ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC MÁY TÍNH

PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

Phân Tích Thuật Toán Không Đệ Quy

Nhóm 5 thực hiện

Môn học: CS112.P11.KHTN

Sinh viên thực hiện:

Nguyễn Thiên Bảo - 23520127

Trần Lê Minh Nhật - 23521098

Giáo viên hướng dẫn:

Nguyễn Thanh Sơn

Ngày 8 tháng 10 năm 2024

Mục lục

1 Bài 1: Huffman Coding

Huffman coding là một thuật toán nén nổi tiếng và 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 Phân tích và xác định độ phức tạp của thuật toán trên:

1.1.1 Phân tích thuật toán:

Thuật toán xây dựng cây Huffman trên thực hiện các bước sau:

- **Khởi tạo:**
  - Với mỗi ký tự a trong bảng chữ cái α, tạo một cây T_a chỉ chứa một nút gốc được gắn nhãn "a".
  - Gán xác suất $P(T_a) = p_a$.
  - Tập hợp F chứa tất cả các cây T_a.
- **Vòng lặp chính:**
  - Trong khi số lượng cây trong F lớn hơn hoặc bằng 2:
    - Tìm hai cây T_1 và T_2 có xác suất nhỏ nhất và nhỏ thứ hai trong F.

    * Loại bỏ T_1 và T_2 khỏi F.
    * Tạo cây mới T_3 bằng cách gộp T_1 và T_2, với T_1 và T_2 là con trái và con phải của T_3.
    * Gán xác suất $P(T_3) = P(T_1) + P(T_2)$.
    * Thêm T_3 vào F.

- **Kết quả:**
  - Thuật toán trả về cây duy nhất còn lại trong F, đó là cây Huffman.

1.1.2 Xác định độ phức tạp:

- **Số lần lặp:** Vòng lặp chính thực hiện $n-1$ lần, với n là số lượng ký tự trong bảng chữ cái α.
- **Chi phí mỗi lần lặp:**
  - **Tìm hai cây có xác suất nhỏ nhất và nhỏ thứ hai:**
    * Nếu sử dụng danh sách đơn giản và tìm kiếm tuyến tính, việc tìm hai cây này tốn $O(|F|)$ thời gian cho mỗi lần lặp.
    * Tổng thời gian cho tất cả các lần lặp là $O(n^2)$ vì $|F|$ giảm từ n xuống 2.
  - **Loại bỏ và thêm cây vào F:**
    * Thao tác này tốn $O(1)$ thời gian nếu sử dụng cấu trúc dữ liệu thích hợp.

1.1.3 Kết luận về độ phức tạp:

- **Độ phức tạp thời gian tổng quát:** $O(n^2)$ nếu sử dụng danh sách đơn giản.
- **Độ phức tạp tối ưu:** $O(n \log n)$ nếu sử dụng **hàng đợi ưu tiên** (heap).

1.1.4 Giải pháp để tối ưu thuật toán:

Sử dụng hàng đợi ưu tiên (heap) để quản lý tập F:

- **Khởi tạo hàng đợi ưu tiên:**

  - Thêm tất cả các cây T_a vào một hàng đợi ưu tiên dựa trên xác suất $P(T_a)$.
  - Việc này có thể thực hiện trong $O(n)$ thời gian bằng cách xây dựng heap từ danh sách ban đầu.

- **Thay đổi vòng lặp chính:**
  - Trong mỗi lần lặp:
    * **Extract-min hai lần:** Lấy ra hai cây có xác suất nhỏ nhất và nhỏ thứ hai trong $O(\log n)$ thời gian mỗi lần.
    * **Gộp cây và thêm vào hàng đợi:** Tạo cây mới và thêm vào hàng đợi ưu tiên trong $O(\log n)$ thời gian.

- **Độ phức tạp thời gian sau tối ưu:**
  - Mỗi lần lặp tốn $O(\log n)$ thời gian.
  - Tổng thời gian cho $n-1$ lần lặp là $O(n \log n)$.

Lợi ích của việc sử dụng hàng đợi ưu tiên:

- Giảm độ phức tạp từ $O(n^2)$ xuống $O(n \log n)$.
- Tối ưu hóa việc tìm kiếm và thao tác với các cây có xác suất nhỏ nhất.

Kết luận:

- Để tối ưu thuật toán xây dựng cây Huffman, cần sử dụng hàng đợi ưu tiên (heap) để quản lý tập F.
- Điều này giúp giảm đáng kể độ phức tạp thời gian từ $O(n^2)$ xuống $O(n \log n)$, làm cho thuật toán hiệu quả hơn cho các bộ dữ liệu lớn.

Tóm lại:

1. Độ phức tạp của thuật toán ban đầu là $O(n^2)$ nếu không sử dụng cấu trúc dữ liệu tối ưu.
2. Để tối ưu thuật toán, sử dụng hàng đợi ưu tiên (heap) để quản lý tập F, giảm độ phức tạp xuống $O(n \log n)$.

2 Bài 2: Thuật toán Minimum Spanning Tree

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

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

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.

2.1 Mã giả chi tiết cho thuật toán Prim và phân tích độ phức tạp

2.1.1 Mã giả chi tiết cho thuật toán Prim:

```
Input: Đồ thị vô hướng liên thông G = (V, E) với biểu diễn danh sách kề
       và chi phí c_e cho mỗi cạnh e thuộc E
Output: Các cạnh của cây khung nhỏ nhất của G

// Khởi tạo
X := {s}          // s là một đỉnh được chọn ngẫu nhiên
T := rỗng            // Các cạnh trong T sẽ kết nối các đỉnh trong X

// Vòng lặp chính
```

```
while có một cạnh (v, w) với v thuộc X, w không thuộc X do
    (v*, w*) := cạnh có chi phí nhỏ nhất trong các cạnh nối từ X đến V \ X
    thêm đỉnh w* vào X
    thêm cạnh (v*, w*) vào T
return T
```

2.1.2 Phân tích độ phức tạp của thuật toán:

Giả sử đồ thị G có n đỉnh và m cạnh. Trong mỗi lần lặp, thuật toán chọn cạnh có trọng số nhỏ nhất nối một đỉnh trong X với một đỉnh chưa thuộc X. Do đó, có các bước sau:

- **Khởi tạo:** Khởi tạo tập X và T có độ phức tạp $O(1)$.
- **Vòng lặp chính:** Vòng lặp thực hiện $n-1$ lần vì cây khung nhỏ nhất có $n-1$ cạnh.

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 thời gian tổng quát: tệ khi dùng hai vòng lặp là $O(n * m)$

2.1.3 Phương pháp cải thiện để đạt độ phức tạp $O((n+m)\log n)$

Độ phức tạp $O((n+m)\log n)$ có thể đạt được khi sử dụng **hàng đợi ưu tiên**:

- Sử dụng hàng đợi ưu tiên (heap) để quản lý các cạnh biên giữa X và $V \setminus X$, và lấy ra cạnh có trọng số nhỏ nhất trong $O(\log n)$ thời gian.
- Cập nhật các đỉnh kề với mỗi lần thêm đỉnh mới vào X cũng tốn $O(\log n)$ cho mỗi cạnh được thêm vào hàng đợi.

Như vậy, độ phức tạp là $O((n+m)\log n)$, đảm bảo hiệu quả hơn khi đồ thị có số cạnh lớn.

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
$T := \emptyset$
sort edges of E by cost // e.g., using MergeSort[26]
// Main loop
for each $e \in E$, in nondecreasing order of cost **do**
  if $T \cup \{e\}$ is acyclic **then**
    $T := T \cup \{e\}$
return T

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

2.2 Mã giả chi tiết cho thuật toán Kruskal và phân tích độ phức tạp

2.2.1 Mã giả chi tiết cho thuật toán Kruskal:

```
Input: Đồ thị vô hướng liên thông G = (V, E) với biểu diễn danh sách kề
       và chi phí c_e cho mỗi cạnh e thuộc E
Output: Các cạnh của cây khung nhỏ nhất của G

// Tiền xử lý
T := rỗng               // Khởi tạo tập T rỗng
sắp xếp các cạnh của E theo thứ tự tăng dần chi phí  // ví dụ sử dụng MergeSort

// Vòng lặp chính
for mỗi cạnh e thuộc E, theo thứ tự không giảm của chi phí do
    nếu T hội {e} không chứa chu trình thì
        T := T hội {e}
return T
```

2.2.2 Phân tích độ phức tạp của thuật toán:

- **Tiền xử lý (sắp xếp các cạnh):** Để sắp xếp các cạnh theo chi phí, nếu sử dụng thuật toán sắp xếp như MergeSort, độ phức tạp là $O(m \log m)$, với m là số cạnh trong đồ thị G.
- **Kiểm tra chu trình:** Để kiểm tra xem cạnh có tạo thành chu trình hay không, có thể sử dụng cấu trúc dữ liệu **Disjoint Set Union (DSU)** (hay Union-Find). Các thao tác 'union' và 'find' với DSU có độ phức tạp là $O(\log n)$ cho mỗi thao tác, nếu sử dụng kỹ thuật **path compression** và **union by rank**.
- **Độ phức tạp của vòng lặp chính:** Vòng lặp chính duyệt qua tất cả các cạnh m. Với mỗi cạnh, thực hiện phép kiểm tra chu trình và thêm vào tập T nếu không có chu trình, tệ nhất là n vòng lặp và thêm n-1 cạnh để tạo cây khung nhỏ nhất, tệ là $O(n^2)$

2.2.3 Phương pháp cải thiện để đạt độ phức tạp $O((n+m)\log n)$

Độ phức tạp $O((n+m)\log n)$ có thể đạt được như sau:

- **Sử dụng Disjoint Set Union (DSU):** Sử dụng cấu trúc DSU với các thao tác 'union' và 'find' kèm theo kỹ thuật **path compression** và **union by rank** giúp giảm độ phức tạp của các phép kiểm tra chu trình xuống $O(\log n)$ cho mỗi thao tác.
- **Sắp xếp các cạnh:** Độ phức tạp của việc sắp xếp là $O(m \log m)$. Tuy nhiên, với m cạnh, độ phức tạp có thể xem xét lại thành $O(m \log n)$ nếu m tỉ lệ thuận với n.

Kết luận: Với phương pháp trên, độ phức tạp của thuật toán Kruskal là $O((n+m)\log n)$, đặc biệt hiệu quả khi số cạnh m lớn hơn số đỉnh n.