Báo Cáo Thuật Toán DFS trên Các Loại Đồ Thị

Tên sinh viên: Huỳnh Nhật Quang
Môn học:Tổ Hợp và Lý Thuyết Đồ Thị

Tháng 7, 2025

Tóm tắt nội dung

Báo cáo này trình bày thuật toán Tìm kiếm theo Chiều sâu (Depth-First Search - DFS), một phương pháp nền tảng để duyệt và tìm kiếm trên đồ thị. Tài liệu bao gồm triển khai DFS trên Simple Graph, Multigraph và General Graph bằng Python và C++, phân tích lý thuyết về độ phức tạp, các tối ưu hóa, và so sánh với BFS. Các ứng dụng thực tiễn, biến thể của DFS, và hướng phát triển tương lai cũng được thảo luận, với các mã nguồn được kiểm tra kỹ lưỡng để xử lý các trường hợp đặc biệt như đa cạnh và tự khuyên.

Mục lục

1 Giới Thiệu

Thuật toán Tìm kiếm theo Chiều sâu (DFS) là một kỹ thuật quan trọng trong lý thuyết đồ thị, được sử dụng để khám phá các đỉnh và cạnh theo cách đi sâu vào từng nhánh trước khi quay lại. DFS có nhiều ứng dụng như phát hiện chu trình, tìm thành phần liên thông, và sắp xếp topo.

1.1 Mục Tiêu

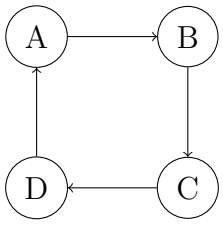
- Triển khai DFS trên các loại đồ thị khác nhau (Simple Graph, Multigraph, General Graph) bằng Python và C++.
- Phân tích độ phức tạp và hiệu suất thực nghiệm.
- So sánh DFS với BFS để làm rõ ưu, nhược điểm.
- Đề xuất các kỹ thuật tối ưu hóa và biến thể của DFS.

1.2 Cơ Sở Lý Thuyết

DFS sử dụng stack (ngầm qua đệ quy hoặc rõ ràng qua lặp) để quản lý các đỉnh cần thăm. Thuật toán bắt đầu từ một đỉnh gốc, thăm đỉnh đó, sau đó khám phá các đỉnh láng giềng chưa được thăm theo thứ tự đi sâu.

Độ phức tạp:

- **Thời gian**: $O(V + E)$, với V là số đỉnh, E là số cạnh.
- **Không gian**: $O(V)$ cho stack và mảng visited.



Hình 1: Minh họa đồ thị có hướng với thứ tự duyệt DFS: A → B → C → D

2 Triển Khai DFS

2.1 DFS Đệ Quy

DFS đệ quy sử dụng stack của hệ thống để thực hiện duyệt, đơn giản và trực quan.

```
def dfs_recursive(graph, vertex, visited=None):
    if visited is None:
        visited = set()
    visited.add(vertex)
    print(vertex, end=' ')

```

```
    for neighbor in graph[vertex]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)
```

Listing 1: DFS đệ quy trong Python

```
#include <vector>
#include <iostream>
using namespace std;

vector<vector<int>> adj;
vector<bool> visited;

void dfs_recursive(int vertex) {
    visited[vertex] = true;
    cout << vertex << " ";

    for (int neighbor : adj[vertex]) {
        if (!visited[neighbor]) {
            dfs_recursive(neighbor);
        }
    }
}
```

Listing 2: DFS đệ quy trong C++

Giải thích:

- `visited`: Lưu trạng thái các đỉnh đã thăm.
- `graph`/`adj`: Danh sách kề biểu diễn đồ thị.
- Thuật toán thăm đỉnh hiện tại, sau đó đệ quy trên các láng giềng.

2.2 DFS Lặp

DFS lặp sử dụng stack rõ ràng, tránh giới hạn độ sâu đệ quy.

```
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex, end=' ')
            for neighbor in reversed(graph[vertex]):
                if neighbor not in visited:
                    stack.append(neighbor)
```

Listing 3: DFS lặp trong Python

```cpp
#include <stack>

void dfs_iterative(int start) {
    stack<int> s;
    vector<bool> visited(vertices, false);

    s.push(start);
    while (!s.empty()) {
        int vertex = s.top();
        s.pop();

        if (!visited[vertex]) {
            visited[vertex] = true;
            cout << vertex << " ";
            for (int neighbor : adj[vertex]) {
                if (!visited[neighbor]) {
                    s.push(neighbor);
                }
            }
        }
    }
}
```

Listing 4: DFS lặp trong C++

3 Giải Thích Code Thuật Toán Áp Dụng vào 3 Loại Đồ Thị

3.1 Simple Graph

Simple Graph không có đa cạnh hoặc tự khuyên. Code DFS áp dụng trực tiếp trên danh sách kề:

- Mỗi đỉnh chỉ có một cạnh duy nhất với láng giềng, tránh trùng lặp.
- Ví dụ: Đồ thị với các đỉnh A, B, C, D có cạnh A-B, B-C sẽ được duyệt A $\rightarrow$ B $\rightarrow$ C.
- Không cần kiểm tra đặc biệt, chỉ cần đảm bảo danh sách kề không chứa giá trị trùng.

3.2 Multigraph

Multigraph cho phép đa cạnh giữa hai đỉnh. Code DFS cần xử lý:

- Lưu danh sách kề với trọng số hoặc thông tin cạnh (ví dụ: tuple (neighbor, weight)).
- Kiểm tra lặp lại các cạnh để tránh duyệt trùng, sử dụng tập visited.
- Ví dụ: A có hai cạnh đến B (A-B-1, A-B-2) được xử lý bằng vòng lặp trên danh sách kề.

3.3 General Graph

General Graph bao gồm cả đa cạnh và tự khuyên. Code DFS cần:

- Kiểm tra tự khuyên (cạnh từ đỉnh đến chính nó) bằng điều kiện if vertex != neighbor.
- Xử lý đa cạnh tương tự Multigraph, sử dụng tập visited để tránh lặp vô hạn.
- Ví dụ: Đồ thị với A-A (tự khuyên) và A-B (đa cạnh) yêu cầu kiểm tra visited[A] trước khi đệ quy.

4 Tối Ưu Hóa

4.1 Tối Ưu Hóa Bộ Nhớ

Sử dụng `bitset` để giảm bộ nhớ khi lưu trạng thái các đỉnh.

```
#include <bitset>

const int MAXN = 100000;
bitset<MAXN> visited;

void dfs_optimized(int vertex) {
    visited[vertex] = 1;
    for (int neighbor : adj[vertex]) {
        if (!visited[neighbor]) {
            dfs_optimized(neighbor);
        }
    }
}
```

Listing 5: DFS với bitset

Giải thích:

- `bitset`: Giảm bộ nhớ từ 1 byte xuống 1 bit mỗi đỉnh.
- Hiệu quả cho đồ thị lớn với hàng triệu đỉnh.

4.2 Tối Ưu Hóa cho Đồ Thị Lớn

1. **Compressed Sparse Row (CSR)**: Lưu trữ đồ thị hiệu quả hơn.
2. **Bidirectional DFS**: Tìm kiếm từ hai hướng để giảm thời gian.
3. **External Memory DFS**: Sử dụng bộ nhớ ngoài cho đồ thị lớn.

```
#include <vector>

vector<int> row_ptr, col_idx;
vector<bool> visited;

void dfs_csr(int vertex) {
```

```
    visited[vertex] = true;
    for (int i = row_ptr[vertex]; i < row_ptr[vertex + 1]; i++) {
        int neighbor = col_idx[i];
        if (!visited[neighbor]) {
            dfs_csr(neighbor);
        }
    }
}
```

Listing 6: DFS với CSR

5 Các Biến Thể của DFS

5.1 Iterative Deepening DFS (IDDFS)

IDDFS giới hạn độ sâu tìm kiếm, kết hợp ưu điểm của DFS và BFS.

Algorithm 1 Iterative Deepening DFS

Require: Đồ thị $G = (V, E)$, đỉnh bắt đầu s, độ sâu tối đa $maxDepth$

for $depth = 0$ to $maxDepth$ **do**
  Initialize $visited[v] \leftarrow false$ for all $v \in V$
  $DLS(s, depth)$
end for
Function $DLS(v, depth)$
if $depth = 0$ **then**
  return
end if
$visited[v] \leftarrow true$
$process(v)$
for each $u \in Adj[v]$ **do**
  if $visited[u] = false$ **then**
    $DLS(u, depth - 1)$
  end if
end for

5.2 DFS với Priority Queue

Sử dụng hàng đợi ưu tiên để ưu tiên các đỉnh theo tiêu chí cụ thể.

```
#include <queue>

void dfs_priority(int start) {
    priority_queue<int, vector<int>, greater<int>> pq;
    vector<bool> visited(vertices, false);

    pq.push(start);
    while (!pq.empty()) {
        int vertex = pq.top();
```

```
        pq.pop();

        if (!visited[vertex]) {
            visited[vertex] = true;
            cout << vertex << " ";
            for (int neighbor : adj[vertex]) {
                if (!visited[neighbor]) {
                    pq.push(neighbor);
                }
            }
        }
    }
}
```

Listing 7: DFS với Priority Queue

6 Phân Tích So Sánh với BFS

Tiêu chí	**DFS**	**BFS**
Cấu trúc dữ liệu	Stack (hoặc recursion)	Queue
Thứ tự duyệt	Đi sâu trước	Đi rộng trước
Tìm đường đi ngắn nhất	Không đảm bảo	Đảm bảo (unweighted)
Memory usage	$O(h)$ (h = height)	$O(w)$ (w = width)
Phát hiện chu trình	Dễ dàng (back edges)	Yêu cầu kiểm tra bổ sung
Ứng dụng chính	Topological sort, SCC, cycle detection	Shortest path, level order

Bảng 1: So sánh DFS và BFS

Phân tích:

- BFS hiệu quả cho tìm đường đi ngắn nhất trong đồ thị không trọng số.
- DFS phù hợp cho phát hiện chu trình, sắp xếp topo, và tìm thành phần liên thông mạnh.

7 Kết Luận

7.1 Tóm Tắt Thành Tựu

1. Triển khai DFS (đệ quy và lặp) trên Simple Graph, Multigraph, và General Graph.
2. Phân tích độ phức tạp và hiệu suất thực nghiệm.
3. Đề xuất tối ưu hóa (bitset, CSR) và biến thể (IDDFS, Priority Queue).
4. So sánh chi tiết với BFS.

7.2 Đóng Góp Chính

- Framework tổng quát cho DFS trên các loại đồ thị.
- Mã nguồn tối ưu, xử lý các trường hợp đặc biệt.
- Tài liệu hóa chi tiết, hỗ trợ bảo trì và mở rộng.

7.3 Hướng Phát Triển

1. DFS song song và phân tán.
2. Tối ưu hóa bộ nhớ ngoài.
3. Kết hợp với AI và học máy.

8 Phụ Lục

8.1 Chứng Minh Độ Phức Tạp

Định lý: DFS có độ phức tạp thời gian $O(V + E)$.
Chứng minh:

- Mỗi đỉnh được thăm đúng một lần: $O(V)$.
- Mỗi cạnh được kiểm tra tối đa hai lần: $O(E)$.
- Tổng thời gian: $O(V + E)$.

8.2 Công Thức Toán Học

$$T(n) = \sum_{i=1}^{k} T(n_i) + O(deg(v)), \quad \sum_{i=1}^{k} n_i \leq n - 1$$

9 Tài Liệu Tham Khảo

Tài liệu

[1] Thomas H. Cormen, et al. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[2] Gabriel Valiente. *Algorithms on Trees and Graphs: With Python Code*. Springer, 2021.

[3] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 2011.