Phân tích độ phức tạp các thuật toán không đệ quy

Nguyễn Trần Quang Minh, Hồ Ngọc Luật
Giảng viên: Nguyễn Thanh Sơn
Môn học: CS112

Ngày 8 tháng 10 năm 2024

Contents

1 Phân tích giải thuật Huffman Coding

1.1 Phân tích độ phức tạp của thuật toán

Thuật toán này là một dạng *greedy algorithm* và được sử dụng để xây dựng cây Huffman cho việc nén dữ liệu hiệu quả.

1.1.1 Bước khởi tạo (init)

Với mỗi ký tự trong tập hợp α, một cây con chỉ có một nút duy nhất được tạo. Việc này có độ phức tạp là $O(n)$, trong đó n là số lượng ký tự.

1.1.2 Vòng lặp chính (main loop)

Vòng lặp này thực hiện việc chọn 2 cây có xác suất nhỏ nhất $P(T_1)$ và $P(T_2)$, rồi hợp nhất chúng thành một cây mới.

- **Chọn hai cây có xác suất nhỏ nhất**: Nếu tập hợp F được lưu trữ dưới dạng một danh sách sắp xếp, việc tìm hai phần tử nhỏ nhất cần thời gian $O(n)$. Tuy nhiên, nếu sử dụng một *heap* (đống nhị phân), việc lấy ra hai phần tử nhỏ nhất sẽ có độ phức tạp là $O(\log n)$.
- **Xóa và thêm cây mới vào tập hợp** F: Thao tác này cũng có độ phức tạp $O(\log n)$ nếu sử dụng heap.

Vòng lặp này chạy tổng cộng $n - 1$ lần (vì mỗi lần hợp nhất, số cây trong F giảm đi một), do đó tổng độ phức tạp của vòng lặp chính là $O(n \log n)$.

1.1.3 Tổng độ phức tạp

Độ phức tạp của thuật toán Huffman chính là $O(n \log n)$, trong đó n là số lượng ký tự trong bảng mã.

1.2 Giải pháp tối ưu thuật toán

Để tối ưu hóa thuật toán này, thay vì lưu các cây trong một danh sách đơn thuần (phải tốn thời gian để sắp xếp và tìm phần tử nhỏ nhất), chúng ta có thể sử dụng một *heap* (min-heap) để lưu trữ và quản lý các cây con. Heap cho phép:

- **Tìm và loại bỏ phần tử nhỏ nhất** trong thời gian $O(\log n)$.
- **Thêm phần tử mới** vào heap cũng trong thời gian $O(\log n)$.

Với cách sử dụng heap này, thuật toán sẽ được tối ưu với độ phức tạp $O(n \log n)$, thay vì $O(n^2)$ nếu sử dụng danh sách không sắp xếp.

Tóm tắt tối ưu thuật toán Huffman

- Sử dụng **min-heap** thay vì danh sách đơn giản để quản lý cây con trong tập F.

2 Phân tích thuật toán Prim

2.1 Mã giả của thuật toán Prim

Thuật toán Prim được sử dụng để tìm cây khung nhỏ nhất (Minimum Spanning Tree - MST) cho một đồ thị liên thông vô hướng. Thuật toán này lần lượt chọn cạnh có trọng số nhỏ nhất kết nối các đỉnh chưa được bao phủ với cây khung hiện tại.

Mã giả chi tiết

```
// Input: Đồ thị liên thông vô hướng G = (V, E), mỗi cạnh e  E có
trọng số c(e)
// Output: Tập hợp các cạnh của cây khung nhỏ nhất (MST)

X := {s}              // s là đỉnh bắt đầu tùy ý
T :=                 // Tập các cạnh của MST ban đầu rỗng

while X  V do        // Khi tập X chưa bao gồm tất cả các đỉnh
    Chọn cạnh (v*, w*) với v*  X và w*  X có trọng số nhỏ nhất
    Thêm w* vào X     // Thêm đỉnh mới vào tập đỉnh đã duyệt
    Thêm cạnh (v*, w*) vào T // Thêm cạnh vào cây MST
return T              // Trả về cây khung nhỏ nhất
```

2.2 Phân tích độ phức tạp của thuật toán

Độ phức tạp của thuật toán Prim phụ thuộc vào cách lưu trữ đồ thị và cách tìm cạnh có trọng số nhỏ nhất trong mỗi bước.

2.2.1 Độ phức tạp thời gian cơ bản

- Nếu sử dụng danh sách kề hoặc ma trận kề để tìm cạnh nhỏ nhất trong mỗi lần lặp, thì việc tìm cạnh nhỏ nhất mất thời gian $O(n)$, trong đó n là số lượng đỉnh. - Do đó, với mỗi lần thêm một đỉnh vào cây khung, ta cần lặp qua tất cả các cạnh để tìm cạnh nhỏ nhất, dẫn đến tổng thời gian là $O(n^2)$ cho toàn bộ thuật toán.

2.2.2 Phương pháp tối ưu với heap (min-heap)

Để tối ưu hóa việc tìm cạnh có trọng số nhỏ nhất, ta có thể sử dụng một **hàng đợi ưu tiên** (priority queue) hoặc **min-heap**. Cấu trúc heap cho phép ta:

- Tìm cạnh nhỏ nhất trong $O(\log n)$ thay vì $O(n)$.
- Thêm cạnh và cập nhật heap trong $O(\log n)$.

Khi sử dụng heap, ta thực hiện các thao tác sau trong mỗi vòng lặp:

1. Lấy ra cạnh có trọng số nhỏ nhất kết nối đỉnh trong X với một đỉnh ngoài X trong thời gian $O(\log n)$.
2. Thêm đỉnh mới và cập nhật heap trong thời gian $O(\log n)$ cho mỗi cạnh kề.

Với đồ thị có n đỉnh và m cạnh, mỗi thao tác cập nhật heap mất $O(\log n)$, và mỗi cạnh chỉ được xét đúng một lần. Do đó, tổng độ phức tạp của thuật toán là:

$$O((n+m)\log n)$$

2.3 Tóm tắt phương pháp tối ưu

Phương pháp tối ưu giúp cải thiện độ phức tạp từ $O(n^2)$ (với cách tiếp cận đơn giản) xuống $O((n+m)\log n)$ bằng cách sử dụng **heap (min-heap)**. Trong đó:

- n là số đỉnh của đồ thị.
- m là số cạnh của đồ thị.

Sử dụng heap giúp giảm thời gian tìm kiếm cạnh nhỏ nhất và thêm đỉnh mới vào cây khung nhỏ nhất.

3 Phân tích thuật toán Kruskal

3.1 Mã giả của thuật toán Kruskal

Thuật toán Kruskal được sử dụng để tìm cây khung nhỏ nhất (Minimum Spanning Tree - MST) cho một đồ thị liên thông vô hướng. Thuật toán hoạt động dựa trên việc sắp xếp các cạnh theo thứ tự tăng dần của trọng số và thêm chúng vào cây khung nhỏ nhất miễn là không tạo ra chu trình.

Mã giả chi tiết

```
// Input: Đồ thị liên thông vô hướng G = (V, E), mỗi cạnh e  E có
trọng số c(e)
// Output: Tập hợp các cạnh của cây khung nhỏ nhất (MST)

// Preprocessing
T :=                    // Tập các cạnh của MST ban đầu rỗng
Sắp xếp các cạnh trong E theo thứ tự tăng dần của trọng số

// Main loop
for mỗi cạnh e  E (theo thứ tự tăng dần của trọng số) do
    if T  {e} không tạo chu trình then
        Thêm e vào T    // Thêm cạnh vào cây khung
return T                // Trả về cây khung nhỏ nhất
```

3.2 Phân tích độ phức tạp của thuật toán

3.2.1 Độ phức tạp thời gian cơ bản

1. **Bước sắp xếp các cạnh**: Ta cần sắp xếp các cạnh theo trọng số. Việc sắp xếp này mất thời gian $O(m \log m)$, với m là số lượng cạnh của đồ thị. 2. **Duyệt qua

các cạnh đã sắp xếp**: Ta lần lượt duyệt qua các cạnh theo thứ tự đã sắp xếp và kiểm tra xem việc thêm cạnh đó vào cây khung có tạo thành chu trình hay không. Để kiểm tra chu trình, ta có thể sử dụng cấu trúc **Union-Find** (còn gọi là **Disjoint Set Union - DSU**). Thao tác kiểm tra và hợp nhất tập hợp với DSU có thể thực hiện trong thời gian gần như hằng số $O(\alpha(n))$, trong đó $\alpha(n)$ là hàm đảo của hàm Ackermann, một hàm tăng rất chậm và có thể coi là nhỏ hơn $O(\log n)$.

3.2.2 Phương pháp tối ưu và độ phức tạp thời gian

- Tổng thời gian để kiểm tra và hợp nhất $n-1$ cạnh (vì cây khung có $n-1$ cạnh, với n là số đỉnh) bằng cách sử dụng DSU là $O(m\alpha(n))$. - Do đó, tổng thời gian chạy của thuật toán Kruskal là:

$$O(m \log m + m\alpha(n)) \approx O(m \log m)$$

Vì $m \geq n-1$, ta có thể coi độ phức tạp là $O(m \log n)$ (do $\log m \leq \log n^2 = 2 \log n$).

3.3 Tóm tắt

Phương pháp sử dụng thuật toán Kruskal với Union-Find giúp đạt được độ phức tạp thời gian là $O(m \log n)$, với m là số cạnh và n là số đỉnh của đồ thị. Thuật toán này hiệu quả cho đồ thị thưa, nơi m nhỏ hơn nhiều so với n^2.