                   // Tap canh cua cay khung nho nhat
    PQ := hang doi uu tien rong                 // Su dung hang doi uu tien de quan ly cac canh theo trong so

    // Buoc 1: Dua cac canh noi voi dinh s vao hang doi uu tien
    for moi dinh v ke voi s:
        Chen canh (s, v) vao PQ voi trong so w(s, v)

    // Buoc 2: Xay dung cay khung nho nhat
    while kich thuoc cua X < so luong dinh trong V:
        (u, v) := PQ.lay_canh_nho_nhat()        // Lay canh co trong so nho nhat trong hang doi
        if v khong thuoc X:                     // Neu dinh v chua co trong cay khung
            Them v vao X                        // Them v vao cay khung
            Them canh (u, v) vao T              // Them canh (u, v) vao tap cac canh cua cay khung

            // Buoc 3: Dua cac canh noi tu v voi cac dinh chua co trong cay khung vao hang doi
            for moi dinh w ke voi v:
                if w khong thuoc X:
                    Chen canh (v, w) vao PQ voi trong so w(v, w)

    Tra ve T                                    // Tra ve tap cac canh cua cay khung nho nhat
```

- Phân tích độ phức tạp:
  - Input:
    - Dòng đầu tiên chứa một số nguyên n, biểu thị số lượng đỉnh của đồ thị.
    - Dòng thứ hai chứa một số nguyên m, biểu thị số lượng cạnh của đồ thị.
  - Cài đặt thuật toán:

* Khởi tạo
  - Tập X chứa các đỉnh đã được thêm vào cây khung.
  - Tập Y chứa các đỉnh còn lại chưa thuộc cây khung.
  - Tập T lưu trữ các cạnh cùng trọng số.
  - Chọn một đỉnh bất kỳ s và thêm các cạnh nối từ s đến các đỉnh kề vào tập T
* Chương trình chính:
  - Chọn cạnh có trọng số nhỏ nhất từ tập T, cạnh đó nối đỉnh u thuộc tập X (cây khung) với đỉnh v thuộc tập Y (ngoài cây khung).
  - Để thực hiện việc này, có thể sử dụng các cấu trúc dữ liệu như danh sách liên kết hoặc ma trận trọng số để biểu diễn đồ thị.
  - Để tìm cạnh có trọng số nhỏ nhất, ta duyệt qua các đỉnh u thuộc tập X và xem xét các cạnh nối từ u đến các đỉnh v thuộc tập Y. Việc tìm cạnh nhỏ nhất mất thời gian O(m) cho mỗi lần duyệt.
  - Quá trình trên được lặp lại cho đến khi n-1 cạnh đã được thêm vào cây khung.

– Tổng độ phức tạp: Vì ta cần thêm tổng cộng n-1 cạnh, và mỗi lần thêm cạnh mất thời gian O(m), tổng độ phức tạp của thuật toán là 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 bạn hãy đề xuất một phương pháp khác cho độ phức tạp như trên.

- Phương pháp dùng Min-Heap (Priority Queue)
  – Khởi tạo:
    * Ta sử dụng các tập hợp để quản lý các đỉnh:
      - Tập X: chứa các đỉnh đã nằm trong cây khung nhỏ nhất.
      - Tập Y: chứa các đỉnh còn lại chưa nằm trong cây khung.
      - Min-Heap Z (hay hàng đợi ưu tiên): quản lý các cạnh với trọng số từ nhỏ đến lớn.

    - Bắt đầu từ một đỉnh khởi tạo s bất kỳ, thêm tất cả các cạnh nối từ s vào Min-Heap dựa trên trọng số của chúng.
- Chương trình chính: Trong mỗi vòng lặp, ta thực hiện:
    - Lấy cạnh có trọng số nhỏ nhất từ Min-Heap Z.
    - Nếu cạnh này kết nối một đỉnh trong cây khung X với một đỉnh chưa có trong cây khung (thuộc tập Y), ta chọn cạnh đó, và thêm đỉnh ngoài cây khung này vào cây khung.
    - Sau khi thêm đỉnh mới vào tập X, ta duyệt qua tất cả các cạnh kết nối từ đỉnh vừa thêm với các đỉnh còn lại thuộc tập Y. Những cạnh này sẽ được thêm vào Min-Heap Z nếu chúng chưa thuộc cây khung.
- Độ phức tạp:
    - Sử dụng Min-Heap cho hàng đợi ưu tiên giúp thao tác lấy ra cạnh có trọng số nhỏ nhất, thêm cạnh, và xóa cạnh có độ phức tạp là O(logn) cho mỗi thao tác.
    - Thuật toán cần duyệt qua n-1 cạnh để hoàn thành cây khung nhỏ nhất. Với mỗi đỉnh, ta có thể thực hiện tối đa m thao tác thêm hoặc xóa cạnh vào/ra khỏi Min-Heap.
    - Vì thế, độ phức tạp tổng thể của thuật toán là O((n+m)*logn).

2.2 Câu 2

Kruskal

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.
Output: the edges of a minimum spanning tree of G.

```
// Preprocessing
T := ∅
sort edges of E by cost  // e.g., using MergeSort[26]
// Main loop
for each e ∈ E, in nondecreasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

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

```
// Input:
// G = (V, E) - Do thi vo huong lien thong voi danh sach ke va trong so cua tung canh e thuoc E

// Output:
// T - Cac canh cua cay khung nho nhat cua G

// Pseudocode Kruskal:

function Kruskal(G):
    T := {}                                  // Tap rong de luu cac canh cua cay khung nho nhat
    Sap xep cac canh trong E theo thu tu tang dan ve trong so (Vi du: Merge Sort)

    // Khoi tao cau truc du lieu Union-Find de quan ly cac tap dinh
    Khoi tao tap roi rac DS cho cac dinh trong V

    // Chuong trinh chinh:
    for moi canh e thuoc E (theo thu tu da sap xep):
        (u, v) = e                           // Lay hai dinh u va v tu canh e
        if DS.tim_tap(u) != DS.tim_tap(v): // Kiem tra xem u va v co thuoc cung mot tap khong
            Them canh e vao T                // Them canh e vao cay khung T
            DS.hop_tap(u, v)                 // Hop nhat tap chua u va tap chua v

    Tra ve T                                 // Tra ve tap T, la cay khung nho nhat (MST)
```

- Phân tích độ phức tạp:
  - Khởi tạo:
    - Bắt đầu bằng việc tạo một tập rỗng T để lưu trữ các cạnh của cây khung nhỏ nhất.
    - Sau đó, sắp xếp các cạnh trong E theo trọng số từ nhỏ đến lớn, để có thể xử lý các cạnh theo thứ tự tăng dần của trọng

số.

- Cấu trúc dữ liệu DS (Union-Find): Sử dụng cấu trúc Union-Find để quản lý các tập hợp đỉnh. Khi cần tìm xem đỉnh thuộc tập hợp nào, ta phải duyệt qua (for trâu) để xác định.
- Chương trình chính:
  * Với danh sách các cạnh đã sắp xếp, ta duyệt qua từng cạnh một.
  * Nếu hai đỉnh của cạnh không thuộc cùng một tập hợp (tức là chưa tạo thành chu trình), cạnh đó sẽ được thêm vào cây khung T, đồng thời ta hợp nhất hai tập hợp chứa các đỉnh của cạnh đó.
- Độ phức tạp cuối cùng:
  * Để kiểm tra xem hai đỉnh có nằm trong cùng một tập hợp hay không, quá trình này trong trường hợp tệ nhất sẽ tốn độ phức tạp O(n) do việc duyệt qua từng tập hợp.
  * Vì có tối đa n - 1 cạnh cần thêm vào cây khung để hình thành cây khung nhỏ nhất, nên độ phức tạp cuối cùng của thuật toán 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 bạn hãy đề xuất một phương pháp khác cho độ phức tạp như trên.

- Một phương pháp khác tối ưu dựa trên DSU (Disjoint Sets Union) có thể giúp kiểm tra xem hai đỉnh có thuộc cùng một tập hợp trong O(log) nhờ thuật toán DSU.

```
    void make_set(int v) {
    lab[u] = -1;  // Khoi tao tap hop rieng le cho moi dinh
}

int find_set(int v) {
    return lab[v] < 0 ? v : lab[v] = find_set(lab[v]);  //
       Tim dai dien cua tap hop va gan lai de toi uu hoa
}

void union_sets(int a, int b) {
```

```
10      a = find_set(a);  // Tim dai dien cua dinh a
11      b = find_set(b);  // Tim dai dien cua dinh b
12
13      if (a != b) {  // Neu chung khong thuoc cung mot tap
14          if (lab[a] > lab[b]) swap(a, b);  // Gop tap nho hon
                vao tap lon hon
15          lab[a] += lab[b];  // Cap nhat so luong dinh trong
                tap
16          lab[b] = a;  // Hop nhat hai tap
17      }
18  }
```

- Phân tích độ phức tạp:
  - Sắp xếp các cạnh: Sắp xếp các cạnh trong danh sách có độ phức tạp O(m * log(m)), với m là số cạnh.
  - DSU (Disjoint Sets Union): Việc tìm tập đại diện của hai đỉnh qua thuật toán Union-Find có độ phức tạp trung bình O(log(n)), tuy nhiên, log(n) là rất nhỏ và có thể coi như O(1). Do đó, việc duyệt qua tất cả các cạnh và sử dụng DSU có thể được tính với độ phức tạp O(m * log(n)), nhưng vì log(n) nhỏ, ta có thể coi là O(m).
- Tổng độ phức tạp:
  - Sắp xếp các cạnh: O(m*log(m))
  - DSU (Union-Find) thực hiện m lần: O(m*log(n)) O(m)
  - Vì vậy, tổng độ phức tạp của thuật toán Kruskal được xác định là: O((n + m) * log(n)), trong đó n là số đỉnh và m là số cạnh.