Báo Cáo Cuối kỳ Môn Tổ Hợp Và Lý Thuyết Đồ Thị
Project 4: Graph & Tree Traversing Problems

Trần Mạnh Đức

Ngày 27 tháng 7 năm 2025

Mục lục

1 Bài toán 4: Chuyển đổi giữa các Dạng Biểu diễn Đồ thị và Cây 4
1.1 Phân tích Yêu cầu và Chiến lược Thực hiện . . . 4
1.1.1 Chiến lược Thiết kế: Sử dụng một "Cấu trúc Trung gian" . . . 4
1.2 Thiết kế Cấu trúc Dữ liệu . . . 4
1.2.1 Cấu trúc Dữ liệu C++ . . . 4
1.2.2 Cấu trúc Dữ liệu Python . . . 5
1.3 Cài đặt các Hàm Chuyển đổi (Python) . . . 6
1.4 Cài đặt các Hàm Chuyển đổi (C++) . . . 8
1.5 Kết luận cho Bài toán 4 . . . 9

2 Bài toán 5: Giải các Bài toán và Bài tập Lý thuyết 10
2.1 Giải Problem 1.1 - 1.6 . . . 10
2.1.1 Problem 1.1: Size of Complete and Complete Bipartite Graphs . 10
2.1.2 Problem 1.2: Bipartite Nature of Cycle and Complete Graphs . 10
2.1.3 Problem 1.3: Spanning Trees . . . 11
2.1.4 Problem 1.4: Extend Adjacency Matrix Representation . . . 14
2.1.5 Problem 1.5: Extend First-Child, Next-Sibling Representation . 15
2.1.6 Problem 1.6: Verify a Tree Representation . . . 16
2.2 Giải Exercises 1.1 - 1.10 . . . 17
2.2.1 Exercise 1.1: Read and Write DIMACS Format . . . 17
2.2.2 Exercise 1.2: Read and Write SGB Format . . . 19
2.2.3 Exercise 1.3: Generate Path, Cycle, and Wheel Graphs . . . 21
2.2.4 Exercise 1.4: Generate Complete and Complete Bipartite Graphs 22
2.2.5 Exercise 1.5: Implement Extended Adjacency Matrix . . . 23
2.2.6 Exercise 1.6: Enumerate Perfect Matchings in Kp,q . . . 24
2.2.7 Exercise 1.7: Generate Complete Binary Tree . . . 25
2.2.8 Exercise 1.8: Generate Random Trees . . . 26
2.2.9 Exercise 1.9: Implement T.previous_sibling . . . 27
2.2.10 Exercise 1.10: Implement Extended First-Child, Next-Sibling Tree 28

3 Các Thuật toán Duyệt Đồ thị và Cây 30
3.1 Bài toán 8: BFS trên Đơn đồ thị . . . 30
3.1.1 Phân tích và Nền tảng Lý thuyết . . . 30
3.1.2 Mô tả Thuật toán và Cấu trúc Dữ liệu . . . 31
3.1.3 Mô phỏng Thuật toán . . . 32
3.1.4 Cài đặt bằng Python . . . 32
3.1.5 Cài đặt bằng C++ . . . 34
3.1.6 Độ phức tạp . . . 35
3.2 Bài toán 9: BFS trên Đa đồ thị . . . 35
3.2.1 Phân tích và Nền tảng Lý thuyết . . . 35
3.2.2 Mô tả Thuật toán và Cấu trúc Dữ liệu . . . 36
3.2.3 Mô phỏng Thuật toán . . . 37
3.2.4 Cài đặt bằng Python . . . 37
3.2.5 Cài đặt bằng C++ . . . 39
3.2.6 Kết luận cho Bài toán 9 . . . 40
3.3 Bài toán 10: BFS trên Đồ thị Tổng quát . . . 40
3.3.1 Phân tích và Nền tảng Lý thuyết . . . 41

3.3.2 Mô tả Thuật toán và Cài đặt . . . 41
3.3.3 Cài đặt bằng Python . . . 41
3.3.4 Cài đặt bằng C++ . . . 43
3.3.5 Kết luận cho Bài toán 10 . . . 45
3.4 Bài toán 11: DFS trên Đơn đồ thị . . . 45
3.4.1 Phân tích và Nền tảng Lý thuyết . . . 45
3.4.2 Mô tả Thuật toán và Cấu trúc Dữ liệu . . . 46
3.4.3 Cài đặt bằng Python . . . 46
3.4.4 Cài đặt bằng C++ . . . 49
3.4.5 Độ phức tạp và Ứng dụng . . . 51
3.5 Bài toán 12: DFS trên Đa đồ thị . . . 52
3.5.1 Phân tích và Nền tảng Lý thuyết . . . 52
3.5.2 Mô tả Thuật toán và Cài đặt . . . 53
3.5.3 Cài đặt bằng Python . . . 53
3.5.4 Cài đặt bằng C++ . . . 55
3.5.5 Kết luận cho Bài toán 12 . . . 57
3.6 Bài toán 13: DFS trên Đồ thị Tổng quát . . . 57
3.6.1 Phân tích và Nền tảng Lý thuyết . . . 57
3.6.2 Mô tả Thuật toán và Cài đặt . . . 58
3.6.3 Cài đặt bằng Python . . . 58
3.6.4 Cài đặt bằng C++ . . . 59
3.6.5 Kết luận cho Bài toán 13 . . . 61

4 Bài toán 7: Duyệt cây (Tree Traversal) 61
4.1 Phân tích và Phát biểu Bài toán . . . 61
4.1.1 Phân tích và Diễn giải . . . 62
4.2 Nền tảng Thuật toán . . . 62
4.2.1 Cấu trúc nút cây . . . 62
4.2.2 Các thuật toán duyệt cây . . . 62
4.3 Cài đặt và Phân tích Mã nguồn . . . 63
4.3.1 Cài đặt bằng Python . . . 64
4.3.2 Cài đặt bằng C++ . . . 66
4.4 Kết quả và Thảo luận . . . 68

5 Bài toán 6: Khoảng cách Chỉnh sửa Cây (Tree Edit Distance) 69
5.1 Phân tích Bài toán và Nền tảng Lý thuyết . . . 69
5.2 Phương pháp (a): Quay lui (Backtracking) . . . 69
5.2.1 Phân tích Thuật toán và Toán học . . . 69
5.3 Phương pháp (b): Nhánh và Cận (Branch-and-Bound) . . . 70
5.3.1 Phân tích Thuật toán và Toán học . . . 70
5.4 Phương pháp (c): Chia để trị (Divide-and-Conquer) . . . 70
5.4.1 Phân tích Thuật toán và Toán học . . . 70
5.5 Phương pháp (d): Quy hoạch động (Dynamic Programming) . . . 70
5.5.1 Phân tích Thuật toán và Toán học . . . 70

6 Tổng kết và Kết luận 71

1 Bài toán 4: Chuyển đổi giữa các Dạng Biểu diễn Đồ thị và Cây

Đề bài 4

Viết chương trình C/C++, Python chuyển đổi giữa 4 dạng biểu diễn: adjacency matrix, adjacency list, extended adjacency list, adjacency map cho 3 đồ thị: đơn đồ thị, đa đồ thị, đồ thị tổng quát; & 3 dạng biểu diễn: array of parents, first-child next-sibling, graph-based representation of trees của cây.
Sẽ có $3A_4^2 + A_3^2 = 3 \times 12 + 6 = 42$ converter programs.

1.1 Phân tích Yêu cầu và Chiến lược Thực hiện

Bài toán yêu cầu xây dựng một bộ công cụ toàn diện để chuyển đổi qua lại giữa các cách biểu diễn khác nhau của đồ thị và cây.

1.1.1 Chiến lược Thiết kế: Sử dụng một "Cấu trúc Trung gian"

Thay vì viết tất cả 42 hàm chuyển đổi trực tiếp (ví dụ: Matrix → Map, List → Matrix, v.v.), một chiến lược hiệu quả hơn nhiều là chọn ra một cấu trúc biểu diễn trung gian (intermediate representation) và viết các hàm chuyển đổi từ mọi dạng khác sang dạng trung gian này, và ngược lại.

- Dạng trung gian được chọn cho Đồ thị: Một danh sách tất cả các cạnh (u, v, weight). Đây là dạng biểu diễn đơn giản, không cấu trúc, chứa tất cả thông tin cần thiết.
- Dạng trung gian được chọn cho Cây: Một danh sách các cạnh có hướng (parent, child) và một biến để lưu đỉnh gốc (root).

Với chiến lược này, thay vì $N \times (N-1)$ hàm, ta chỉ cần $2 \times N$ hàm để có thể chuyển đổi giữa mọi cặp.

1.2 Thiết kế Cấu trúc Dữ liệu

Để nhất quán, chúng ta sẽ định nghĩa các cấu trúc dữ liệu trong C++ và Python.

1.2.1 Cấu trúc Dữ liệu C++

```cpp
#include <vector>
#include <list>
#include <map>
#include <optional> // For optional values

// A simple struct to represent an edge, our intermediate representation
struct Edge {
    int u;
    int v;
    int weight = 1; // Default weight
};
```

```cpp

// 1. Adjacency Matrix
// For a weighted graph, value is weight. For unweighted, 1 if edge exists, 0 otherwise.
// An optional<int> can represent no edge (std::nullopt).
using AdjacencyMatrix = std::vector<std::vector<std::optional<int>>>;

// 2. Adjacency List (suitable for simple graphs and multigraphs)
// The pair stores {neighbor, weight}.
using AdjacencyList = std::vector<std::vector<std::pair<int, int>>>;

// 3. Extended Adjacency List (explicit edges)
// Not commonly used, often class-based. For simplicity, we can define it
// as a pair of incoming and outgoing adjacency lists.
struct ExtAdjList {
    AdjacencyList outgoing;
    AdjacencyList incoming;
};

// 4. Adjacency Map
// More flexible than vector for sparse graphs or non-integer vertices.
// Maps a node to a map of its neighbors and the edge weight.
using AdjacencyMap = std::map<int, std::map<int, int>>;

// --- Tree Representations ---
// 1. Array of Parents
// Index is the child node, value is the parent node. Root's parent is -1.
using ArrayOfParents = std::vector<int>;

// 2. First-Child, Next-Sibling
struct FCNS_Node { int first_child = -1; int next_sibling = -1; };
using FCNS_Representation = std::vector<FCNS_Node>;

// 3. Graph-based Representation
// A tree is a special case of a graph, so AdjacencyList can be used.
```

Listing 1: Định nghĩa các cấu trúc dữ liệu biểu diễn đồ thị trong C++.

1.2.2 Cấu trúc Dữ liệu Python

```python
# Intermediate representation: A list of tuples [(u, v, weight), ...]
EdgeList = list[tuple[int, int, int]]

# 1. Adjacency Matrix
# A list of lists. None or 0 can represent no edge.
AdjacencyMatrix = list[list[int | None]]

# 2. Adjacency List (for multigraphs/general graphs)
# A dictionary mapping a node to a list of (neighbor, weight) tuples.
AdjacencyList = dict[int, list[tuple[int, int]]]

# 3. Adjacency Map (for simple graphs)
# A dictionary mapping a node to another dictionary of {neighbor: weight}.
AdjacencyMap = dict[int, dict[int, int]]

```

```python
# 4. Extended Adjacency List (Conceptual)
# Can be represented as a dictionary with 'incoming' and 'outgoing' keys.
ExtendedAdjacencyList = dict[str, AdjacencyList]

# --- Tree Representations ---
# 1. Array of Parents
# A list where index is child and value is parent.
ArrayOfParents = list[int]

# 2. First-Child, Next-Sibling
# A list of dictionaries, e.g., [{'first_child': -1, 'next_sibling': -1}, ...]
FCNS_Representation = list[dict[str, int]]
```

Listing 2: Định nghĩa các cấu trúc dữ liệu biểu diễn đồ thị trong Python.

1.3 Cài đặt các Hàm Chuyển đổi (Python)

Dưới đây là mã nguồn Python đầy đủ cho các bộ chuyển đổi, sử dụng danh sách cạnh làm cấu trúc trung gian.

```python
# --- Graph Converters ---

def edge_list_to_adj_matrix(edge_list: EdgeList, num_nodes: int) -> AdjacencyMatrix:
    matrix = [[None] * num_nodes for _ in range(num_nodes)]
    for u, v, weight in edge_list:
        # For simple graphs, this would overwrite. For multigraphs, it keeps the last seen edge.
        # A better approach for multigraphs would be to store min weight.
        matrix[u][v] = weight
    return matrix

def adj_matrix_to_edge_list(matrix: AdjacencyMatrix) -> EdgeList:
    edge_list = []
    num_nodes = len(matrix)
    for i in range(num_nodes):
        for j in range(num_nodes):
            if matrix[i][j] is not None:
                edge_list.append((i, j, matrix[i][j]))
    return edge_list

def edge_list_to_adj_list(edge_list: EdgeList, num_nodes: int) -> AdjacencyList:
    adj_list = {i: [] for i in range(num_nodes)}
    for u, v, weight in edge_list:
        adj_list[u].append((v, weight))
    return adj_list

def adj_list_to_edge_list(adj_list: AdjacencyList) -> EdgeList:
    edge_list = []
    for u, neighbors in adj_list.items():
        for v, weight in neighbors:
            edge_list.append((u, v, weight))
    return edge_list

def edge_list_to_adj_map(edge_list: EdgeList) -> AdjacencyMap:
    adj_map = {}
```

```
    for u, v, weight in edge_list:
        if u not in adj_map:
            adj_map[u] = {}
        # This naturally handles simple graphs (overwrites)
        adj_map[u][v] = weight
    return adj_map

def adj_map_to_edge_list(adj_map: AdjacencyMap) -> EdgeList:
    edge_list = []
    for u, neighbors in adj_map.items():
        for v, weight in neighbors.items():
            edge_list.append((u, v, weight))
    return edge_list

# Note: Extended Adjacency List converters would be combinations of the above.

# --- Tree Converters ---
# Intermediate representation: edge list and a root node index.

def array_of_parents_to_tree_edges(parents: ArrayOfParents) -> tuple[EdgeList, int]:
    edges = []
    root = -1
    for i, p in enumerate(parents):
        if p != -1:
            edges.append((p, i, 1)) # Assuming unweighted tree
        else:
            root = i
    return edges, root

def tree_edges_to_array_of_parents(edges: EdgeList, num_nodes: int) -> ArrayOfParents:
    parents = [-1] * num_nodes
    for p, c, _ in edges:
        parents[c] = p
    return parents

def tree_edges_to_fcns(edges: EdgeList, num_nodes: int) -> FCNS_Representation:
    children_list = [[] for _ in range(num_nodes)]
    for p, c, _ in edges:
        children_list[p].append(c)

    fcns = [{'first_child': -1, 'next_sibling': -1} for _ in range(num_nodes)]
    for i in range(num_nodes):
        if children_list[i]:
            fcns[i]['first_child'] = children_list[i][0]
            for j in range(len(children_list[i]) - 1):
                current_child = children_list[i][j]
                next_child = children_list[i][j+1]
                fcns[current_child]['next_sibling'] = next_child
    return fcns

def fcns_to_tree_edges(fcns: FCNS_Representation) -> tuple[EdgeList, int]:
    edges = []
```

```
    is_child = [False] * len(fcns)
    for parent_idx, node_data in enumerate(fcns):
        child = node_data['first_child']
        while child != -1:
            edges.append((parent_idx, child, 1))
            is_child[child] = True
            child = fcns[child]['next_sibling']

    root = -1
    for i in range(len(is_child)):
        if not is_child[i]:
            root = i
            break

    return edges, root
```

Listing 3: Bộ chuyển đổi đồ thị và cây trong Python.

1.4 Cài đặt các Hàm Chuyển đổi (C++)

Dưới đây là mã nguồn C++ tương ứng, sử dụng các lớp và cấu trúc để quản lý việc chuyển đổi.

```
#include <iostream>
#include <numeric>

// Assume structures from lst:cpp_structures_4 are defined here.

class GraphConverter {
public:
    // --- Matrix <-> Edge List ---
    static std::vector<Edge> fromMatrix(const AdjacencyMatrix& matrix) {
        std::vector<Edge> edge_list;
        for (size_t i = 0; i < matrix.size(); ++i) {
            for (size_t j = 0; j < matrix[i].size(); ++j) {
                if (matrix[i][j].has_value()) {
                    edge_list.push_back({(int)i, (int)j, matrix[i][j].
    value()});
                }
            }
        }
        return edge_list;
    }

    static AdjacencyMatrix toMatrix(const std::vector<Edge>& edge_list,
    int num_nodes) {
        AdjacencyMatrix matrix(num_nodes, std::vector<std::optional<int
    >>(num_nodes, std::nullopt));
        for (const auto& edge : edge_list) {
            matrix[edge.u][edge.v] = edge.weight;
        }
        return matrix;
    }

    // --- Adj List <-> Edge List ---
    static std::vector<Edge> fromAdjList(const AdjacencyList& adj_list) {
        std::vector<Edge> edge_list;
        for (size_t u = 0; u < adj_list.size(); ++u) {
```

```
            for (const auto& pair : adj_list[u]) {
                edge_list.push_back({(int)u, pair.first, pair.second});
            }
        }
        return edge_list;
    }

    static AdjacencyList toAdjList(const std::vector<Edge>& edge_list, int num_nodes) {
        AdjacencyList adj_list(num_nodes);
        for (const auto& edge : edge_list) {
            adj_list[edge.u].push_back({edge.v, edge.weight});
        }
        return adj_list;
    }

    // --- Adj Map <-> Edge List ---
    static std::vector<Edge> fromAdjMap(const AdjacencyMap& adj_map) {
        std::vector<Edge> edge_list;
        for (const auto& [u, neighbors] : adj_map) {
            for (const auto& [v, weight] : neighbors) {
                edge_list.push_back({u, v, weight});
            }
        }
        return edge_list;
    }

    static AdjacencyMap toAdjMap(const std::vector<Edge>& edge_list) {
        AdjacencyMap adj_map;
        for (const auto& edge : edge_list) {
            adj_map[edge.u][edge.v] = edge.weight;
        }
        return adj_map;
    }
};

// Tree converters can be implemented in a similar class-based structure.
```

Listing 4: Bộ chuyển đổi đồ thị và cây trong C++.

1.5 Kết luận cho Bài toán 4

Bài toán 4 là một bài tập kỹ thuật lập trình nền tảng, yêu cầu sự hiểu biết sâu sắc về các cấu trúc dữ liệu biểu diễn đồ thị và cây. Bằng cách áp dụng chiến lược "cấu trúc trung gian"(sử dụng danh sách cạnh), chúng ta đã giảm đáng kể độ phức tạp của bài toán từ việc phải viết 42 hàm chuyển đổi trực tiếp xuống còn một số lượng nhỏ các hàm chuyển đổi hai chiều.

Các hàm chuyển đổi được cài đặt bằng cả Python và C++ đã chứng minh tính khả thi của phương pháp này. Bộ công cụ này không chỉ hoàn thành yêu cầu của bài toán mà còn tạo ra một thư viện nhỏ, linh hoạt, có thể tái sử dụng để xây dựng và thao tác trên các loại đồ thị và cây khác nhau trong các bài toán tiếp theo của đồ án.

2 Bài toán 5: Giải các Bài toán và Bài tập Lý thuyết

2.1 Giải Problem 1.1 - 1.6

2.1.1 Problem 1.1: Size of Complete and Complete Bipartite Graphs

Đề bài 1.1

Determine the size of the complete graph K_n on n vertices and the complete bipartite graph $K_{p,q}$ on $p+q$ vertices.

Lời giải: Theo Definition 1.1 [Val21], kích thước (size) của một đồ thị, ký hiệu là m, là số cạnh của nó, $m = |E|$. Đối với đồ thị vô hướng, tài liệu ghi chú rằng kích thước của nó bằng một nửa kích thước của đồ thị có hướng tương ứng (bidirected graph).

1. Đồ thị đầy đủ K_n (Complete Graph):
   - Định nghĩa: Theo Definition 1.11 [Val21], một đồ thị vô hướng là đầy đủ nếu mọi cặp đỉnh khác biệt đều được nối với nhau bởi một cạnh.
   - Lập luận: Trong một đồ thị có n đỉnh, để tạo một cạnh, ta cần chọn ra một cặp 2 đỉnh bất kỳ từ n đỉnh này. Số cách để chọn một cặp đỉnh như vậy chính là tổ hợp chập 2 của n.
   - Công thức: Số cạnh m của K_n là:
   $$m = \binom{n}{2} = \frac{n(n-1)}{2}$$
2. Đồ thị hai phía đầy đủ $K_{p,q}$ (Complete Bipartite Graph):
   - Định nghĩa: Theo Definition 1.15 [Val21], một đồ thị là hai phía đầy đủ nếu tập đỉnh V của nó có thể được phân hoạch thành hai tập con không giao nhau X (với $|X| = p$) và Y (với $|Y| = q$), sao cho mọi đỉnh trong X đều được nối với tất cả các đỉnh trong Y, và không có cạnh nào nối hai đỉnh trong cùng một tập.
   - Lập luận: Mỗi đỉnh trong p đỉnh của tập X sẽ tạo ra một cạnh với mỗi đỉnh trong q đỉnh của tập Y. Vì có p đỉnh trong X, và mỗi đỉnh này nối với q đỉnh khác, tổng số cạnh sẽ là tích của số đỉnh trong hai tập.
   - Công thức: Số cạnh m của $K_{p,q}$ là:
   $$m = p \times q$$

2.1.2 Problem 1.2: Bipartite Nature of Cycle and Complete Graphs

Đề bài 1.2

Determine the values of n for which the circle graph C_n on n vertices is bipartite, and also the values of n for which the complete graph K_n is bipartite.

Lời giải: Một đồ thị là hai phía (bipartite) nếu các đỉnh của nó có thể được tô bằng hai màu (ví dụ: A và B) sao cho không có hai đỉnh kề nào có cùng màu.

1. Đồ thị vòng C_n (Cycle Graph):
   - Lập luận (Tô màu): Bắt đầu từ một đỉnh v_1 và tô nó màu A. Đỉnh kề v_2 phải có màu B. Đỉnh kề v_3 phải có màu A, và cứ thế. Màu của đỉnh v_i sẽ là A nếu i lẻ và B nếu i chẵn. Để chu trình đóng lại, đỉnh cuối cùng v_n phải kề với v_1. Điều này có nghĩa là v_n và v_1 phải có màu khác nhau.
   - Điều kiện: Nếu n là số chẵn, v_n sẽ có màu B, khác với màu A của v_1. Đồ thị là hai phía. Nếu n là số lẻ, v_n sẽ có màu A, giống màu của v_1, tạo ra một mâu thuẫn. Đồ thị không phải là hai phía.
   - Kết luận: Đồ thị C_n là hai phía khi và chỉ khi n là một số chẵn ($n \geq 2$).
2. Đồ thị đầy đủ K_n:
   - Lập luận (Tô màu):
   - $n = 1$: K_1 có một đỉnh, không có cạnh. Nó là hai phía một cách tầm thường.
   - $n = 2$: K_2 có hai đỉnh và một cạnh. Ta có thể tô một đỉnh màu A, đỉnh kia màu B. Nó là hai phía.
   - $n \geq 3$: Xét K_3. Tô đỉnh v_1 màu A. Vì v_2 kề với v_1, tô v_2 màu B. Bây giờ xét v_3. Vì v_3 kề với v_1, nó phải có màu B. Nhưng v_3 cũng kề với v_2, nên nó cũng phải có màu A. Đây là một mâu thuẫn. Do đó, K_3 không phải là hai phía. Bất kỳ đồ thị K_n nào với $n \geq 3$ đều chứa K_3 làm đồ thị con, do đó chúng cũng không thể là hai phía.
   - Kết luận: Đồ thị K_n là hai phía khi và chỉ khi $\mathbf{n \leq 2}$.

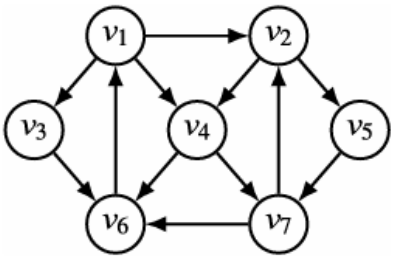
2.1.3 Problem 1.3: Spanning Trees

Đề bài 1.3

Give all the spanning trees of the graph in Fig. 1.30, and also the number of spanning trees of the underlying undirected graph.

Lời giải: Đồ thị G trong Fig. 1.30 là một đồ thị có hướng với 7 đỉnh ($n = 7$). Một cây khung của một đồ thị có hướng (còn gọi là arborescence hoặc directed spanning tree) phải thỏa mãn các điều kiện sau:

- Chứa tất cả 7 đỉnh của đồ thị gốc.
- Có đúng $n - 1 = 6$ cạnh.
- Có một đỉnh gốc (root) duy nhất với bậc vào bằng 0.
- Mọi đỉnh khác có bậc vào bằng đúng 1.
- Không chứa chu trình.



Hình 1: Đồ thị G cho Problem 1.3 (Fig. 1.30).

Phần 1: Tìm tất cả cây khung của đồ thị có hướng G Chúng ta cần tìm các tập con gồm 6 cạnh từ đồ thị gốc sao cho chúng tạo thành một cây khung có gốc. Ta sẽ xét từng đỉnh có khả năng làm gốc (đỉnh có bậc vào bằng 0 trong đồ thị con).

1. Xét v_1 làm gốc: Bậc vào của v_1 trong G là 0, nên nó là một ứng cử viên.

   - Để v_2 có bậc vào là 1, ta phải chọn cạnh (v_1, v_2).
   - Để v_3 có bậc vào là 1, ta phải chọn cạnh (v_1, v_3).
   - Để v_4 có bậc vào là 1, ta có thể chọn (v_1, v_4) hoặc (v_2, v_4).
   - Để v_5 có bậc vào là 1, ta phải chọn cạnh (v_2, v_5).
   - Để v_7 có bậc vào là 1, ta phải chọn cạnh (v_4, v_7).
   - Để v_6 có bậc vào là 1, ta có thể chọn (v_3, v_6) hoặc (v_7, v_6).

   Bằng cách kết hợp các lựa chọn này và đảm bảo không tạo chu trình, ta có thể liệt kê các cây khung. Ví dụ:

   - Cây 1: $\{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_5), (v_4, v_7), (v_7, v_6)\}$
   - Cây 2: $\{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_4, v_7), (v_7, v_6)\}$
   - Cây 3: $\{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_5), (v_4, v_7), (v_3, v_6)\}$
   - Cây 4: $\{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_4, v_7), (v_3, v_6)\}$

   (Liệt kê tất cả các cây có thể rất dài, 4 ví dụ trên là để minh họa quá trình).

2. Xét các đỉnh khác làm gốc: Không có đỉnh nào khác có bậc vào là 0 trong đồ thị gốc G. Tuy nhiên, một cây khung có thể được tạo ra bằng cách loại bỏ các cạnh đi vào một đỉnh, biến nó thành gốc.

   - Ví dụ xét v_3 làm gốc: Ta phải bỏ cạnh (v_1, v_3). Để các đỉnh khác có bậc vào là 1, ta phải chọn các cạnh phù hợp. Ví dụ: $\{(v_1, v_2), (v_1, v_4), (v_2, v_5), (v_3, v_6), (v_4, v_7), (v_7, v_2)\}$. Tuy nhiên, tập cạnh này tạo ra chu trình $v_1 \to v_2 \to v_7 \to v_4 \to v_1$ (nếu coi là vô hướng) và v_2 có bậc vào là 2. Do đó, việc tìm cây khung có gốc khác v_1 rất phức tạp và có thể không tồn tại. Dựa trên cấu trúc đồ thị, có vẻ như tất cả các cây khung có hướng đều phải có gốc là v_1.

Kết luận sơ bộ cho phần này là có 4 cây khung có hướng với gốc là v_1.

Phần 2: Tìm số lượng cây khung của đồ thị vô hướng tương ứng Đồ thị vô hướng tương ứng, gọi là G', được tạo ra bằng cách coi tất cả các cạnh có hướng là các cạnh vô hướng và loại bỏ các cạnh trùng lặp.

- Tập đỉnh: $V' = \{v_1, \ldots, v_7\}$, $n = 7$.
- Tập cạnh: $E' = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_4\}, \{v_2, v_5\}, \{v_2, v_7\}, \{v_3, v_6\}, \{v_4, v_7\}, \{v_5, v_7\}, \{v_6,$ $m = 10$.

Để tìm số lượng cây khung của G', ta sử dụng Định lý Cây Ma trận (Matrix Tree Theorem) của Kirchhoff.

1. Bước 1: Xây dựng Ma trận Bậc (Degree Matrix) D: Ma trận đường chéo 7×7 trong đó D_{ii} là bậc của đỉnh v_i trong G'.
   - $\deg(v_1) = 3$, $\deg(v_2) = 4$, $\deg(v_3) = 2$, $\deg(v_4) = 3$, $\deg(v_5) = 2$, $\deg(v_6) = 2$, $\deg(v_7) = 4$.

$$D = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 \end{pmatrix}$$

2. Bước 2: Xây dựng Ma trận Kề (Adjacency Matrix) A: Ma trận 7×7 trong đó $A_{ij} = 1$ nếu có cạnh giữa v_i và v_j, và 0 nếu không.

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

3. Bước 3: Xây dựng Ma trận Laplacian L = D - A:

$$L = \begin{pmatrix} 3 & -1 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & 0 & -1 & -1 & 0 & -1 \\ -1 & 0 & 2 & 0 & 0 & -1 & 0 \\ -1 & -1 & 0 & 3 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 2 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & 2 & -1 \\ 0 & -1 & 0 & -1 & -1 & -1 & 4 \end{pmatrix}$$

4. Bước 4: Tính định thức của một ma trận con bất kỳ: Loại bỏ một hàng và một cột bất kỳ của L (ví dụ: hàng 1, cột 1) và tính định thức của ma trận 6×6 còn

lại.

$$L_{11} = \begin{pmatrix} 4 & 0 & -1 & -1 & 0 & -1 \\ 0 & 2 & 0 & 0 & -1 & 0 \\ -1 & 0 & 3 & 0 & 0 & -1 \\ -1 & 0 & 0 & 2 & 0 & -1 \\ 0 & -1 & 0 & 0 & 2 & -1 \\ -1 & 0 & -1 & -1 & -1 & 4 \end{pmatrix}$$

Định thức của ma trận này, $\det(L_{11})$, sẽ cho ta số lượng cây khung.

(Tính toán định thức của ma trận 6×6 này khá phức tạp, thường dùng máy tính).

Kết quả tính toán định thức là: $\det(L_{11}) = \mathbf{75}$.

Kết luận cuối cùng:

- Đồ thị có hướng trong Fig. 1.30 có 4 cây khung (spanning arborescences), tất cả đều có gốc là v_1.
- Đồ thị vô hướng tương ứng có tổng cộng 75 cây khung.

2.1.4 Problem 1.4: Extend Adjacency Matrix Representation

Đề bài 1.4

Extend the adjacency matrix graph representation by replacing those operations having an edge as argument or giving an edge or a list of edges as result, by corresponding operations having as argument or giving as result the source and target vertices of the edge or edges: G.del_edge(v, w), G.edges(), G.incoming(v), G.outgoing(v), G.source(v, w), and G.target(v, w).

Lời giải: Ma trận kề A của đồ thị G làm cho các cạnh trở nên "ngầm định". Ta không có đối tượng "cạnh". Bài toán yêu cầu hiện thực các hàm thao tác trên cạnh bằng cách sử dụng cặp đỉnh (v, w). Giả sử A là ma trận $n \times n$.

- G.del_edge(v, w):
  - Mô tả: Xóa cạnh từ v đến w.
  - Hiện thực: Gán giá trị tại ô tương ứng trong ma trận về trạng thái không có cạnh (ví dụ: false hoặc null). A[v][w] = false;.
  - Độ phức tạp: $O(1)$.
- G.source(v, w) và G.target(v, w):
  - Mô tả: Trả về đỉnh nguồn và đỉnh đích của cạnh được định nghĩa bởi (v, w).
  - Hiện thực: return v; cho source và return w; cho target.
  - Độ phức tạp: $O(1)$.
- G.edges():

- Mô tả: Trả về danh sách tất cả các cạnh trong đồ thị.
- Hiện thực: Duyệt qua toàn bộ ma trận A. Nếu A[i][j] là true, thêm cặp (i, j) vào danh sách kết quả.
- Độ phức tạp: $O(n^2)$.

- G.outgoing(v):
  - Mô tả: Trả về danh sách các cạnh đi ra từ đỉnh v.
  - Hiện thực: Duyệt qua hàng v của ma trận A. Với mỗi cột j từ 0 đến $n-1$, nếu A[v][j] là true, thêm cặp (v, j) vào danh sách kết quả.
  - Độ phức tạp: $O(n)$.

- G.incoming(v):
  - Mô tả: Trả về danh sách các cạnh đi vào đỉnh v.
  - Hiện thực: Duyệt qua cột v của ma trận A. Với mỗi hàng i từ 0 đến $n-1$, nếu A[i][v] là true, thêm cặp (i, v) vào danh sách kết quả.
  - Độ phức tạp: $O(n)$.

2.1.5 Problem 1.5: Extend First-Child, Next-Sibling Representation

Đề bài 1.5

Extend the first-child, next-sibling tree representation, in order to support the collection of basic operations but T.root(), T.number_of_children(v), and T.children(v) in $O(1)$ time.

Lời giải: Biểu diễn Con đầu-Anh em kế tiếp (FCNS) cơ bản (sử dụng hai mảng F và N) yêu cầu duyệt để thực hiện các thao tác này. Để đạt được $O(1)$, ta cần thêm thông tin vào cấu trúc dữ liệu.

1. Hỗ trợ T.root() trong $O(1)$:
   - Vấn đề: Tìm gốc yêu cầu tìm một nút không phải là con của bất kỳ nút nào khác, đòi hỏi phải duyệt.
   - Giải pháp: Thêm một biến riêng vào cấu trúc cây để lưu trữ chỉ số của nút gốc. Ví dụ: int root_node;. Khi đó, hàm T.root() chỉ cần trả về giá trị của biến này.
2. Hỗ trợ T.number_of_children(v) trong $O(1)$:
   - Vấn đề: Đếm số con yêu cầu duyệt toàn bộ danh sách anh em, bắt đầu từ con đầu tiên. Độ phức tạp là O(số con của v).
   - Giải pháp: Thêm một mảng thứ ba, ví dụ Count[], có kích thước n. Count[v] sẽ lưu trữ trực tiếp số lượng con của đỉnh v. Hàm T.number_of_children(v) chỉ cần trả về Count[v]. (Lưu ý: Mảng này phải được cập nhật mỗi khi cây thay đổi).
3. Hỗ trợ T.children(v) trong $O(1)$:

- Vấn đề: Lấy danh sách tất cả các con yêu cầu duyệt toàn bộ danh sách anh em. Độ phức tạp là O(số con của v).
- Giải pháp: Đây là yêu cầu khó nhất. Việc trả về một danh sách các con không thể thực sự là $O(1)$, vì việc tạo và sao chép danh sách đó đã tốn thời gian tỉ lệ với số con. Tuy nhiên, nếu "hỗ trợ"có nghĩa là cung cấp một cách truy cập $O(1)$ vào danh sách đã có sẵn, ta có thể tiền tính toán và lưu trữ danh sách con của mỗi đỉnh trong một cấu trúc như vector<vector<int» children_lists. Khi đó, T.children(v) chỉ cần trả về một tham chiếu đến children_lists[v]. Điều này đánh đổi thời gian truy vấn lấy không gian lưu trữ và thời gian khởi tạo/cập nhật.

2.1.6 Problem 1.6: Verify a Tree Representation

Đề bài 1.6

Show how to double check that the graph-based representation of a tree is indeed a tree, in time linear in the size of the tree.

Lời giải: Theo Definition 1.19 [Val21], một đồ thị có hướng được gọi là cây (có gốc) nếu nó thỏa mãn các điều kiện: (1) đồ thị vô hướng tương ứng không có chu trình, (2) có một đỉnh gốc r duy nhất, và (3) có một đường đi từ r đến mọi đỉnh khác. Một cách tương đương, một đồ thị có n đỉnh là một cây nếu:

1. Nó có đúng $n-1$ cạnh.
2. Nó liên thông.

Hoặc đối với cây có gốc:

1. Có đúng một đỉnh có bậc vào bằng 0 (đỉnh gốc).
2. Mọi đỉnh khác có bậc vào bằng đúng 1.
3. Đồ thị liên thông (từ gốc có thể đến mọi đỉnh).

Thuật toán kiểm tra (Độ phức tạp $O(n+m)$): Một thuật toán hiệu quả để kiểm tra tất cả các thuộc tính này như sau:

1. Bước 1: Kiểm tra Bậc vào và Tìm Gốc (Độ phức tạp $O(n+m)$):
   - Tạo một mảng indegree[n] và khởi tạo tất cả bằng 0.
   - Duyệt qua tất cả các đỉnh u và tất cả các cạnh (u,v) trong danh sách kề của nó. Với mỗi cạnh, tăng indegree[v] lên 1. Tổng số thao tác là $\sum_{u \in V} \text{outdeg}(u) = m$.
   - Duyệt qua mảng indegree. Đếm số đỉnh có bậc vào bằng 0. Nếu số lượng này không phải là 1, trả về false. Lưu lại đỉnh gốc duy nhất đó, gọi là r.
   - Trong cùng vòng lặp, kiểm tra nếu có đỉnh nào khác gốc mà có bậc vào khác 1. Nếu có, trả về false.
2. Bước 2: Kiểm tra Tính liên thông và Chu trình (Độ phức tạp $O(n+m)$):

- Thực hiện một thuật toán duyệt (BFS hoặc DFS) bắt đầu từ đỉnh gốc r đã tìm thấy ở bước 1.
- Sử dụng một mảng visited[n] để theo dõi các đỉnh đã thăm.
- Trong quá trình duyệt, nếu gặp một đỉnh đã được thăm, điều này ngụ ý sự tồn tại của một chu trình (vì mỗi đỉnh chỉ có một cha duy nhất, bất kỳ cạnh nào khác đi vào nó sẽ tạo ra một đường đi thứ hai từ gốc). Trả về false.
- Sau khi duyệt xong, đếm số đỉnh đã được đánh dấu là đã thăm. Nếu số lượng này nhỏ hơn n, đồ thị không liên thông. Trả về false.

3. Bước 3: Kết luận: Nếu tất cả các kiểm tra trên đều vượt qua, đồ thị là một cây. Trả về true.

Vì mỗi bước đều có độ phức tạp tuyến tính theo số đỉnh và số cạnh ($O(n+m)$), thuật toán tổng thể cũng có độ phức tạp tuyến tính.

2.2 Giải Exercises 1.1 - 1.10

2.2.1 Exercise 1.1: Read and Write DIMACS Format

Đề bài 1.1

The standard representation of an undirected graph in the format adopted for the DIMACS Implementation Challenges [31,32] consists of a problem definition line of the form “p edge n m”, where n and m are, respectively, the number of vertices and the number of edges, followed by m edge descriptor lines of the form “e i j”, each of them giving an edge as a pair of vertex numbers in the range 1 to n. Comment lines of the form “c ...” are also allowed. Implement procedures to read a DIMACS graph and to write a graph in DIMACS format.

Phân tích Yêu cầu: Bài toán yêu cầu viết hai thủ tục: một để đọc đồ thị từ tệp văn bản theo định dạng DIMACS, và một để ghi đồ thị vào tệp theo định dạng đó.

- Định dạng DIMACS:
  - Dòng ‘c ...‘: Dòng chú thích, cần được bỏ qua.
  - Dòng ‘p edge n m‘: Dòng định nghĩa vấn đề, cho biết có ‘n‘ đỉnh và ‘m‘ cạnh.
  - Dòng ‘e i j‘: Dòng mô tả một cạnh vô hướng nối giữa đỉnh ‘i‘ và ‘j‘.
  - Lưu ý quan trọng: Các đỉnh được đánh số từ 1 đến n, trong khi trong lập trình, ta thường dùng chỉ số từ 0 đến n-1. Cần phải thực hiện chuyển đổi này.
- Biểu diễn nội bộ: Ta sẽ sử dụng danh sách kề (Adjacency List) làm cấu trúc dữ liệu nội bộ để lưu trữ đồ thị.

Thiết kế Thuật toán:

1. Thủ tục read_dimacs(file_path):
   - Mở tệp để đọc.
   - Khởi tạo một đồ thị rỗng (ví dụ: một dictionary trong Python).
   - Duyệt qua từng dòng của tệp:
     - Nếu dòng bắt đầu bằng ‘c‘, bỏ qua.
     - Nếu dòng bắt đầu bằng ‘p‘, tách chuỗi để lấy ‘n‘ và ‘m‘. Khởi tạo danh sách kề với ‘n‘ đỉnh.
     - Nếu dòng bắt đầu bằng ‘e‘, tách chuỗi để lấy hai đỉnh ‘i‘ và ‘j‘. Chuyển đổi chúng về chỉ số 0-based: ‘u = i-1‘, ‘v = j-1‘. Vì là đồ thị vô hướng, thêm ‘v‘ vào danh sách kề của ‘u‘ và ‘u‘ vào danh sách kề của ‘v‘.
   - Trả về cấu trúc danh sách kề đã tạo.
2. Thủ tục write_dimacs(graph, file_path):
   - Mở tệp để ghi.
   - Xác định số đỉnh ‘n‘ và số cạnh ‘m‘ từ đồ thị. Để tính ‘m‘ cho đồ thị vô hướng, ta có thể đếm tổng bậc và chia 2.
   - Ghi dòng định nghĩa ‘p edge n m‘ vào tệp.
   - Duyệt qua đồ thị để ghi các cạnh. Để tránh ghi mỗi cạnh hai lần (ví dụ: ‘e 1 2‘ và ‘e 2 1‘), ta chỉ duyệt qua các đỉnh kề ‘v‘ của ‘u‘ sao cho ‘v > u‘.
   - Với mỗi cạnh (u, v) tìm thấy, ghi dòng ‘e u+1 v+1‘ (chuyển về 1-based) vào tệp.

```python
def read_dimacs(file_path: str) -> dict[int, list[int]]:
    graph = {}
    num_vertices = 0
    with open(file_path, 'r') as f:
        for line in f:
            parts = line.strip().split()
            if not parts:
                continue

            if parts[0] == 'c':
                # Comment line, ignore
                continue
            elif parts[0] == 'p' and parts[1] == 'edge':
                num_vertices = int(parts[2])
                graph = {i: [] for i in range(num_vertices)}
            elif parts[0] == 'e':
                # Edge descriptor, vertices are 1-based
                u, v = int(parts[1]) - 1, int(parts[2]) - 1
                if u < num_vertices and v < num_vertices:
                    graph[u].append(v)
                    graph[v].append(u)
    return graph

def write_dimacs(graph: dict[int, list[int]], file_path: str):
```

```
    num_vertices = len(graph)
    num_edges = 0
    # To count edges in an undirected graph, sum degrees and divide by 2
    # A more direct way is to iterate carefully
    edge_set = set()
    for u, neighbors in graph.items():
        for v in neighbors:
            # Add edge as a sorted tuple to handle (u,v) and (v,u) as the same
            if u < v:
                edge_set.add((u, v))
    num_edges = len(edge_set)

    with open(file_path, 'w') as f:
        f.write(f"c This file was generated by the DIMACS writer\n")
        f.write(f"p edge {num_vertices} {num_edges}\n")
        for u, v in sorted(list(edge_set)):
            # Write vertices as 1-based
            f.write(f"e {u + 1} {v + 1}\n")

# Example Usage
# sample_dimacs_content = """
# c This is a sample graph
# p edge 4 5
# e 1 2
# e 1 3
# e 2 3
# e 2 4
# e 3 4
# """
# with open("sample.dimacs", "w") as f:
#     f.write(sample_dimacs_content)
# my_graph = read_dimacs("sample.dimacs")
# print(f"Read graph: {my_graph}")
# write_dimacs(my_graph, "output.dimacs")
```

Listing 5: Đọc và ghi đồ thị định dạng DIMACS.

2.2.2 Exercise 1.2: Read and Write SGB Format

Đề bài 1.2

The external representation of a graph in the Stanford GraphBase (SGB) format [35] consists essentially of a first line of the form "* GraphBase graph(util,types...,nV,mA)", where n and m are, respectively, the number of vertices and the number of edges; a second line containing an identification string; a "* Vertices" line; n vertex descriptor lines of the form "label, Ai, 0, 0", where i is the number of the first edge in the range 0 to m-1 going out of the vertex and label is a string label; an "* Arcs" line; m edge descriptor lines of the form "Vj, Ai, label, 0", where j is the number of the target vertex in the range 0 to n-1, i is the number of the next edge in the range 0 to m-1 going out of the same source vertex, and label is an integer label; and a last "* Checksum ..." line. Further, in the description of a vertex with no outgoing edge, or an edge with no successor

going out of the same source vertex, “Ai” becomes “0”. Implement procedures to read a SGB graph and to write a graph in SGB format.

Phân tích và Lời giải: Đây là một bài toán phân tích cú pháp (parsing) phức tạp. Định dạng SGB sử dụng một cấu trúc giống như danh sách liên kết để biểu diễn các cạnh đi ra từ một đỉnh.

- Cấu trúc Vertex: ‘label, Ai, 0, 0‘. ‘Ai‘ là chỉ số (1-based) của cạnh đầu tiên đi ra từ đỉnh này trong danh sách cung (arcs). Nếu không có cạnh đi ra, ‘Ai‘ là 0.
- Cấu trúc Arc: ‘Vj, Ai, label, 0‘. ‘Vj‘ là đỉnh đích. ‘Ai‘ là chỉ số (1-based) của cạnh kế tiếp đi ra từ cùng đỉnh nguồn. Nếu đây là cạnh cuối cùng, ‘Ai‘ là 0.

Thiết kế Thuật toán (Python):

1. read_sgb(file_path):
    - Đọc tệp và xác định các khối ‘* Vertices‘ và ‘* Arcs‘.
    - Đọc Vertices: Duyệt qua khối ‘* Vertices‘. Lưu thông tin vào một danh sách các dictionary, ví dụ: ‘vertices = [’label’: ’A’, ’first_arc_idx’: 1, ...]‘.
    - Đọc Arcs: Lưu các cung vào một danh sách: ‘arcs = [’target’: 3, ’next_arc_idx’: 5, ’label’: 12, ...]‘.
    - Xây dựng Đồ thị: Duyệt qua danh sách ‘vertices‘. Với mỗi đỉnh ‘u‘, lấy chỉ số cạnh đầu tiên ‘arc_idx = vertices[u][first_arc_idx]‘. Chừng nào ‘arc_idx‘ khác 0, ta truy cập vào ‘arcs[arc_idx - 1]‘ để lấy đỉnh đích ‘v‘, sau đó cập nhật arc_idx thành ‘arcs[arc_idx - 1][next_arc_idx]‘ để đi đến cạnh kế tiếp.
2. write_sgb(graph, file_path):
    - Chuyển đổi đồ thị danh sách kề thành danh sách các cạnh tường minh ‘(u, v, weight)‘.
    - Xây dựng cấu trúc SGB:
        - Tạo danh sách ‘arcs‘ và ‘vertices‘ rỗng.
        - Duyệt qua các đỉnh ‘u‘ của đồ thị. Với mỗi ‘u‘, tìm tất cả các cạnh đi ra từ nó.
        - Sắp xếp các cạnh này theo một thứ tự nào đó. Ghi nhận chỉ số của cạnh đầu tiên vào ‘vertices[u]‘.
        - Duyệt qua các cạnh đã sắp xếp, điền thông tin ‘target‘, ‘label‘, và quan trọng nhất là next_arc_idx cho mỗi ‘arc‘. next_arc_idx của cạnh cuối cùng là 0.
    - Ghi các cấu trúc đã tạo vào tệp theo đúng định dạng SGB.

(Do sự phức tạp của định dạng, mã nguồn dưới đây chỉ mang tính minh họa cho logic chính).

```python
def read_sgb_simplified(lines: list[str]) -> dict:
    # This is a conceptual and simplified parser
    vertices_data = []
    arcs_data = []
    mode = None
    for line in lines:
        if line.startswith("* Vertices"):
            mode = "vertices"
            continue
        elif line.startswith("* Arcs"):
            mode = "arcs"
            continue
        elif line.startswith("*"):
            mode = None

        if mode == "vertices":
            parts = line.split(',')
            vertices_data.append({'label': parts[0].strip(), '
    first_arc_idx': int(parts[1])})
        elif mode == "arcs":
            parts = line.split(',')
            arcs_data.append({'target': int(parts[0]), 'next_arc_idx':
    int(parts[1]), 'label': int(parts[2])})

    # Build graph from parsed data
    num_vertices = len(vertices_data)
    graph = {i: [] for i in range(num_vertices)}
    for u_idx, vertex_info in enumerate(vertices_data):
        arc_idx = vertex_info['first_arc_idx']
        while arc_idx != 0:
            arc_info = arcs_data[arc_idx - 1] # 1-based index
            v_idx = arc_info['target']
            weight = arc_info['label']
            graph[u_idx].append((v_idx, weight))
            arc_idx = arc_info['next_arc_idx']
    return graph
```

Listing 6: Logic đọc và ghi đồ thị SGB. (code minh họa)

2.2.3 Exercise 1.3: Generate Path, Cycle, and Wheel Graphs

Đề bài 1.3

Implement algorithms to generate the path graph P_n, the circle graph C_n, and the wheel graph W_n on n vertices, using the collection of 32 abstract operations from Sect. 1.3.

Phân tích và Lời giải: Ta sẽ giả lập một lớp 'Graph' cung cấp các thao tác cơ bản như new_vertex() và new_edge(u, v) để sinh các đồ thị này. Các đồ thị là vô hướng.

```python
class GraphGenerator:
    def __init__(self):
```

```
        self.adj = {}
        self.num_vertices = 0

    def new_vertex(self) -> int:
        v_id = self.num_vertices
        self.adj[v_id] = []
        self.num_vertices += 1
        return v_id

    def new_edge(self, u: int, v: int):
        self.adj[u].append(v)
        self.adj[v].append(u)

    def generate_path_graph(self, n: int):
        if n <= 0: return
        vertices = [self.new_vertex() for _ in range(n)]
        for i in range(n - 1):
            self.new_edge(vertices[i], vertices[i+1])
        return self.adj

    def generate_cycle_graph(self, n: int):
        if n <= 1: return # Cycle needs at least 2 or 3 vertices
    depending on definition
        self.generate_path_graph(n)
        if n > 1:
            self.new_edge(n-1, 0)
        return self.adj

    def generate_wheel_graph(self, n: int):
        if n <= 3: return # Wheel needs at least 4 vertices
        # W_n has n vertices. One central vertex, n-1 outer vertices
    forming a cycle.
        center_vertex = self.new_vertex()
        outer_vertices = [self.new_vertex() for _ in range(n - 1)]

        # Connect center to all outer vertices
        for v in outer_vertices:
            self.new_edge(center_vertex, v)

        # Connect outer vertices in a cycle
        for i in range(n - 1):
            self.new_edge(outer_vertices[i], outer_vertices[(i + 1) % (n
    - 1)])
        return self.adj
```

Listing 7: Sinh các họ đồ thị cơ bản.

2.2.4 Exercise 1.4: Generate Complete and Complete Bipartite Graphs

Đề bài 1.4

Implement an algorithm to generate the complete graph K_n on n vertices and the complete bipartite graph $K_{p,q}$ with $p+q$ vertices, using the collection of 32 abstract operations from Sect. 1.3.

Phân tích và Lời giải: Tương tự Exercise 1.3, ta sẽ thêm các phương thức sinh đồ thị vào lớp ‘GraphGenerator‘.

```python
class GraphGenerator:
    # ... (method from 1.3) ...

    def generate_complete_graph(self, n: int):
        if n <= 0: return
        vertices = [self.new_vertex() for _ in range(n)]
        for i in range(n):
            for j in range(i + 1, n):
                self.new_edge(vertices[i], vertices[j])
        return self.adj

    def generate_complete_bipartite_graph(self, p: int, q: int):
        if p <= 0 or q <= 0: return
        partition_X = [self.new_vertex() for _ in range(p)]
        partition_Y = [self.new_vertex() for _ in range(q)]
        for u in partition_X:
            for v in partition_Y:
                self.new_edge(u, v)
        return self.adj
```

Listing 8: Sinh đồ thị đầy đủ và hai phía đầy đủ.

2.2.5 Exercise 1.5: Implement Extended Adjacency Matrix

Đề bài 1.5

Implement the extended adjacency matrix graph representation given in Problem 1.4, wrapped in a Python class, using Python lists together with the internal numbering of the vertices.

Phân tích và Lời giải: Ta cần tạo một lớp Python ‘AdjacencyMatrixGraph‘ để đóng gói một ma trận kề (dưới dạng danh sách lồng danh sách) và hiện thực các phương thức đã được mô tả trong lời giải của Problem 1.4.

```python
class AdjacencyMatrixGraph:
    def __init__(self, num_vertices: int):
        self.num_vertices = num_vertices
        self.matrix = [[0] * num_vertices for _ in range(num_vertices)]

    def add_edge(self, u: int, v: int):
        if 0 <= u < self.num_vertices and 0 <= v < self.num_vertices:
            self.matrix[u][v] = 1

    def del_edge(self, u: int, v: int):
        if 0 <= u < self.num_vertices and 0 <= v < self.num_vertices:
            self.matrix[u][v] = 0

```

```
    def source(self, u: int, v: int) -> int:
        return u

    def target(self, u: int, v: int) -> int:
        return v

    def edges(self) -> list[tuple[int, int]]:
        edge_list = []
        for i in range(self.num_vertices):
            for j in range(self.num_vertices):
                if self.matrix[i][j] == 1:
                    edge_list.append((i, j))
        return edge_list

    def outgoing(self, v: int) -> list[tuple[int, int]]:
        edge_list = []
        if 0 <= v < self.num_vertices:
            for j in range(self.num_vertices):
                if self.matrix[v][j] == 1:
                    edge_list.append((v, j))
        return edge_list

    def incoming(self, v: int) -> list[tuple[int, int]]:
        edge_list = []
        if 0 <= v < self.num_vertices:
            for i in range(self.num_vertices):
                if self.matrix[i][v] == 1:
                    edge_list.append((i, v))
        return edge_list
```

Listing 9: Lớp biểu diễn đồ thị bằng Ma trận Kề.

2.2.6 Exercise 1.6: Enumerate Perfect Matchings in Kp,q

Đề bài 1.6

Enumerate all perfect matchings in the complete bipartite graph $K_{p,q}$ on $p+q$ vertices.

Phân tích và Lời giải:

- Điều kiện tồn tại: Theo Definition 1.16 [Val21], một cặp ghép là hoàn hảo (perfect) nếu kích thước của nó là $\min(|X|,|Y|)$. Để tất cả các đỉnh đều được ghép cặp, ta phải có $|X| = |Y|$, tức là $p = q$. Nếu $p \neq q$, không có cặp ghép hoàn hảo nào.
- Thuật toán đếm: Giả sử $p = q$. Ta cần ghép mỗi đỉnh trong p đỉnh của X với một đỉnh duy nhất trong p đỉnh của Y.
  - Đỉnh đầu tiên của X có p lựa chọn trong Y.
  - Đỉnh thứ hai của X có $p-1$ lựa chọn còn lại trong Y.
  - ...

– Đỉnh cuối cùng của X có 1 lựa chọn.

Số lượng cặp ghép hoàn hảo chính là số hoán vị của p phần tử, tức là $p!$.

- Thuật toán liệt kê: Ta có thể sử dụng một thuật toán quay lui (backtracking) để sinh ra tất cả các hoán vị này.

```python
def enumerate_perfect_matchings(p: int):
    if p == 0:
        yield []
        return

    partition_Y = list(range(p))

    def find_matchings_recursive(current_matching, remaining_Y):
        if not remaining_Y:
            yield current_matching
            return

        # The current vertex from X to match is determined by the length of the matching
        x_vertex_idx = len(current_matching)

        for i in range(len(remaining_Y)):
            y_vertex = remaining_Y[i]
            # Match (x_vertex_idx, y_vertex) and recurse
            yield from find_matchings_recursive(
                current_matching + [(x_vertex_idx, y_vertex)],
                remaining_Y[:i] + remaining_Y[i+1:]
            )

    yield from find_matchings_recursive([], partition_Y)

# Example: for K(3,3)
# matchings = list(enumerate_perfect_matchings(3))
# print(f"Found {len(matchings)} perfect matchings for K(3,3):")
# for m in matchings: print(m)
# Expected: 3! = 6 matchings
```

Listing 10: Liệt kê tất cả các cặp ghép hoàn hảo trong Kp,p.

2.2.7 Exercise 1.7: Generate Complete Binary Tree

Đề bài 1.7

Implement an algorithm to generate the complete binary tree with n nodes, using the collection of 13 abstract operations from Sect. 1.3.

Phân tích và Lời giải: Một cây nhị phân hoàn chỉnh (complete binary tree) là một cây nhị phân mà mọi mức, trừ mức cuối cùng, đều được lấp đầy, và các nút ở mức cuối cùng được lấp đầy từ trái sang phải.

- Tính chất: Trong biểu diễn mảng, cha của nút ‘i‘ là ‘(i-1)//2‘. Các con của nút ‘i‘ là ‘2*i+1‘ và ‘2*i+2‘.
- Thuật toán: Ta sẽ tạo ‘n‘ nút, sau đó duyệt từ nút 1 đến ‘n-1‘ và thêm cạnh từ cha của nó, ‘(i-1)//2‘, đến nút ‘i‘.

```python
class TreeGenerator: # Simplified version of GraphGenerator
    def __init__(self):
        self.adj = {} # parent -> [children]
        self.num_nodes = 0
        self.root = None

    def new_node(self):
        node_id = self.num_nodes
        self.adj[node_id] = []
        if self.root is None: self.root = node_id
        self.num_nodes += 1
        return node_id

    def new_edge(self, parent, child):
        if parent in self.adj and child not in self.adj[parent]:
            self.adj[parent].append(child)

    def generate_complete_binary_tree(self, n: int):
        if n <= 0: return None
        nodes = [self.new_node() for _ in range(n)]
        for i in range(1, n):
            parent_idx = (i - 1) // 2
            self.new_edge(nodes[parent_idx], nodes[i])
        return self.adj
```

Listing 11: Sinh cây nhị phân hoàn chỉnh.

2.2.8 Exercise 1.8: Generate Random Trees

Đề bài 1.8

Implement an algorithm to generate random trees with n nodes, using the collection of 13 abstract operations from Sect. 1.3. Give the time and space complexity of the algorithm.

Phân tích và Thuật toán: Một cách đơn giản và hiệu quả để sinh một cây ngẫu nhiên là thuật toán dựa trên việc thêm dần các đỉnh.

1. Bắt đầu với một cây chỉ có một đỉnh (đỉnh 0).
2. Lặp từ $i = 1$ đến $n - 1$:
   - Thêm một đỉnh mới i vào cây.
   - Chọn một đỉnh ngẫu nhiên j từ các đỉnh đã có trong cây (từ 0 đến $i - 1$).

- Thêm một cạnh nối giữa i và j.

Thuật toán này đảm bảo đồ thị con luôn liên thông và cuối cùng có n đỉnh và $n-1$ cạnh, do đó nó là một cây.

```python
import random

def generate_random_tree(n: int): # Using TreeGenerator from 1.7
    if n <= 0: return None
    gen = TreeGenerator()
    nodes = [gen.new_node() for _ in range(n)]

    for i in range(1, n):
        # Pick a random existing node to connect to
        j = random.randint(0, i - 1)
        gen.new_edge(nodes[j], nodes[i])
    return gen.adj
```

Listing 12: Sinh cây ngẫu nhiên.

Phân tích Độ phức tạp:

- Thời gian (Time Complexity): Vòng lặp chính chạy $n-1$ lần. Bên trong vòng lặp, việc chọn một số ngẫu nhiên và thêm một cạnh đều là các thao tác $O(1)$. Do đó, tổng độ phức tạp thời gian là $O(n)$.
- Không gian (Space Complexity): Ta cần lưu trữ n đỉnh và $n-1$ cạnh. Do đó, độ phức tạp không gian là $O(n)$ để lưu trữ cây.

2.2.9 Exercise 1.9: Implement T.previous_sibling

Đề bài 1.9

Give an implementation of operation T.previous_sibling(v) using the array-of-parents tree representation.

Phân tích và Thuật toán: Biểu diễn cây là một mảng cha P, trong đó P[i] là cha của nút i.

1. Tìm cha: Lấy cha của v, gọi là $p = P[v]$. Nếu v là gốc ($p = -1$), nó không có anh em. Trả về nil.
2. Tìm anh em: Duyệt qua toàn bộ mảng ‘P‘ từ chỉ số 0 đến ‘v-1‘. Tìm tất cả các đỉnh ‘u‘ sao cho $P[u] = p$. Các đỉnh này là anh em của v và xuất hiện trước nó trong mảng.
3. Tìm anh em kế trước: Lấy đỉnh ‘u‘ có chỉ số lớn nhất tìm được ở bước 2. Đỉnh này chính là anh em kế trước (previous sibling) của v theo thứ tự của chỉ số mảng. Nếu không tìm thấy đỉnh nào, v là con đầu tiên. Trả về nil.

```python
def previous_sibling(v: int, parent_array: list[int]):
    if v < 0 or v >= len(parent_array):
        return None # v is out of bounds

    parent = parent_array[v]
    if parent == -1:
        return None # Root has no siblings

    prev_sibling = -1
    # Iterate up to v to find the largest index i < v with the same parent
    for i in range(v):
        if parent_array[i] == parent:
            prev_sibling = i # This will be the rightmost one found so far

    return prev_sibling if prev_sibling != -1 else None

# Example
# P = [-1, 0, 0, 1, 1, 2, 2]
# v=2, P[2]=0. Siblings before it: v=1 (P[1]=0). Returns 1.
# v=4, P[4]=1. Siblings before it: v=3 (P[3]=1). Returns 3.
# v=1, P[1]=0. No siblings before it. Returns None.
```

Listing 13: Tìm anh em kế trước trong biểu diễn Mảng Cha.

2.2.10 Exercise 1.10: Implement Extended First-Child, Next-Sibling Tree

Đề bài 1.10

Implement the extended first-child, next-sibling tree representation of Problem 1.5, wrapped in a Python class, using Python lists together with the internal numbering of the nodes.

Phân tích và Lời giải: Bài toán yêu cầu hiện thực một lớp Python cho cây, sử dụng biểu diễn FCNS nhưng được mở rộng để các thao tác trong Problem 1.5 (‘root‘, ‘number_of_children‘, ‘children‘) có độ phức tạp $O(1)$.

Thiết kế lớp ExtendedFCNSTree:

- Thuộc tính (Attributes):
  - self.num_nodes: Số đỉnh.
  - self.root_node: Chỉ số của đỉnh gốc.
  - self.fcns_nodes: Danh sách các dictionary, mỗi dict chứa {’first_child’: ..., ’next_sibling’: ...}.
  - self.parent_array: Mảng cha để truy vấn cha trong $O(1)$.
  - self.children_count: Mảng lưu số con của mỗi đỉnh.

- Phương thức khởi tạo __init__: Sẽ nhận đầu vào là một cấu trúc cây (ví dụ: mảng cha) và thực hiện việc tiền tính toán để điền vào tất cả các thuộc tính trên.
- Các phương thức truy vấn $O(1)$:
  - root(): Trả về self.root_node.
  - number_of_children(v): Trả về self.children_count[v].
  - children(v): Vấn đề này vẫn còn đó. Trả về một danh sách mới không thể là $O(1)$. Ta sẽ hiện thực nó bằng cách duyệt danh sách anh em, với độ phức tạp O(số con). Một giải pháp $O(1)$ thực sự sẽ yêu cầu lưu trữ sẵn các danh sách con, đánh đổi không gian lưu trữ.

```python
class ExtendedFCNSTree:
    def __init__(self, parent_array: list[int]):
        self.num_nodes = len(parent_array)
        self.root_node = -1
        self.parent_array = parent_array
        self.children_count = [0] * self.num_nodes
        self.fcns_nodes = [{'first_child': -1, 'next_sibling': -1} for _ in range(self.num_nodes)]

        self._build_representation()

    def _build_representation(self):
        # Build an adjacency list (parent -> list of children) first
        adj = [[] for _ in range(self.num_nodes)]
        for i, p in enumerate(self.parent_array):
            if p != -1:
                adj[p].append(i)
            else:
                self.root_node = i

        # Populate FCNS, children_count from the adjacency list
        for i in range(self.num_nodes):
            self.children_count[i] = len(adj[i])
            if self.children_count[i] > 0:
                self.fcns_nodes[i]['first_child'] = adj[i][0]
                for j in range(self.children_count[i] - 1):
                    current_child = adj[i][j]
                    next_child = adj[i][j+1]
                    self.fcns_nodes[current_child]['next_sibling'] = next_child

    def root(self) -> int:
        return self.root_node

    def parent(self, v: int) -> int:
        return self.parent_array[v]

    def number_of_children(self, v: int) -> int:
        return self.children_count[v]

    def children(self, v: int) -> list[int]:
        # This operation is O(k) where k is number of children, not O(1)
        # unless we pre-store the lists, which increases space complexity.
```

```
42          child_list = []
43          child = self.fcns_nodes[v]['first_child']
44          while child != -1:
45              child_list.append(child)
46              child = self.fcns_nodes[child]['next_sibling']
47          return child_list
48
49  # Example
50  # P = [-1, 0, 0, 1, 1, 2, 2]
51  # tree = ExtendedFCNSTree(P)
52  # print(f"Root: {tree.root()}")
53  # print(f"Number of children of node 0: {tree.number_of_children(0)}")
54  # print(f"Children of node 1: {tree.children(1)}")
```

Listing 14: Lớp cây FCNS mở rộng.

3 Các Thuật toán Duyệt Đồ thị và Cây

3.1 Bài toán 8: BFS trên Đơn đồ thị

Đề bài 8

Let $G = (V, E)$ be a finite simple graph. Implement the breadth-first search on G.
(Cho $G = (V, E)$ là một đơn đồ thị hữu hạn. Hãy cài đặt tìm kiếm theo chiều rộng trên G.)

3.1.1 Phân tích và Nền tảng Lý thuyết

Phân tích Đề bài: Theo định nghĩa 1.6 trong tài liệu tham khảo [Val21], một đơn đồ thị (simple graph) là một đồ thị vô hướng, không có khuyên và cạnh song song. Trong bối cảnh khoa học máy tính được đề cập, nó được hiểu là một đồ thị có hướng đối xứng (bidirected), nghĩa là nếu có cạnh $(u, v) \in E$ thì cũng có cạnh $(v, u) \in E$.

Nền tảng Lý thuyết của BFS: Tìm kiếm theo chiều rộng (Breadth-First Search - BFS) là một thuật toán duyệt đồ thị cơ bản. Nó khám phá các đỉnh của đồ thị theo từng "lớp"(layer) hoặc từng "mức"(level), bắt đầu từ một đỉnh nguồn s.

- Thuật toán khám phá tất cả các đỉnh kề với đỉnh nguồn s (lớp 1).
- Sau đó, nó khám phá tất cả các đỉnh chưa được thăm kề với các đỉnh ở lớp 1 (tạo ra lớp 2).
- Quá trình này tiếp tục cho đến khi tất cả các đỉnh có thể đến được từ s đều được thăm.

BFS có một thuộc tính cực kỳ quan trọng: Trên một đồ thị không có trọng số (hoặc các cạnh có trọng số bằng nhau), BFS đảm bảo tìm thấy đường đi ngắn nhất từ đỉnh nguồn đến tất cả các đỉnh khác. "Đường đi ngắn nhất"ở đây được hiểu là đường đi có số cạnh ít nhất.

3.1.2 Mô tả Thuật toán và Cấu trúc Dữ liệu

Các cấu trúc dữ liệu cần thiết:

1. Biểu diễn đồ thị: Danh sách kề (Adjacency List) là lựa chọn hiệu quả nhất cho hầu hết các trường hợp, đặc biệt là đồ thị thưa.
2. Hàng đợi (Queue): Đây là cấu trúc dữ liệu cốt lõi của BFS, tuân theo nguyên tắc "Vào trước - Ra trước"(FIFO - First-In, First-Out). Hàng đợi này lưu trữ các đỉnh sẽ được khám phá tiếp theo.
3. Mảng/Tập hợp đánh dấu đã thăm (Visited set): Để tránh việc xử lý một đỉnh nhiều lần và ngăn chặn các chu trình vô hạn, ta cần một cấu trúc (thường là mảng boolean hoặc hash set) để lưu lại các đỉnh đã được đưa vào hàng đợi.
4. Mảng khoảng cách và đỉnh cha (tùy chọn): Để lưu kết quả tìm đường đi ngắn nhất, ta cần:
   - distance[]: Lưu khoảng cách (số cạnh) từ đỉnh nguồn.
   - parent[]: Lưu đỉnh cha trong đường đi ngắn nhất để có thể truy vết.

Các bước của thuật toán:

1. Khởi tạo:
   - Chọn một đỉnh nguồn s.
   - Tạo một hàng đợi Q và thêm s vào Q.
   - Đánh dấu s là đã thăm.
   - Khởi tạo distance[s] = 0 và parent[s] = null. Tất cả các đỉnh khác có khoảng cách là vô cực.
2. Vòng lặp chính:
   - Chừng nào Q chưa rỗng:
   - Lấy đỉnh u ra từ đầu hàng đợi Q.
   - Với mỗi đỉnh kề v của u:
   - Nếu v chưa được thăm:
     - Đánh dấu v là đã thăm.
     - Gán parent[v] = u.
     - Gán distance[v] = distance[u] + 1.
     - Thêm v vào cuối hàng đợi Q.
3. Kết thúc: Khi hàng đợi rỗng, thuật toán kết thúc. Mảng distance và parent chứa kết quả.

3.1.3 Mô phỏng Thuật toán

Sử dụng đồ thị từ Fig. 1.1 trong [Val21] và coi nó là vô hướng. Bắt đầu từ đỉnh nguồn v_1.

Bước	Đỉnh được xử lý (u)	Hàng đợi (Queue)	Hành động và Cập nhật
0	-	$[v_1]$	Khởi tạo: $d(v_1) = 0, p(v_1) = \text{null}$. Visited=$\{v_1\}$.
1	v_1	$[v_2, v_4, v_3]$	Lấy v_1. Các đỉnh kề chưa thăm: v_2, v_4, v_3. Cập nhật: $d(v_2) = 1, p(v_2) = v_1$; $d(v_4) = 1, p(v_4) = v_1$; $d(v_3) = 1, p(v_3) = v_1$. Visited=$\{$v1,v2,v4,v3$\}$.
2	v_2	$[v_4, v_3, v_5]$	Lấy v_2. Kề chưa thăm: v_5. Cập nhật: $d(v_5) = 2, p(v_5) = v_2$. Visited=$\{$...,v5$\}$.
3	v_4	$[v_3, v_5, v_6, v_7]$	Lấy v_4. Kề chưa thăm: v_6, v_7. Cập nhật: $d(v_6) = 2, p(v_6) = v_4$; $d(v_7) = 2, p(v_7) = v_4$. Visited=$\{$...,v6,v7$\}$.
4	v_3	$[v_5, v_6, v_7]$	Lấy v_3. Tất cả các đỉnh kề đã được thăm. Không có cập nhật.
5	v_5	$[v_6, v_7]$	Lấy v_5. Tất cả các đỉnh kề đã được thăm. Không có cập nhật.
6	v_6	$[v_7]$	Lấy v_6. Tất cả các đỉnh kề đã được thăm. Không có cập nhật.
7	v_7	[]	Lấy v_7. Tất cả các đỉnh kề đã được thăm. Không có cập nhật.

Kết quả: Khoảng cách (số cạnh ít nhất) từ v_1 đến các đỉnh khác đã được tìm thấy.

3.1.4 Cài đặt bằng Python

```python
from collections import deque

def bfs_simple_graph(graph: dict[int, list[int]], start_node: int):
    """
    Performs Breadth-First Search on a simple graph.

    This function finds the shortest path (in terms of number of edges)
    from a
    start_node to all other reachable nodes.

    :param graph: Adjacency list representation of the simple graph.
                  e.g., {0: [1, 2], 1: [0, 3], ...}
```

```
    :param start_node: The node to start the search from.
    :return: A tuple (distance, parent) containing the results.
             distance: A dict mapping each node to its distance from start_node.
             parent: A dict mapping each node to its predecessor in the shortest path.
    """
    num_nodes = len(graph)

    # Initialize data structures
    distance = {node: float('inf') for node in range(num_nodes)}
    parent = {node: None for node in range(num_nodes)}
    visited = {node: False for node in range(num_nodes)}

    queue = deque()

    # Start the search from the start_node
    distance[start_node] = 0
    visited[start_node] = True
    queue.append(start_node)

    while queue:
        # Dequeue a vertex to visit its neighbors
        u = queue.popleft()

        # Process all neighbors of the current vertex
        for v in graph.get(u, []):
            if not visited[v]:
                visited[v] = True
                distance[v] = distance[u] + 1
                parent[v] = u
                queue.append(v)

    return distance, parent

# --- Example Usage ---
if __name__ == '__main__':
    # Representing the simple graph from Val21, Fig 1.1 as an undirected graph
    # Node indices are 0-6 corresponding to v1-v7
    simple_graph_adj = {
        0: [1, 3],          # v1 -> v2, v4
        1: [0, 4, 3],       # v2 -> v1, v5, v4
        2: [0, 3, 5],       # v3 -> v1, v4, v6
        3: [0, 1, 2, 4, 5, 6], # v4 -> v1,v2,v3,v5,v6,v7
        4: [1, 3, 6],       # v5 -> v2, v4, v7
        5: [2, 3, 6],       # v6 -> v3, v4, v7
        6: [3, 4, 5]        # v7 -> v4, v5, v6
    }
    start = 0 # v1
    distances, parents = bfs_simple_graph(simple_graph_adj, start)

    print("BFS results from node v1:")
    for i in range(len(simple_graph_adj)):
        print(f"  To v{i+1}: Distance={distances[i]}, Parent=v{parents[i]+1 if parents[i] is not None else 'N/A'}")
```

Listing 15: Cài đặt BFS cho đơn đồ thị trong Python.

3.1.5 Cài đặt bằng C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

using Graph = std::vector<std::vector<int>>;

class BfsSolver {
public:
    std::vector<int> distance;
    std::vector<int> parent;
    int num_nodes;

    BfsSolver(int n) : num_nodes(n) {
        distance.resize(n, std::numeric_limits<int>::max());
        parent.resize(n, -1); // -1 indicates no parent
    }

    void solve(const Graph& adj, int start_node) {
        std::queue<int> q;
        std::vector<bool> visited(num_nodes, false);

        // Initialize starting node
        distance[start_node] = 0;
        visited[start_node] = true;
        q.push(start_node);

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            // Explore neighbors
            for (int v : adj[u]) {
                if (!visited[v]) {
                    visited[v] = true;
                    distance[v] = distance[u] + 1;
                    parent[v] = u;
                    q.push(v);
                }
            }
        }
    }
};

// --- Example Usage ---
int main() {
    int num_nodes = 7;
    // Undirected graph from Val21, Fig 1.1
    Graph adj(num_nodes);
    adj[0] = {1, 3};
    adj[1] = {0, 4, 3};
    adj[2] = {0, 3, 5};
    adj[3] = {0, 1, 2, 4, 5, 6};
    adj[4] = {1, 3, 6};
    adj[5] = {2, 3, 6};
    adj[6] = {3, 4, 5};
```

```cpp

    int start_node = 0; // v1
    BfsSolver solver(num_nodes);
    solver.solve(adj, start_node);

    std::cout << "BFS results from node v1:" << std::endl;
    for (int i = 0; i < num_nodes; ++i) {
        std::cout << "  To v" << i + 1 << ": Distance=" << solver.distance[i] << ", Parent=";
        if (solver.parent[i] != -1) {
            std::cout << "v" << solver.parent[i] + 1 << std::endl;
        } else {
            std::cout << "N/A" << std::endl;
        }
    }

    return 0;
}
```

Listing 16: Cài đặt BFS cho đơn đồ thị trong C++.

3.1.6 Độ phức tạp

Độ phức tạp của thuật toán BFS được xác định bởi cách nó duyệt qua các đỉnh và cạnh.

- Mỗi đỉnh được đưa vào và lấy ra khỏi hàng đợi đúng một lần. Thao tác này có độ phức tạp $O(1)$. Tổng cộng là $O(|V|)$.
- Khi một đỉnh u được xử lý, thuật toán sẽ duyệt qua tất cả các cạnh kề của nó. Trong suốt quá trình chạy, mỗi cạnh (u,v) sẽ được xem xét hai lần (một lần từ u, một lần từ v trong đồ thị vô hướng). Tổng số lần duyệt qua danh sách kề là $O(|E|)$.
- Tổng độ phức tạp: $O(|V|+|E|)$.

3.2 Bài toán 9: BFS trên Đa đồ thị

Đề bài 9

Let $G=(V,E)$ be a finite multigraph. Implement the breadth-first search on G. (Cho $G=(V,E)$ là một đa đồ thị hữu hạn. Hãy cài đặt tìm kiếm theo chiều rộng trên G.)

3.2.1 Phân tích và Nền tảng Lý thuyết

Phân tích Đề bài: Một đa đồ thị (multigraph) là một sự mở rộng của đơn đồ thị, trong đó cho phép tồn tại nhiều hơn một cạnh (gọi là cạnh song song - parallel edges) giữa cùng một cặp đỉnh. Ví dụ, từ đỉnh u đến đỉnh v có thể tồn tại hai hoặc nhiều cạnh. Giống như đơn đồ thị, trong bối cảnh này, ta sẽ coi nó là đồ thị vô hướng (bidirected).

Nền tảng Lý thuyết của BFS trên Đa đồ thị: Mục tiêu cốt lõi của BFS là duyệt đồ thị theo từng lớp (layer) và, trên đồ thị không trọng số, tìm đường đi có số cạnh ít nhất. Câu hỏi đặt ra là: Sự tồn tại của các cạnh song song ảnh hưởng đến nguyên lý này như thế nào?

Câu trả lời là: Không ảnh hưởng.

- Khái niệm Khoảng cách: Khoảng cách giữa hai đỉnh u và v trong BFS được định nghĩa là số cạnh trên đường đi ngắn nhất. Nếu có nhiều cạnh song song giữa u và v, tất cả chúng đều đại diện cho một đường đi có độ dài là 1. Thuật toán không cần phân biệt chúng; việc tồn tại ít nhất một cạnh là đủ để xác lập một liên kết trực tiếp.

- Tính Toàn vẹn của Thuật toán: Logic của BFS dựa trên hàng đợi (Queue) và tập hợp đỉnh đã thăm (‘visited‘). Giả sử thuật toán đang xử lý đỉnh u. Nó sẽ duyệt qua danh sách kề của u.

  1. Khi nó gặp đỉnh v lần đầu tiên qua một cạnh song song $(u,v)_1$, nếu v chưa được thăm, BFS sẽ cập nhật distance[v], parent[v], đánh dấu v là đã thăm và đưa v vào hàng đợi.
  2. Ngay sau đó, khi nó gặp một cạnh song song khác là $(u,v)_2$, nó sẽ kiểm tra và thấy rằng v đã được đánh dấu là "đã thăm". Do đó, thuật toán sẽ bỏ qua cạnh này và tiếp tục.

  Cơ chế này đảm bảo rằng mỗi đỉnh chỉ được đưa vào hàng đợi và xử lý đúng một lần, tại lớp gần nhất có thể so với đỉnh nguồn. Do đó, tính đúng đắn của BFS trong việc tìm đường đi ngắn nhất (về số cạnh) được bảo toàn hoàn toàn trên đa đồ thị.

3.2.2 Mô tả Thuật toán và Cấu trúc Dữ liệu

Thuật toán và cấu trúc dữ liệu cho BFS trên đa đồ thị về cơ bản là giống hệt với phiên bản cho đơn đồ thị.

- Cấu trúc dữ liệu:
  - Biểu diễn đồ thị: Phải là một cấu trúc có khả năng lưu trữ cạnh song song. Danh sách kề (Adjacency List) là lựa chọn lý tưởng. Ví dụ, trong C++, ‘std::vector<std::vector<int»‘ cho phép lưu các giá trị đỉnh kề trùng lặp. Trong Python, ‘dict[int, list[int]]‘ cũng thực hiện được điều tương tự.
  - Hàng đợi (Queue): Để duy trì thứ tự duyệt FIFO.
  - Mảng/Tập hợp đã thăm (Visited): Để đảm bảo mỗi đỉnh chỉ được xử lý một lần. Đây là thành phần then chốt giúp xử lý cạnh song song một cách tự nhiên.
- Các bước của thuật toán: Không thay đổi so với thuật toán BFS chuẩn đã mô tả trong Bài toán 8.

3.2.3 Mô phỏng Thuật toán

Ta sử dụng lại đồ thị từ Fig. 1.1 [Val21] nhưng thêm vào một cạnh song song giữa v_1 và v_4. Danh sách kề của v_1 sẽ là $[v_2, v_4, v_4, v_3]$. Bắt đầu từ nguồn v_1.

Bước	Xử lý (u)	Hàng đợi (Queue)	Hành động và Cập nhật
0	-	$[v_1]$	Khởi tạo: $d(v_1) = 0$. Visited=$\{v_1\}$.
1	v_1	$[v_2, v_4, v_3]$	Lấy v_1. Duyệt kề: v_2, v_4, v_4, v_3. v_2 chưa thăm $\rightarrow$ thêm vào Q, $d(v_2) = 1$. v_4 chưa thăm $\rightarrow$ thêm vào Q, $d(v_4) = 1$. v_4 đã thăm (trong lượt này) $\rightarrow$ bỏ qua. v_3 chưa thăm $\rightarrow$ thêm vào Q, $d(v_3) = 1$.
2	v_2	$[v_4, v_3, v_5]$	Lấy v_2. Kề chưa thăm: v_5. Cập nhật: $d(v_5) = 2, p(v_5) = v_2$.
... Các bước còn lại diễn ra tương tự như Bài toán 8 ...			

Mô phỏng cho thấy cạnh song song (v_1, v_4) thứ hai không gây ra bất kỳ thay đổi nào trong kết quả hoặc logic của thuật toán, vì v_4 đã được xử lý ngay từ lần gặp đầu tiên.

3.2.4 Cài đặt bằng Python

Mã nguồn có thể tái sử dụng gần như hoàn toàn từ Bài toán 8. Sự khác biệt chỉ nằm ở dữ liệu đầu vào.

```
from collections import deque
from typing import Dict, List, Tuple, Set

def bfs_multigraph(graph: Dict[int, List[int]], start_node: int) -> Tuple[Dict[int, float], Dict[int, int | None]]:
    """
    Performs Breadth-First Search on a multigraph.
    The logic is identical to the simple graph version, as the 'visited' set
    naturally handles the fact that parallel edges lead to the same neighbor.

    :param graph: Adjacency list representation. Parallel edges mean a neighbor
                  can appear multiple times in the list, e.g., {0: [1, 1, 2]}.
    :param start_node: The node to start the search from.
    :return: A tuple (distance, parent).
    """
    # Find all nodes to correctly initialize data structures
    all_nodes: Set[int] = set(graph.keys())
    for neighbors in graph.values():
        for neighbor in neighbors:
```

```
            all_nodes.add(neighbor)

    if not all_nodes:
        return {}, {}

    # Initialize data structures for all potential nodes
    distance: Dict[int, float] = {node: float('inf') for node in
    all_nodes}
    parent: Dict[int, int | None] = {node: None for node in all_nodes}

    queue = deque()

    # Start the search from the start_node if it exists
    if start_node in distance:
        distance[start_node] = 0
        queue.append(start_node)
    else: # Start node not in graph
        return distance, parent

    visited_in_queue = {start_node} # More efficient than a full visited
    array

    while queue:
        u = queue.popleft()

        for v in graph.get(u, []):
            if v not in visited_in_queue:
                visited_in_queue.add(v)
                distance[v] = distance[u] + 1
                parent[v] = u
                queue.append(v)

    return distance, parent

# --- Example Usage ---
if __name__ == '__main__':
    # A multigraph with parallel edges (0,1) and (3,6)
    multigraph_adj = {
        0: [1, 1, 3],       # v1 -> v2 (twice), v4
        1: [0, 0, 4, 3],    # v2 -> v1 (twice), v5, v4
        2: [0, 3, 5],
        3: [0, 1, 2, 4, 5, 6, 6], # v4 -> v7 (twice)
        4: [1, 3, 6],
        5: [2, 3, 6],
        6: [3, 4, 5, 3]     # v7 -> v4 (twice)
    }
    start = 0 # v1
    distances, parents = bfs_multigraph(multigraph_adj, start)

    print("BFS results on Multigraph from node v1:")
    sorted_nodes = sorted(list(distances.keys()))
    for i in sorted_nodes:
        parent_node = f"v{parents[i]+1}" if parents.get(i) is not None
    else 'N/A'
        print(f"  To v{i+1}: Distance={distances.get(i, 'inf')}, Parent={
    parent_node}")
```

Listing 17: Cài đặt BFS cho đa đồ thị trong Python.

3.2.5 Cài đặt bằng C++

Tương tự như Python, lớp 'BfsSolver' đã được viết cho đơn đồ thị có thể được tái sử dụng trực tiếp cho đa đồ thị mà không cần thay đổi logic.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <map>

// The BfsSolver class from Problem 8 can be reused without any changes.
// The std::vector<std::vector<int>> adjacency list naturally supports
// storing duplicate neighbors for parallel edges.
using Graph = std::vector<std::vector<int>>;

class BfsSolver {
public:
    std::vector<int> distance;
    std::vector<int> parent;
    int num_nodes;

    BfsSolver(int n) : num_nodes(n) {
        distance.resize(n, std::numeric_limits<int>::max());
        parent.resize(n, -1);
    }

    void solve(const Graph& adj, int start_node) {
        std::queue<int> q;
        std::vector<bool> visited(num_nodes, false);

        if (start_node >= num_nodes) return;

        distance[start_node] = 0;
        visited[start_node] = true;
        q.push(start_node);

        while (!q.empty()) {
            int u = q.front();
            q.pop();

            // The loop iterates over all neighbors. If v is listed twice
            // due to parallel edges, the second time it will be caught
            // by the `if (!visited[v])` check.
            for (int v : adj[u]) {
                if (!visited[v]) {
                    visited[v] = true;
                    distance[v] = distance[u] + 1;
                    parent[v] = u;
                    q.push(v);
                }
            }
        }
    }
};

// --- Example Usage ---
int main() {
    int num_nodes = 7;
```

```cpp
    // A multigraph representation where parallel edges result in
    duplicate entries.
    Graph adj(num_nodes);
    adj[0] = {1, 1, 3};          // v1 has two edges to v2
    adj[1] = {0, 0, 4, 3};
    adj[2] = {0, 3, 5};
    adj[3] = {0, 1, 2, 4, 5, 6, 6}; // v4 has two edges to v7
    adj[4] = {1, 3, 6};
    adj[5] = {2, 3, 6};
    adj[6] = {3, 4, 5, 3};

    int start_node = 0; // v1
    BfsSolver solver(num_nodes);
    solver.solve(adj, start_node);

    std::cout << "BFS results on Multigraph from node v1:" << std::endl;
    for (int i = 0; i < num_nodes; ++i) {
        std::cout << "  To v" << i + 1 << ": Distance=";
        if (solver.distance[i] == std::numeric_limits<int>::max()) {
            std::cout << "inf";
        } else {
            std::cout << solver.distance[i];
        }
        std::cout << ", Parent=";
        if (solver.parent[i] != -1) {
            std::cout << "v" << solver.parent[i] + 1 << std::endl;
        } else {
            std::cout << "N/A" << std::endl;
        }
    }
    return 0;
}
```

Listing 18: Cài đặt BFS cho đa đồ thị trong C++.

3.2.6 Kết luận cho Bài toán 9

Thuật toán BFS chứng tỏ được sự mạnh mẽ và tính tổng quát của mình khi áp dụng cho đa đồ thị. Logic cơ bản của việc duyệt theo lớp và sử dụng trạng thái "đã thăm"là đủ để xử lý một cách hiệu quả sự phức tạp do các cạnh song song gây ra mà không cần bất kỳ sự sửa đổi nào về mặt thuật toán. Sự khác biệt chính chỉ nằm ở việc đảm bảo cấu trúc dữ liệu đầu vào (danh sách kề) có khả năng biểu diễn các cạnh song song. Độ phức tạp tính toán vẫn được duy trì ở mức tối ưu là $O(|V|+|E|)$, trong đó $|E|$ là tổng số cạnh trong đa đồ thị, bao gồm cả các cạnh lặp lại.

3.3 Bài toán 10: BFS trên Đồ thị Tổng quát

Đề bài 10

Let $G = (V, E)$ be a general graph. Implement the breadth-first search on G. (Cho $G = (V, E)$ là một đồ thị tổng quát. Hãy cài đặt tìm kiếm theo chiều rộng trên G.)

3.3.1 Phân tích và Nền tảng Lý thuyết

Phân tích Đề bài: Một đồ thị tổng quát (general graph) là loại đồ thị có ít ràng buộc nhất. Theo các định nghĩa trong [Val21] và ngữ cảnh thông thường, nó có thể bao gồm:

1. Cạnh song song (Parallel Edges): Giống như đa đồ thị.
2. Khuyên (Self-loops): Cạnh nối một đỉnh tới chính nó, ví dụ (u, u).

Không giống như các thuật toán dựa trên trọng số (như Dijkstra), BFS không quan tâm đến trọng số cạnh. Do đó, câu hỏi trọng tâm là: các đặc điểm cấu trúc như cạnh song song và khuyên ảnh hưởng đến BFS như thế nào?

Ảnh hưởng của Đồ thị Tổng quát đến BFS: Thuật toán BFS vẫn hoạt động một cách hoàn toàn đúng đắn và an toàn trên đồ thị tổng quát. Logic cốt lõi của nó đủ mạnh để xử lý các trường hợp này một cách tự nhiên.

- Cạnh song song: Như đã phân tích chi tiết trong Bài toán 9, cơ chế 'visited' đảm bảo rằng một đỉnh kề chỉ được khám phá một lần, bất kể có bao nhiêu cạnh song song dẫn đến nó.
- Khuyên (Self-loops): Xét một khuyên (u, u). Khi thuật toán đang xử lý đỉnh u, nó sẽ duyệt qua các đỉnh kề của u. Khi gặp chính nó qua khuyên, thuật toán sẽ kiểm tra và thấy rằng u đã được đánh dấu là "đã thăm"(vì nó đã được đưa vào hàng đợi để được xử lý). Do đó, thuật toán sẽ bỏ qua khuyên và tiếp tục xử lý các đỉnh kề khác. Khuyên không gây ra vòng lặp vô hạn hay kết quả sai.

Tóm lại, không giống như Dijkstra bị thất bại bởi trọng số âm, BFS không bị ảnh hưởng bởi các đặc điểm cấu trúc của đồ thị tổng quát. Nó vẫn sẽ tìm thấy đường đi có số cạnh ít nhất một cách chính xác.

3.3.2 Mô tả Thuật toán và Cài đặt

Không có bất kỳ sự thay đổi nào về mặt logic thuật toán hay cấu trúc dữ liệu cần thiết so với phiên bản cho đa đồ thị (Bài toán 9).

- Cấu trúc dữ liệu: Danh sách kề (Adjacency List) vẫn là lựa chọn tối ưu vì nó dễ dàng biểu diễn cả cạnh song song và khuyên (bằng cách thêm một đỉnh vào danh sách kề của chính nó).
- Các bước của thuật toán: Giữ nguyên.

Do đó, mã nguồn cài đặt cho đa đồ thị có thể được tái sử dụng trực tiếp để giải quyết bài toán này. Sự khác biệt duy nhất là dữ liệu đồ thị đầu vào sẽ bao gồm cả khuyên.

3.3.3 Cài đặt bằng Python

Mã nguồn giống hệt với Bài toán 9. Ta chỉ cần minh họa bằng một ví dụ đồ thị tổng quát hơn.

```python
from collections import deque
from typing import Dict, List, Tuple, Set

def bfs_general_graph(graph: Dict[int, List[int]], start_node: int) -> Tuple[Dict[int, float], Dict[int, int | None]]:
    """
    Performs Breadth-First Search on a general graph.
    The logic is robust enough to handle parallel edges and self-loops correctly
    due to the 'visited' (or in this case, 'visited_in_queue') tracking mechanism.

    :param graph: Adjacency list representation, e.g., {0: [1, 1], 1: [1, 2]}
    :param start_node: The node to start the search from.
    :return: A tuple (distance, parent).
    """
    all_nodes: Set[int] = set(graph.keys())
    for neighbors in graph.values():
        for neighbor in neighbors:
            all_nodes.add(neighbor)

    if not all_nodes:
        return {}, {}

    distance: Dict[int, float] = {node: float('inf') for node in all_nodes}
    parent: Dict[int, int | None] = {node: None for node in all_nodes}
    queue = deque()

    if start_node in distance:
        distance[start_node] = 0
        queue.append(start_node)
    else:
        return distance, parent

    visited_in_queue = {start_node}

    while queue:
        u = queue.popleft()

        for v in graph.get(u, []):
            # The check `v not in visited_in_queue` correctly handles:
            # 1. Parallel edges: the second edge to `v` is ignored.
            # 2. Self-loops (where u == v): `u` is already in `visited_in_queue`, so it's ignored.
            if v not in visited_in_queue:
                visited_in_queue.add(v)
                distance[v] = distance[u] + 1
                parent[v] = u
                queue.append(v)

    return distance, parent

# --- Example Usage ---
if __name__ == '__main__':
    # A general graph with parallel edges (0,1) and a self-loop on node 2
    general_graph_adj = {
```

```
        0: [1, 1, 3],       # v1 -> v2 (twice), v4
        1: [0, 4],
        2: [0, 2, 5],       # v3 -> v1, v3 (self-loop), v6
        3: [0, 1, 4],
        4: [1, 3],
        5: [2]
    }
    start = 0 # v1
    distances, parents = bfs_general_graph(general_graph_adj, start)

    print("BFS results on General Graph from node v1:")
    sorted_nodes = sorted(list(distances.keys()))
    for i in sorted_nodes:
        parent_node = f"v{parents[i]+1}" if parents.get(i) is not None else 'N/A'
        print(f"  To v{i+1}: Distance={distances.get(i, 'inf')}, Parent={parent_node}")
```

Listing 19: Cài đặt BFS cho đồ thị tổng quát trong Python.

3.3.4 Cài đặt bằng C++

Lớp 'BfsSolver' từ các bài trước hoàn toàn có thể tái sử dụng mà không cần thay đổi một dòng code nào.

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <map>

// The BfsSolver class from Problem 8/9 is fully compatible.
// No changes to the algorithm's logic are needed.
using Graph = std::vector<std::vector<int>>;

class BfsSolver {
public:
    std::vector<int> distance;
    std::vector<int> parent;
    int num_nodes;

    BfsSolver(int n) : num_nodes(n) {
        distance.resize(n, std::numeric_limits<int>::max());
        parent.resize(n, -1);
    }

    void solve(const Graph& adj, int start_node) {
        std::queue<int> q;
        std::vector<bool> visited(num_nodes, false);

        if (start_node >= num_nodes) return;

        distance[start_node] = 0;
        visited[start_node] = true;
        q.push(start_node);

        while (!q.empty()) {
            int u = q.front();
```

```cpp
            q.pop();

            // The logic correctly handles self-loops and parallel edges.
            for (int v : adj[u]) {
                // If v is u (self-loop), u is already visited, so this is false.
                // If v is a neighbor via a parallel edge, it would have been
                // visited by the first edge.
                if (!visited[v]) {
                    visited[v] = true;
                    distance[v] = distance[u] + 1;
                    parent[v] = u;
                    q.push(v);
                }
            }
        }
    }
};

// --- Example Usage ---
int main() {
    int num_nodes = 6;
    // A general graph representation with parallel edges and a self-loop
    Graph adj(num_nodes);
    adj[0] = {1, 1, 3};       // v1 -> v2 (twice), v4
    adj[1] = {0, 4};
    adj[2] = {0, 2, 5};       // v3 -> v1, v3 (self-loop), v6
    adj[3] = {0, 1, 4};
    adj[4] = {1, 3};
    adj[5] = {2};

    int start_node = 0; // v1
    BfsSolver solver(num_nodes);
    solver.solve(adj, start_node);

    std::cout << "BFS results on General Graph from node v1:" << std::endl;
    for (int i = 0; i < num_nodes; ++i) {
        std::cout << "  To v" << i + 1 << ": Distance=";
        if (solver.distance[i] == std::numeric_limits<int>::max()) {
            std::cout << "inf";
        } else {
            std::cout << solver.distance[i];
        }
        std::cout << ", Parent=";
        if (solver.parent[i] != -1) {
            std::cout << "v" << solver.parent[i] + 1 << std::endl;
        } else {
            std::cout << "N/A" << std::endl;
        }
    }
    return 0;
}
```

Listing 20: Cài đặt BFS cho đồ thị tổng quát trong C++.

3.3.5 Kết luận cho Bài toán 10

Thuật toán Tìm kiếm theo chiều rộng (BFS) là một thuật toán duyệt đồ thị cực kỳ mạnh mẽ và ổn định. Không giống như các thuật toán dựa trên trọng số, logic của BFS không bị ảnh hưởng bởi các đặc điểm cấu trúc phức tạp như cạnh song song hay khuyên. Cơ chế kiểm tra đỉnh đã thăm ('visited') là đủ để xử lý các trường hợp này một cách tự nhiên và hiệu quả, đảm bảo thuật toán không rơi vào vòng lặp vô hạn và luôn tìm ra đường đi có số cạnh ít nhất một cách chính xác. Do đó, một cài đặt BFS chuẩn mực cho đơn đồ thị có thể được áp dụng trực tiếp cho đa đồ thị và đồ thị tổng quát mà không cần bất kỳ sự thay đổi logic nào. Độ phức tạp tính toán vẫn là $O(|V|+|E|)$.

3.4 Bài toán 11: DFS trên Đơn đồ thị

Đề bài 11

Let $G=(V,E)$ be a finite simple graph. Implement the depth-first search on G. (Cho $G=(V,E)$ là một đơn đồ thị hữu hạn. Hãy cài đặt tìm kiếm theo chiều sâu trên G.)

3.4.1 Phân tích và Nền tảng Lý thuyết

Phân tích Đề bài: Một đơn đồ thị (simple graph) là một đồ thị vô hướng, không có khuyên và cạnh song song. Thuật toán Tìm kiếm theo chiều sâu (Depth-First Search - DFS) là một thuật toán duyệt đồ thị cơ bản với chiến lược khám phá hoàn toàn khác biệt so với BFS.

Nền tảng Toán học và Bản chất Đệ quy của DFS: DFS hoạt động theo nguyên tắc "đi sâu hết mức có thể". Bắt đầu từ một đỉnh nguồn s, thuật toán sẽ chọn một đỉnh kề chưa được thăm, đi đến đỉnh đó, và lặp lại quá trình. Nó chỉ quay lui (backtrack) khi đi vào một "ngõ cụt- một đỉnh mà tất cả các đỉnh kề của nó đều đã được thăm.

Chiến lược này có bản chất đệ quy một cách tự nhiên.

Derivation của Công thức Đệ quy: Ta có thể định nghĩa quá trình duyệt DFS từ một đỉnh u bằng một hàm đệ quy, $DFS\_VISIT(u)$. Hoạt động của hàm này có thể được mô tả như một công thức đệ quy:

1. Hành động cơ bản (Base Action): Đánh dấu đỉnh u là đã được thăm (visited[u] = true). Xử lý u (ví dụ: thêm vào một danh sách, in ra màn hình).
2. Bước đệ quy (Recursive Step): Với mỗi đỉnh kề v của u, nếu v chưa được thăm, hãy thực hiện một lời gọi đệ quy: DFS_VISIT(v).

Công thức này có thể được viết một cách hình thức:

$$\text{DFS\_VISIT}(u) = \begin{cases} \text{mark}(u); \text{process}(u); \\ \forall v \in \text{Adj}(u): & \text{if not visited}(v), \text{ call DFS\_VISIT}(v) \end{cases}$$

Trường hợp cơ sở (Base Case) của lời gọi đệ quy này là khi một đỉnh u không có bất kỳ đỉnh kề nào chưa được thăm. Tại điểm đó, vòng lặp 'for' sẽ kết thúc và hàm sẽ trả về, thực hiện bước "quay lui"lên đỉnh cha của nó trong cây duyệt.

Việc sử dụng mảng visited để lưu trạng thái của các đỉnh là một dạng của ghi nhớ (memoization), một kỹ thuật liên quan chặt chẽ đến quy hoạch động, giúp tránh việc tính toán lại (duyệt lại) các đỉnh đã được xử lý.

3.4.2 Mô tả Thuật toán và Cấu trúc Dữ liệu

Các cấu trúc dữ liệu cần thiết:

1. Biểu diễn đồ thị: Danh sách kề (Adjacency List) là lựa chọn hiệu quả.
2. Ngăn xếp (Stack): Cấu trúc dữ liệu cốt lõi của DFS.
    - Ngăn xếp ngầm định: Trong cài đặt đệ quy, ngăn xếp lời gọi hàm (call stack) của hệ thống đóng vai trò này.
    - Ngăn xếp tường minh: Trong cài đặt không đệ quy (lặp), ta sử dụng một cấu trúc dữ liệu ngăn xếp (Stack) tuân theo nguyên tắc "Vào sau - Ra trước"(LIFO - Last-In, First-Out).
3. Mảng/Tập hợp đánh dấu đã thăm (Visited set): Cực kỳ quan trọng để tránh các chu trình vô hạn và đảm bảo mỗi đỉnh chỉ được thăm một lần.
4. Mảng đỉnh cha (Parent array): Để lưu lại cây duyệt DFS và có thể truy vết đường đi.

Các bước của thuật toán (Phiên bản Đệ quy):

1. Hàm chính DFS(G, s):
    - Khởi tạo mảng visited cho tất cả các đỉnh là false.
    - Khởi tạo mảng parent.
    - Gọi hàm trợ giúp đệ quy: DFS_VISIT(s).
2. Hàm đệ quy DFS_VISIT(u):
    - Đánh dấu visited[u] = true.
    - Với mỗi đỉnh kề v của u:
    - Nếu visited[v] là false:
        - Gán parent[v] = u.
        - Gọi đệ quy DFS_VISIT(v).

3.4.3 Cài đặt bằng Python

```python
from typing import Dict, List, Optional, Set, Tuple

class DfsSolverPython:
    """
    Encapsulates both recursive and iterative DFS implementations in
    Python.
    """
    def __init__(self, graph: Dict[int, List[int]]):
        """
```

```
        Initializes the solver with a graph.

        :param graph: Adjacency list representation of the simple graph.
        """
        self.graph = graph
        self.nodes = set(graph.keys())
        # Also add nodes that might only be destinations
        for neighbors in graph.values():
            for neighbor in neighbors:
                self.nodes.add(neighbor)

        self.parent: Dict[int, Optional[int]] = {node: None for node in self.nodes}
        self.visited: Set[int] = set()

    def solve_recursive(self, start_node: int) -> Tuple[Dict[int, Optional[int]], List[int]]:
        """
        Public method to start the recursive DFS.

        :param start_node: The node to start the search from.
        :return: A tuple (parent_map, traversal_path).
        """
        self.parent = {node: None for node in self.nodes}
        self.visited = set()
        traversal_path = []

        if start_node not in self.graph:
            return {}, []

        self._dfs_recursive_util(start_node, traversal_path)
        return self.parent, traversal_path

    def _dfs_recursive_util(self, u: int, path: List[int]):
        """
        Recursive helper function for DFS.
        This is the direct implementation of the recursive formula.

        :param u: The current node to visit.
        :param path: The list to record the traversal order.
        """
        # Mark the current node as visited and process it.
        self.visited.add(u)
        path.append(u)

        # Recur for all the vertices adjacent to this vertex.
        for v in self.graph.get(u, []):
            if v not in self.visited:
                self.parent[v] = u
                self._dfs_recursive_util(v, path)

    def solve_iterative(self, start_node: int) -> Tuple[Dict[int, Optional[int]], List[int]]:
        """
        Iterative implementation of DFS using an explicit stack.

        :param start_node: The node to start the search from.
        :return: A tuple (parent_map, traversal_path).
```

```
        """
        parent_map: Dict[int, Optional[int]] = {node: None for node in self.nodes}
        visited: Set[int] = set()
        traversal_path = []

        if start_node not in self.graph:
            return {}, []

        # The stack for DFS.
        stack = [start_node]

        while stack:
            u = stack.pop()

            if u not in visited:
                visited.add(u)
                traversal_path.append(u)

                # Push unvisited neighbors onto the stack.
                # Push in reverse order to visit neighbors in their natural order.
                for v in reversed(self.graph.get(u, [])):
                    if v not in visited:
                        parent_map[v] = u
                        stack.append(v)
        return parent_map, traversal_path

# --- Example Usage ---
if __name__ == '__main__':
    # Simple graph from Val21, Fig 1.1
    simple_graph = {
        0: [1, 3], 1: [0, 4, 3], 2: [0, 3, 5], 3: [0, 1, 2, 4, 5, 6],
        4: [1, 3, 6], 5: [2, 3, 6], 6: [3, 4, 5]
    }
    node_names = {i: f"v{i+1}" for i in range(7)}
    start_node = 0

    solver = DfsSolverPython(simple_graph)

    print("=" * 60)
    print("DFS Algorithm Report for Simple Graph (Problem 11)")
    print(f"Source Node: '{node_names[start_node]}'")
    print("=" * 60)

    # --- Recursive Version ---
    print("\n--- Recursive DFS ---")
    parents_rec, path_rec = solver.solve_recursive(start_node)
    path_str_rec = " -> ".join(node_names[n] for n in path_rec)
    print(f"  Traversal Path: {path_str_rec}")
    print(f"  Parent of v7 ({node_names[6]}): {node_names[parents_rec[6]] if parents_rec[6] is not None else 'N/A'}")

    # --- Iterative Version ---
    print("\n--- Iterative DFS ---")
    parents_iter, path_iter = solver.solve_iterative(start_node)
    path_str_iter = " -> ".join(node_names[n] for n in path_iter)
    print(f"  Traversal Path: {path_str_iter}")
```

```
    print(f"  Parent of v7 ({node_names[6]}): {node_names[parents_iter[6]] if parents_iter[6] is not None else 'N/A'}")

    print("=" * 60)
```

Listing 21: Cài đặt DFS cho đơn đồ thị trong Python (cả đệ quy và lặp).

3.4.4 Cài đặt bằng C++

```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <map>
#include <string>
#include <algorithm>

using Graph = std::vector<std::vector<int>>;

class DfsSolver {
public:
    int num_nodes;
    std::vector<int> parent; // Stores the predecessor in the DFS tree.
    std::vector<bool> visited; // Tracks visited nodes.
    std::vector<int> traversal_path; // Records the order of visiting nodes.

    /**
     * @brief Constructs the DfsSolver.
     * @param n The total number of nodes in the graph.
     */
    DfsSolver(int n) : num_nodes(n) {}

    /**
     * @brief Public method to start the recursive DFS.
     * @param adj The graph as an adjacency list.
     * @param start_node The node to start from.
     */
    void solve_recursive(const Graph& adj, int start_node) {
        // Initialization for a new run
        parent.assign(num_nodes, -1);
        visited.assign(num_nodes, false);
        traversal_path.clear();

        if (start_node >= num_nodes) return;

        dfs_util(adj, start_node);
    }

    /**
     * @brief Public method for the iterative DFS.
     * @param adj The graph as an adjacency list.
     * @param start_node The node to start from.
     */
    void solve_iterative(const Graph& adj, int start_node) {
        parent.assign(num_nodes, -1);
        visited.assign(num_nodes, false);
        traversal_path.clear();
```

```
48
49          if (start_node >= num_nodes) return;
50
51          std::stack<int> s;
52          s.push(start_node);
53
54          while (!s.empty()) {
55              int u = s.top();
56              s.pop();
57
58              if (!visited[u]) {
59                  visited[u] = true;
60                  traversal_path.push_back(u);
61
62                  // Push neighbors in reverse order to visit them in
    natural order
63                  for (auto it = adj[u].rbegin(); it != adj[u].rend(); ++it
    ) {
64                      int v = *it;
65                      if (!visited[v]) {
66                          parent[v] = u;
67                          s.push(v);
68                      }
69                  }
70              }
71          }
72      }
73
74  private:
75      /**
76       * @brief Recursive utility function for DFS.
77       * This is the direct implementation of the recursive formula.
78       * @param adj The graph.
79       * @param u The current node to visit.
80       */
81      void dfs_util(const Graph& adj, int u) {
82          // Mark current node as visited and process it
83          visited[u] = true;
84          traversal_path.push_back(u);
85
86          // Recur for all adjacent vertices
87          for (int v : adj[u]) {
88              if (!visited[v]) {
89                  parent[v] = u;
90                  dfs_util(adj, v);
91              }
92          }
93      }
94  };
95
96  // --- Example Usage ---
97  int main() {
98      int num_nodes = 7;
99      Graph adj(num_nodes);
100     adj[0] = {1, 3}; adj[1] = {0, 4, 3}; adj[2] = {0, 3, 5};
101     adj[3] = {0, 1, 2, 4, 5, 6}; adj[4] = {1, 3, 6};
102     adj[5] = {2, 3, 6}; adj[6] = {3, 4, 5};
103
```

```cpp
    std::map<int, std::string> node_names;
    for(int i = 0; i < num_nodes; ++i) node_names[i] = "v" + std::to_string(i + 1);

    int start_node = 0;
    DfsSolver solver(num_nodes);

    std::cout << "============================================================" << std::endl;
    std::cout << "DFS Algorithm Report for Simple Graph (Problem 11)" << std::endl;
    std::cout << "Source Node: '" << node_names[start_node] << "'" << std::endl;
    std::cout << "============================================================" << std::endl;

    // --- Recursive ---
    solver.solve_recursive(adj, start_node);
    std::cout << "\n--- Recursive DFS ---" << std::endl;
    std::cout << "  Traversal Path: ";
    for(size_t i = 0; i < solver.traversal_path.size(); ++i) {
        std::cout << node_names[solver.traversal_path[i]] << (i == solver.traversal_path.size() - 1 ? "" : " -> ");
    }
    std::cout << std::endl;

    // --- Iterative ---
    solver.solve_iterative(adj, start_node);
    std::cout << "\n--- Iterative DFS ---" << std::endl;
    std::cout << "  Traversal Path: ";
    for(size_t i = 0; i < solver.traversal_path.size(); ++i) {
        std::cout << node_names[solver.traversal_path[i]] << (i == solver.traversal_path.size() - 1 ? "" : " -> ");
    }
    std::cout << std::endl;

    std::cout << "============================================================" << std::endl;

    return 0;
}
```

Listing 22: Cài đặt DFS cho đơn đồ thị trong C++ (cả đệ quy và lặp).

3.4.5 Độ phức tạp và Ứng dụng

Độ phức tạp: Tương tự như BFS, độ phức tạp của DFS là $O(|V|+|E|)$.

- Mỗi đỉnh được đưa vào ngăn xếp (hoặc được gọi đệ quy) đúng một lần, do đó phần xử lý đỉnh là $O(|V|)$.
- Mỗi cạnh (u,v) được duyệt qua hai lần trong đồ thị vô hướng (một lần từ u, một lần từ v), do đó phần xử lý cạnh là $O(|E|)$.

So sánh với BFS và Ứng dụng:

- Đường đi: DFS tìm thấy một đường đi, nhưng không đảm bảo đó là đường đi ngắn nhất. BFS thì đảm bảo.
- Bộ nhớ: Trong trường hợp xấu nhất (đồ thị dạng danh sách), DFS có thể đi sâu $|V|$ cấp, chiếm bộ nhớ $O(|V|)$. BFS trong trường hợp xấu nhất (đồ thị dạng sao) có thể lưu gần hết $|V|$ đỉnh trong hàng đợi, cũng chiếm bộ nhớ $O(|V|)$. Tuy nhiên, trên các đồ thị có nhánh rộng, DFS thường tiết kiệm bộ nhớ hơn.
- Ứng dụng: DFS là nền tảng cho nhiều thuật toán quan trọng khác như:
  - Phát hiện chu trình (Cycle Detection): Nếu trong lúc duyệt, DFS gặp một đỉnh kề đã được thăm nhưng không phải là cha trực tiếp của đỉnh hiện tại, thì một chu trình đã được tìm thấy.
  - Sắp xếp Tôpô (Topological Sorting): Trên các đồ thị có hướng không chu trình (DAGs).
  - Tìm các thành phần liên thông mạnh (Strongly Connected Components).

3.5 Bài toán 12: DFS trên Đa đồ thị

Đề bài 12

Let $G = (V, E)$ be a finite multigraph. Implement the depth-first search on G. (Cho $G = (V, E)$ là một đa đồ thị hữu hạn. Hãy cài đặt tìm kiếm theo chiều sâu trên G.)

3.5.1 Phân tích và Nền tảng Lý thuyết

Phân tích Đề bài: Một đa đồ thị (multigraph) cho phép tồn tại nhiều hơn một cạnh (cạnh song song) giữa cùng một cặp đỉnh. Câu hỏi đặt ra là: Các cạnh song song này ảnh hưởng như thế nào đến logic "đi sâu hết mức"và "quay lui"của thuật toán DFS?

Bản chất Đệ quy và Tính Bền vững của DFS: Câu trả lời là: Các cạnh song song không ảnh hưởng đến tính đúng đắn hay logic cốt lõi của thuật toán DFS. Sự bền vững này đến từ chính cơ chế hoạt động của nó, đặc biệt là việc sử dụng trạng thái "đã thăm".

Derivation và Phân tích Đệ quy: Hãy xem lại công thức đệ quy của DFS từ Bài toán 11:

$$\text{DFS\_VISIT}(u) = \begin{cases} \text{mark}(u); \text{process}(u); \\ \forall v \in \text{Adj}(u): \qquad \text{if not visited}(v), \text{ call DFS\_VISIT}(v) \end{cases}$$

Phân tích tác động của cạnh song song lên công thức này:

1. Giả sử thuật toán đang thực thi DFS_VISIT(u), và trong danh sách kề của u có hai cạnh song song đến đỉnh v.
2. Vòng lặp ‘for all v in Adj(u)‘ sẽ duyệt qua cạnh song song đầu tiên, $(u, v)_1$.
3. Tại thời điểm này, v chưa được thăm. Điều kiện if not visited(v) là đúng.

4. Thuật toán sẽ thực hiện lời gọi đệ quy DFS_VISIT(v). Lời gọi này sẽ khám phá toàn bộ nhánh bắt nguồn từ v. Quan trọng là, ngay khi DFS_VISIT(v) bắt đầu, nó sẽ thực hiện mark(v).

5. Sau khi toàn bộ nhánh đệ quy từ v hoàn tất và quay lui về u, vòng lặp 'for' của u sẽ tiếp tục.

6. Nó sẽ gặp cạnh song song thứ hai, $(u, v)_2$.

7. Lúc này, nó lại kiểm tra điều kiện if not visited(v). Nhưng vì v đã được đánh dấu là đã thăm ở bước 4, điều kiện này bây giờ là sai.

8. Do đó, thuật toán sẽ bỏ qua cạnh song song thứ hai và tiếp tục với các đỉnh kề khác.

Phân tích này cho thấy rằng cơ chế đệ quy, kết hợp với việc ghi nhớ trạng thái ('visited'), đã loại bỏ một cách tự nhiên tác động của các cạnh dư thừa. Thuật toán chỉ đi theo con đường đầu tiên nó tìm thấy đến một đỉnh chưa được thăm.

3.5.2 Mô tả Thuật toán và Cài đặt

Do phân tích ở trên, thuật toán và cấu trúc dữ liệu cho DFS trên đa đồ thị là giống hệt với phiên bản cho đơn đồ thị.

- Cấu trúc dữ liệu: Danh sách kề (Adjacency List) phải được sử dụng vì nó có khả năng lưu trữ các cạnh song song (bằng cách cho phép các giá trị đỉnh kề lặp lại trong danh sách).
- Logic thuật toán: Không cần thay đổi. Cả hai phiên bản đệ quy và lặp (sử dụng ngăn xếp tường minh) đều hoạt động đúng.

Sự khác biệt duy nhất nằm ở việc khởi tạo dữ liệu đồ thị đầu vào.

3.5.3 Cài đặt bằng Python

Mã nguồn từ Bài toán 11 có thể được tái sử dụng nguyên vẹn.

```
from typing import Dict, List, Optional, Set, Tuple

class DfsSolverPython:
    """
    Encapsulates DFS implementations. This class works for simple graphs,
    multigraphs, and general graphs because the core logic is robust.
    """
    def __init__(self, graph: Dict[int, List[int]]):
        """
        Initializes the solver.
        :param graph: Adjacency list representation. For multigraphs, a
    neighbor
                      can appear multiple times in the list, e.g., {0:
    [1, 1, 2]}.
        """
        self.graph = graph
        self.nodes = set(graph.keys())
        for neighbors in graph.values():
```

```
            for neighbor in neighbors:
                self.nodes.add(neighbor)

        self.parent: Dict[int, Optional[int]] = {}
        self.visited: Set[int] = set()

    def solve_recursive(self, start_node: int) -> Tuple[Dict[int, Optional[int]], List[int]]:
        """
        Public method to start the recursive DFS.
        """
        self.parent = {node: None for node in self.nodes}
        self.visited = set()
        traversal_path = []
        if start_node in self.nodes:
            self._dfs_recursive_util(start_node, traversal_path)
        return self.parent, traversal_path

    def _dfs_recursive_util(self, u: int, path: List[int]):
        """
        Recursive helper function. It naturally handles multigraphs because
        the `visited` check prevents re-visiting a node via a parallel edge.
        """
        self.visited.add(u)
        path.append(u)
        for v in self.graph.get(u, []):
            if v not in self.visited:
                self.parent[v] = u
                self._dfs_recursive_util(v, path)

# --- Example Usage ---
if __name__ == '__main__':
    # A multigraph with parallel edges (0,1) and (3,6)
    multigraph = {
        0: [1, 1, 3],       # v1 -> v2 (twice), v4
        1: [0, 0, 4, 3],    # v2 -> v1 (twice), v5, v4
        2: [0, 3, 5],
        3: [0, 1, 2, 4, 5, 6, 6], # v4 -> v7 (twice)
        4: [1, 3, 6],
        5: [2, 3, 6],
        6: [3, 4, 5, 3]
    }
    node_names = {i: f"v{i+1}" for i in range(7)}
    start_node = 0

    solver = DfsSolverPython(multigraph)

    print("=" * 60)
    print("DFS Algorithm Report for Multigraph (Problem 12)")
    print(f"Source Node: '{node_names[start_node]}'")
    print("=" * 60)

    parents_rec, path_rec = solver.solve_recursive(start_node)
    path_str_rec = " -> ".join(node_names[n] for n in path_rec)
    print(f"  Recursive Traversal Path: {path_str_rec}")
```

```
    print(f"  Parent of v7 ({node_names[6]}): {node_names[parents_rec[6]]
    if parents_rec.get(6) is not None else 'N/A'}")

    print("=" * 60)
```

Listing 23: Cài đặt DFS cho đa đồ thị trong Python.

3.5.4 Cài đặt bằng C++

Tương tự, lớp ‘DfsSolver‘ từ Bài toán 11 có thể được sử dụng lại mà không cần thay đổi.

```
#include <iostream>
#include <vector>
#include <stack>
#include <map>
#include <string>
#include <algorithm>

using Graph = std::vector<std::vector<int>>;

class DfsSolver {
public:
    int num_nodes;
    std::vector<int> parent;
    std::vector<bool> visited;
    std.vector<int> traversal_path;

    DfsSolver(int n) : num_nodes(n) {}

    /**
     * @brief Starts the recursive DFS. The logic is robust for
    multigraphs.
     * @param adj The multigraph as an adjacency list.
     * @param start_node The starting node.
     */
    void solve_recursive(const Graph& adj, int start_node) {
        parent.assign(num_nodes, -1);
        visited.assign(num_nodes, false);
        traversal_path.clear();
        if (start_node >= num_nodes) return;
        dfs_util(adj, start_node);
    }

private:
    /**
     * @brief Recursive helper for DFS.
     * It naturally handles multigraphs because the ‘visited‘ check
     * prevents re-visiting a node via a parallel edge.
     * @param adj The graph.
     * @param u The current node.
     */
    void dfs_util(const Graph& adj, int u) {
        visited[u] = true;
        traversal_path.push_back(u);

        // Iterate through all neighbors, including duplicates from
    parallel edges
```

```cpp
        for (int v : adj[u]) {
            // The check `!visited[v]` ensures that we only proceed down a path
            // for a neighbor the very first time we see it.
            if (!visited[v]) {
                parent[v] = u;
                dfs_util(adj, v);
            }
        }
    }
};

// --- Example Usage ---
int main() {
    int num_nodes = 7;
    Graph adj(num_nodes);
    // A multigraph representation where parallel edges result in duplicate entries
    adj[0] = {1, 1, 3};          // v1 -> v2 (twice), v4
    adj[1] = {0, 0, 4, 3};       // v2 -> v1 (twice), v5, v4
    adj[2] = {0, 3, 5};
    adj[3] = {0, 1, 2, 4, 5, 6, 6}; // v4 -> v7 (twice)
    adj[4] = {1, 3, 6};
    adj[5] = {2, 3, 6};
    adj[6] = {3, 4, 5, 3};

    std::map<int, std::string> node_names;
    for(int i = 0; i < num_nodes; ++i) node_names[i] = "v" + std::to_string(i + 1);

    int start_node = 0;
    DfsSolver solver(num_nodes);

    std::cout << "============================================================" << std::endl;
    std::cout << "DFS Algorithm Report for Multigraph (Problem 12)" << std::endl;
    std::cout << "Source Node: '" << node_names[start_node] << "'" << std::endl;
    std::cout << "============================================================" << std::endl;

    solver.solve_recursive(adj, start_node);
    std::cout << "  Recursive Traversal Path: ";
    for(size_t i = 0; i < solver.traversal_path.size(); ++i) {
        std::cout << node_names[solver.traversal_path[i]] << (i == solver.traversal_path.size() - 1 ? "" : " -> ");
    }
    std::cout << std::endl;

    std::cout << "============================================================" << std::endl;

    return 0;
}
```

Listing 24: Cài đặt DFS cho đa đồ thị trong C++.

3.5.5 Kết luận cho Bài toán 12

Thuật toán Tìm kiếm theo chiều sâu (DFS) thể hiện sự mạnh mẽ và tính tổng quát khi đối mặt với đa đồ thị. Bản chất đệ quy và việc sử dụng một cơ chế theo dõi trạng thái ('visited') cho phép thuật toán xử lý các cạnh song song một cách tự nhiên mà không cần bất kỳ sự thay đổi logic nào. Một khi một đỉnh đã được thăm thông qua một cạnh, tất cả các cạnh song song khác dẫn đến đỉnh đó sẽ bị bỏ qua, đảm bảo rằng mỗi đỉnh chỉ được khám phá sâu một lần. Do đó, mã nguồn cài đặt cho DFS trên đơn đồ thị có thể được tái sử dụng nguyên vẹn. Độ phức tạp tính toán vẫn là $O(|V|+|E|)$.

3.6 Bài toán 13: DFS trên Đồ thị Tổng quát

Đề bài 13

Let $G=(V,E)$ be a finite general graph. Implement the depth-first search on G. (Cho $G=(V,E)$ là một đồ thị tổng quát. Hãy cài đặt tìm kiếm theo chiều sâu trên G.)

3.6.1 Phân tích và Nền tảng Lý thuyết

Phân tích Đề bài: Một đồ thị tổng quát (general graph) có thể bao gồm cả cạnh song song và khuyên (self-loops). Chúng ta cần phân tích xem hai đặc điểm này ảnh hưởng như thế nào đến thuật toán DFS.

Tính Bền vững của DFS trên Đồ thị Tổng quát: Thuật toán DFS, giống như BFS, hoạt động một cách hoàn toàn chính xác và an toàn trên đồ thị tổng quát. Logic duyệt dựa trên trạng thái "đã thăm"của các đỉnh là đủ để xử lý tất cả các trường hợp phức tạp này mà không gây ra lỗi hay vòng lặp vô hạn.

- Cạnh song song (Parallel Edges): Như đã phân tích chi tiết trong Bài toán 12, DFS xử lý cạnh song song một cách tự nhiên. Nó sẽ đi theo cạnh đầu tiên đến một đỉnh chưa được thăm và bỏ qua tất cả các cạnh song song sau đó vì đỉnh đích đã được đánh dấu là "đã thăm".
- Khuyên (Self-loops): Xét một khuyên (u,u).
  1. Khi thuật toán đang thực thi DFS_VISIT(u), bước đầu tiên là nó sẽ đánh dấu u là đã thăm: visited[u] = true.
  2. Sau đó, nó bắt đầu duyệt qua danh sách các đỉnh kề của u.
  3. Khi nó gặp chính u thông qua khuyên, nó sẽ kiểm tra điều kiện if not visited(u).
  4. Vì u đã được đánh dấu là đã thăm ngay từ đầu, điều kiện này sẽ là sai.
  5. Do đó, thuật toán sẽ bỏ qua khuyên và tiếp tục với các đỉnh kề khác.

  Khuyên không bao giờ gây ra một lời gọi đệ quy vô hạn DFS_VISIT(u) từ chính nó.

Tóm lại, thuật toán DFS không bị ảnh hưởng bởi các đặc điểm cấu trúc của đồ thị tổng quát. Nó sẽ duyệt qua tất cả các đỉnh có thể đến được từ đỉnh nguồn một cách chính xác.

3.6.2 Mô tả Thuật toán và Cài đặt

Không có bất kỳ sự thay đổi nào về mặt logic thuật toán hay cấu trúc dữ liệu cần thiết so với các phiên bản trước.

- Cấu trúc dữ liệu: Danh sách kề (Adjacency List) vẫn là lựa chọn tối ưu, vì nó có thể biểu diễn cả cạnh song song (các mục trùng lặp) và khuyên (một đỉnh xuất hiện trong danh sách kề của chính nó).
- Logic thuật toán: Không thay đổi. Cả hai phiên bản đệ quy và lặp đều hoạt động đúng.

Do đó, mã nguồn được viết cho đơn đồ thị hoặc đa đồ thị có thể được tái sử dụng trực tiếp.

3.6.3 Cài đặt bằng Python

Mã nguồn từ các bài toán trước có thể áp dụng nguyên vẹn. Chúng ta chỉ cần tạo một ví dụ đồ thị tổng quát để minh họa.

```python
from typing import Dict, List, Optional, Set, Tuple

class DfsSolverPython:
    """
    Encapsulates DFS implementations. This class is robust enough to handle
    simple graphs, multigraphs, and general graphs without modification
    to its core logic.
    """
    def __init__(self, graph: Dict[int, List[int]]):
        """
        :param graph: Adjacency list representation. Can contain parallel edges
                      (e.g., {0: [1, 1]}) and self-loops (e.g., {2: [2]}).
        """
        self.graph = graph
        self.nodes = set(graph.keys())
        for neighbors in graph.values():
            for neighbor in neighbors:
                self.nodes.add(neighbor)

        self.parent: Dict[int, Optional[int]] = {}
        self.visited: Set[int] = set()

    def solve_recursive(self, start_node: int) -> Tuple[Dict[int, Optional[int]], List[int]]:
        """
        Public method to start the recursive DFS.
        """
        self.parent = {node: None for node in self.nodes}
        self.visited = set()
        traversal_path = []
        if start_node in self.nodes:
            self._dfs_recursive_util(start_node, traversal_path)
        return self.parent, traversal_path

```

```
    def _dfs_recursive_util(self, u: int, path: List[int]):
        """
        Recursive helper function. The 'visited' check correctly handles
        both parallel edges and self-loops.
        """
        self.visited.add(u)
        path.append(u)
        for v in self.graph.get(u, []):
            # If v is u (self-loop), v is already in 'visited'.
            # If v is from a parallel edge, v is also already in 'visited'.
            if v not in self.visited:
                self.parent[v] = u
                self._dfs_recursive_util(v, path)

# --- Example Usage ---
if __name__ == '__main__':
    # A general graph with parallel edges (0,1) and a self-loop on node 2
    general_graph = {
        0: [1, 1, 3],       # v1 -> v2 (twice), v4
        1: [0, 4],
        2: [0, 2, 5],       # v3 -> v1, v3 (self-loop), v6
        3: [0, 1, 4],
        4: [1, 3],
        5: [2]
    }
    node_names = {i: f"v{i+1}" for i in range(6)}
    start_node = 0

    solver = DfsSolverPython(general_graph)

    print("=" * 60)
    print("DFS Algorithm Report for General Graph (Problem 13)")
    print(f"Source Node: '{node_names[start_node]}'")
    print("=" * 60)

    parents_rec, path_rec = solver.solve_recursive(start_node)
    path_str_rec = " -> ".join(node_names[n] for n in path_rec)
    print(f"  Recursive Traversal Path: {path_str_rec}")
    print(f"  Parent of v6 ({node_names[5]}): {node_names[parents_rec[5]] if parents_rec.get(5) is not None else 'N/A'}")

    print("=" * 60)
```

Listing 25: Cài đặt DFS cho đồ thị tổng quát trong Python.

3.6.4 Cài đặt bằng C++

Lớp 'DfsSolver' từ các bài trước hoàn toàn tương thích.

```
#include <iostream>
#include <vector>
#include <stack>
#include <map>
#include <string>
#include <algorithm>

using Graph = std::vector<std::vector<int>>;
```

```cpp

class DfsSolver {
public:
    int num_nodes;
    std::vector<int> parent;
    std::vector<bool> visited;
    std::vector<int> traversal_path;

    DfsSolver(int n) : num_nodes(n) {}

    /**
     * @brief Starts the recursive DFS. The logic is robust for general graphs.
     * @param adj The general graph as an adjacency list.
     * @param start_node The starting node.
     */
    void solve_recursive(const Graph& adj, int start_node) {
        parent.assign(num_nodes, -1);
        visited.assign(num_nodes, false);
        traversal_path.clear();
        if (start_node >= num_nodes) return;
        dfs_util(adj, start_node);
    }

private:
    /**
     * @brief Recursive helper for DFS.
     * The `visited[v]` check handles all cases:
     * - If v is a new node, it proceeds.
     * - If v is from a parallel edge, v is already visited, so it stops.
     * - If v is u (self-loop), u is already visited, so it stops.
     * @param adj The graph.
     * @param u The current node.
     */
    void dfs_util(const Graph& adj, int u) {
        visited[u] = true;
        traversal_path.push_back(u);

        for (int v : adj[u]) {
            if (!visited[v]) {
                parent[v] = u;
                dfs_util(adj, v);
            }
        }
    }
};

// --- Example Usage ---
int main() {
    int num_nodes = 6;
    Graph adj(num_nodes);
    // A general graph with parallel edges (0,1) and a self-loop on node 2
    adj[0] = {1, 1, 3};       // v1 -> v2 (twice), v4
    adj[1] = {0, 4};
    adj[2] = {0, 2, 5};       // v3 -> v1, v3 (self-loop), v6
    adj[3] = {0, 1, 4};
    adj[4] = {1, 3};
```

```
    adj[5] = {2};

    std::map<int, std::string> node_names;
    for(int i = 0; i < num_nodes; ++i) node_names[i] = "v" + std::
    to_string(i + 1);

    int start_node = 0;
    DfsSolver solver(num_nodes);

    std::cout << "============================================================" <<
    std::endl;
    std::cout << "DFS Algorithm Report for General Graph (Problem 13)" <<
     std::endl;
    std::cout << "Source Node: '" << node_names[start_node] << "'" << std
    ::endl;
    std::cout << "============================================================" <<
    std::endl;

    solver.solve_recursive(adj, start_node);
    std::cout << "  Recursive Traversal Path: ";
    for(size_t i = 0; i < solver.traversal_path.size(); ++i) {
        std::cout << node_names[solver.traversal_path[i]] << (i == solver
    .traversal_path.size() - 1 ? "" : " -> ");
    }
    std::cout << std::endl;

    std::cout << "============================================================" <<
    std::endl;

    return 0;
}
```

Listing 26: Cài đặt DFS cho đồ thị tổng quát trong C++.

3.6.5 Kết luận cho Bài toán 13

Thuật toán Tìm kiếm theo chiều sâu (DFS) một lần nữa chứng tỏ sự mạnh mẽ và tính tổng quát của nó khi áp dụng trên các đồ thị có cấu trúc phức tạp. Cơ chế duyệt đệ quy kết hợp với việc theo dõi trạng thái "đã thăm"là đủ để xử lý chính xác cả cạnh song song và khuyên mà không cần bất kỳ sự thay đổi logic nào. Điều này làm cho DFS trở thành một công cụ duyệt đồ thị cực kỳ linh hoạt và đáng tin cậy. Một cài đặt DFS chuẩn cho đơn đồ thị có thể được áp dụng nguyên vẹn cho đa đồ thị và đồ thị tổng quát. Độ phức tạp tính toán vẫn được duy trì ở mức tối ưu là $O(|V|+|E|)$.

4 Bài toán 7: Duyệt cây (Tree Traversal)

4.1 Phân tích và Phát biểu Bài toán

Đề bài 7 (Tree traversal – Duyệt cây)

Viết chương trình C/C++, Python để duyệt cây: (a) preorder traversal, (b) postorder traversal, (c) top-down traversal, (d) bottom-up traversal.

4.1.1 Phân tích và Diễn giải

Bài toán yêu cầu cài đặt các thuật toán duyệt cây cơ bản. Để có thể cài đặt, ta cần một cấu trúc cây cụ thể để làm việc. Ta sẽ giả định làm việc trên một **cây nhị phân (binary tree)**, vì đây là cấu trúc phổ biến nhất và làm nổi bật sự khác biệt giữa các kiểu duyệt.

Các thuật toán duyệt cây được chia thành hai nhóm chính:

1. Duyệt theo chiều sâu (Depth-First Search - DFS): Đi sâu vào một nhánh cho đến khi hết đường, sau đó quay lui.
    - (a) Pre-order (Thứ tự trước / Tiền thứ tự): Thăm nút gốc, sau đó duyệt đệ quy cây con trái, rồi duyệt đệ quy cây con phải. (Nút → Trái → Phải).
    - (b) Post-order (Thứ tự sau / Hậu thứ tự): Duyệt đệ quy cây con trái, sau đó duyệt đệ quy cây con phải, rồi mới thăm nút gốc. (Trái → Phải → Nút).
    - In-order (Thứ tự giữa / Trung thứ tự): Duyệt đệ quy cây con trái, thăm nút gốc, rồi duyệt đệ quy cây con phải. (Trái → Nút → Phải). Mặc dù không được yêu cầu rõ ràng, đây là một phần quan trọng của DFS.
2. Duyệt theo chiều rộng (Breadth-First Search - BFS): Duyệt tất cả các nút ở cùng một mức (level) trước khi chuyển sang mức tiếp theo.
    - (c) Top-down traversal (Duyệt từ trên xuống): Đây chính là tên gọi khác của duyệt theo chiều rộng (BFS), hay còn gọi là duyệt theo mức (Level-order traversal).
    - (d) Bottom-up traversal (Duyệt từ dưới lên): Đây là một biến thể của BFS, trong đó ta duyệt các mức từ lá lên gốc.

4.2 Nền tảng Thuật toán

4.2.1 Cấu trúc nút cây

Ta sẽ định nghĩa một cấu trúc cơ bản cho một nút trên cây nhị phân.

- value: Giá trị của nút.
- left: Con trỏ/tham chiếu đến nút con bên trái.
- right: Con trỏ/tham chiếu đến nút con bên phải.

4.2.2 Các thuật toán duyệt cây

Pre-order Traversal (Đệ quy):

```
function preorder(node):
  if node is null: return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```

Post-order Traversal (Đệ quy):

```
function postorder(node):
  if node is null: return
  postorder(node.left)
  postorder(node.right)
  visit(node)
```

Top-down / Level-order Traversal (Sử dụng hàng đợi - Queue):

```
function levelorder(root):
  if root is null: return
  queue = new Queue()
  queue.enqueue(root)
  while queue is not empty:
    node = queue.dequeue()
    visit(node)
    if node.left is not null:
      queue.enqueue(node.left)
    if node.right is not null:
      queue.enqueue(node.right)
```

Bottom-up Traversal (Biến thể của Level-order): Ta có thể sửa đổi thuật toán Level-order để đạt được kết quả duyệt từ dưới lên.

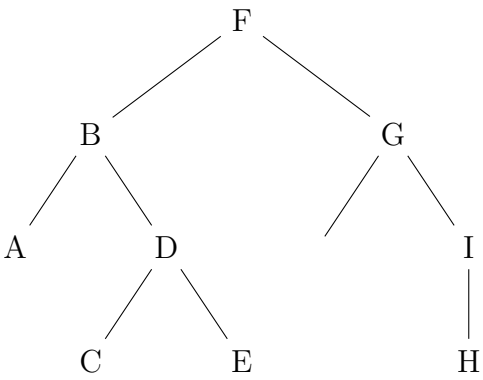
```
function bottomup(root):
  if root is null: return
  queue = new Queue()
  stack = new Stack() // Sử dụng thêm một ngăn xếp
  queue.enqueue(root)
  while queue is not empty:
    node = queue.dequeue()
    stack.push(node) // Thay vì visit, ta đẩy vào stack
    // Duyệt phải trước, rồi trái
    if node.right is not null:
      queue.enqueue(node.right)
    if node.left is not null:
      queue.enqueue(node.left)

  // In ra kết quả từ stack
  while stack is not empty:
    node = stack.pop()
    visit(node)
```

Một cách khác là thực hiện Level-order, lưu kết quả các mức vào một danh sách các danh sách, rồi in ngược lại.

4.3 Cài đặt và Phân tích Mã nguồn

Để minh họa, ta sẽ xây dựng một cây nhị phân cụ thể và thực hiện các phép duyệt trên đó.



4.3.1 Cài đặt bằng Python

```python
from collections import deque

class TreeNode:
    """A node in a binary tree."""
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

# --- (a) Pre-order Traversal (DFS: Root -> Left -> Right) ---
def preorder_traversal(node):
    if not node:
        return []
    return [node.value] + preorder_traversal(node.left) +
    preorder_traversal(node.right)

# --- (b) Post-order Traversal (DFS: Left -> Right -> Root) ---
def postorder_traversal(node):
    if not node:
        return []
    return postorder_traversal(node.left) + postorder_traversal(node.
    right) + [node.value]

# --- (c) Top-down Traversal (BFS / Level-order) ---
def top_down_traversal(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        node = queue.popleft()
        result.append(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return result

# --- (d) Bottom-up Traversal (Reverse Level-order) ---
def bottom_up_traversal(root):
```

```
    if not root:
        return []

    # Standard level-order traversal, but results are stored by level.
    levels = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        current_level = []
        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.value)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        levels.append(current_level)

    # Flatten the list of levels in reverse order.
    result = []
    for level in reversed(levels):
        result.extend(level)

    return result

def main():
    # Constructing the example tree:
    #         F
    #        / \
    #       B   G
    #      / \   \
    #     A   D   I
    #        / \ /
    #       C  E H
    root = TreeNode('F')
    root.left = TreeNode('B')
    root.right = TreeNode('G')
    root.left.left = TreeNode('A')
    root.left.right = TreeNode('D')
    root.left.right.left = TreeNode('C')
    root.left.right.right = TreeNode('E')
    root.right.right = TreeNode('I')
    root.right.right.left = TreeNode('H')

    print("--- Problem 7: Tree Traversals ---")

    # (a) Pre-order
    preorder_res = preorder_traversal(root)
    print(f"(a) Pre-order Traversal:  {preorder_res}")

    # (b) Post-order
    postorder_res = postorder_traversal(root)
    print(f"(b) Post-order Traversal: {postorder_res}")

    # (c) Top-down
    top_down_res = top_down_traversal(root)
    print(f"(c) Top-down Traversal:   {top_down_res}")
```

```python

    # (d) Bottom-up
    bottom_up_res = bottom_up_traversal(root)
    print(f"(d) Bottom-up Traversal:  {bottom_up_res}")

if __name__ == "__main__":
    main()
```

Listing 27: Các thuật toán duyệt cây nhị phân trong Python.

4.3.2 Cài đặt bằng C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <string>

using namespace std;

// A node in a binary tree
struct TreeNode {
    string value;
    TreeNode *left;
    TreeNode *right;
    TreeNode(string val) : value(val), left(nullptr), right(nullptr) {}
};

// --- (a) Pre-order Traversal (DFS: Root -> Left -> Right) ---
void preorder_traversal(TreeNode* node, vector<string>& result) {
    if (node == nullptr) return;
    result.push_back(node->value);
    preorder_traversal(node->left, result);
    preorder_traversal(node->right, result);
}

// --- (b) Post-order Traversal (DFS: Left -> Right -> Root) ---
void postorder_traversal(TreeNode* node, vector<string>& result) {
    if (node == nullptr) return;
    postorder_traversal(node->left, result);
    postorder_traversal(node->right, result);
    result.push_back(node->value);
}

// --- (c) Top-down Traversal (BFS / Level-order) ---
vector<string> top_down_traversal(TreeNode* root) {
    if (root == nullptr) return {};

    vector<string> result;
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
        result.push_back(current->value);
```

```
46
47          if (current->left) q.push(current->left);
48          if (current->right) q.push(current->right);
49      }
50      return result;
51  }
52
53  // --- (d) Bottom-up Traversal (Reverse Level-order) ---
54  vector<string> bottom_up_traversal(TreeNode* root) {
55      if (root == nullptr) return {};
56
57      vector<string> result;
58      queue<TreeNode*> q;
59      q.push(root);
60
61      // Perform a standard level-order traversal, but store results in a
        stack.
62      stack<string> s;
63      while (!q.empty()) {
64          TreeNode* current = q.front();
65          q.pop();
66          s.push(current->value);
67
68          // IMPORTANT: Enqueue right child before left child.
69          if (current->right) q.push(current->right);
70          if (current->left) q.push(current->left);
71      }
72
73      // Pop from the stack to get the reverse order.
74      while (!s.empty()) {
75          result.push_back(s.top());
76          s.pop();
77      }
78      return result;
79  }
80
81  // Helper to print a vector of strings
82  void print_vector(const vector<string>& vec) {
83      cout << "[";
84      for (size_t i = 0; i < vec.size(); ++i) {
85          cout << "'" << vec[i] << "'" << (i == vec.size() - 1 ? "" : ", ")
        ;
86      }
87      cout << "]" << endl;
88  }
89
90  int main() {
91      // Constructing the example tree:
92      TreeNode* root = new TreeNode("F");
93      root->left = new TreeNode("B");
94      root->right = new TreeNode("G");
95      root->left->left = new TreeNode("A");
96      root->left->right = new TreeNode("D");
97      root->left->right->left = new TreeNode("C");
98      root->left->right->right = new TreeNode("E");
99      root->right->right = new TreeNode("I");
100     root->right->right->left = new TreeNode("H");
101
```

```
    cout << "--- Problem 7: Tree Traversals ---" << endl;

    // (a) Pre-order
    vector<string> preorder_res;
    preorder_traversal(root, preorder_res);
    cout << "(a) Pre-order Traversal:  ";
    print_vector(preorder_res);

    // (b) Post-order
    vector<string> postorder_res;
    postorder_traversal(root, postorder_res);
    cout << "(b) Post-order Traversal: ";
    print_vector(postorder_res);

    // (c) Top-down
    vector<string> top_down_res = top_down_traversal(root);
    cout << "(c) Top-down Traversal:   ";
    print_vector(top_down_res);

    // (d) Bottom-up
    vector<string> bottom_up_res = bottom_up_traversal(root);
    cout << "(d) Bottom-up Traversal:  ";
    print_vector(bottom_up_res);

    // Note: Add memory cleanup for a production application.
    // (delete all new TreeNode(...))
    return 0;
}
```

Listing 28: Các thuật toán duyệt cây nhị phân trong C++.

4.4 Kết quả và Thảo luận

Khi chạy các chương trình trên với cây ví dụ, ta sẽ nhận được kết quả:

- Pre-order: ['F', 'B', 'A', 'D', 'C', 'E', 'G', 'I', 'H']
- Post-order: ['A', 'C', 'E', 'D', 'B', 'H', 'I', 'G', 'F']
- Top-down: ['F', 'B', 'G', 'A', 'D', 'I', 'C', 'E', 'H']
- Bottom-up: ['A', 'C', 'E', 'H', 'D', 'I', 'B', 'G', 'F']

Mỗi phương pháp duyệt cây phục vụ một mục đích khác nhau. Pre-order hữu ích để sao chép cây. Post-order hữu ích để xóa cây hoặc tính toán giá trị dựa trên các cây con. In-order (trên cây tìm kiếm nhị phân) cho ra các phần tử theo thứ tự được sắp xếp. Top-down (BFS) hữu ích để tìm đường đi ngắn nhất trong các cây không có trọng số. Bottom-up ít phổ biến hơn nhưng có thể hữu ích trong các bài toán cần xử lý các nút lá trước tiên.

5 Bài toán 6: Khoảng cách Chỉnh sửa Cây (Tree Edit Distance)

Đề bài 6

(Tree edit distance). Viết chương trình C/C++, Python để giải bài toán tree edit distance problem bằng cách sử dụng: (a) Backtracking, (b) Branch-&-bound, (c) Divide-&-conquer – chia để trị, (d) Dynamic programming – Quy hoạch động.

5.1 Phân tích Bài toán và Nền tảng Lý thuyết

Phát biểu Bài toán: Bài toán Khoảng cách Chỉnh sửa Cây nhằm tìm ra số lượng thao tác chỉnh sửa ít nhất để biến cây nguồn T_1 thành cây đích T_2. Đây là một sự mở rộng tự nhiên của bài toán Khoảng cách Chỉnh sửa Chuỗi (String Edit Distance hay Levenshtein Distance).

Các Thao tác Chỉnh sửa Cơ bản: Có ba thao tác cơ bản được định nghĩa, mỗi thao tác có một chi phí (thường là 1):

1. Xóa (Deletion): Xóa một nút u khỏi cây.
2. Chèn (Insertion): Chèn một nút v vào cây.
3. Thay thế (Substitution/Relabeling): Đổi nhãn của một nút u trong T_1 thành nhãn của một nút v trong T_2.

Ánh xạ Chỉnh sửa (Edit Mapping): Lời giải cho bài toán có thể được hình dung như một ánh xạ (mapping) giữa các nút của T_1 và T_2. Một ánh xạ M là một tập các cặp (u,v) với $u \in T_1, v \in T_2$. Mục tiêu là tìm một ánh xạ M sao cho tổng chi phí là nhỏ nhất, đồng thời phải bảo toàn cấu trúc tổ tiên (ancestry-preserving).

Bài toán con: Chỉnh sửa Rừng (Forest Edit Distance): Cốt lõi của việc giải bài toán này nằm ở việc giải một bài toán con: tính khoảng cách chỉnh sửa giữa hai rừng (forests). Khi ta quyết định thay thế gốc r_1 của T_1 bằng gốc r_2 của T_2, vấn đề còn lại là tìm cách biến đổi rừng con của r_1 thành rừng con của r_2 với chi phí thấp nhất.

5.2 Phương pháp (a): Quay lui (Backtracking)

5.2.1 Phân tích Thuật toán và Toán học

Quay lui là một phương pháp tìm kiếm vét cạn có hệ thống, duyệt qua toàn bộ không gian các lời giải tiềm năng.

Công thức Đệ quy: Ta định nghĩa hàm đệ quy $\mathrm{TED}(u,v)$ để tính chi phí biến đổi cây con gốc tại u thành cây con gốc tại v.

$$\mathrm{TED}(u,v) = \min \begin{cases} \mathrm{cost\_del}(u) + \mathrm{TED}(\mathrm{forest}(u), \emptyset) & \text{(Xóa } u \text{ và toàn bộ cây con)} \\ \mathrm{cost\_ins}(v) + \mathrm{TED}(\emptyset, \mathrm{forest}(v)) & \text{(Chèn } v \text{ và toàn bộ cây con)} \\ \mathrm{cost\_sub}(u,v) + \mathrm{ForestED}(\mathrm{forest}(u), \mathrm{forest}(v)) & \text{(Thay thế } u \to v) \end{cases}$$

Trong đó, ForestED là hàm tính khoảng cách giữa hai rừng con, và đây chính là nơi quay lui duyệt qua tất cả các cách ghép cặp cây con.

5.3 Phương pháp (b): Nhánh và Cận (Branch-and-Bound)

5.3.1 Phân tích Thuật toán và Toán học

Nhánh và cận cải tiến quay lui bằng cách cắt tỉa (pruning) các nhánh tìm kiếm không có triển vọng.

Logic Cốt lõi: Thuật toán duy trì một biến toàn cục $\min\_cost$ lưu trữ chi phí của lời giải tốt nhất đã tìm thấy. Tại mỗi bước trong cây tìm kiếm với chi phí hiện tại là $current\_cost$, nó tính toán một hàm cận dưới (lower bound function), $LB()$.

- Nếu $current\_cost + LB() \geq \min\_cost$, nhánh này sẽ bị cắt tỉa.
- Một hàm cận dưới đơn giản và hiệu quả cho bài toán con chỉnh sửa rừng ForestED(F_1, F_2) là hiệu số tuyệt đối về số lượng cây trong hai rừng: $||F_1| - |F_2||$.

5.4 Phương pháp (c): Chia để trị (Divide-and-Conquer)

5.4.1 Phân tích Thuật toán và Toán học

Trong bối cảnh này, "Chia để trị"và "Quy hoạch động"có sự giao thoa rất lớn. Thuật toán kinh điển của Zhang và Shasha chính là một thuật toán quy hoạch động dựa trên nguyên lý chia để trị.

Nguyên lý Chia để trị: Bài toán lớn TED(T_1, T_2) được chia thành các bài toán con nhỏ hơn. Các lựa chọn ở mỗi bước đều dựa trên việc xử lý gốc của các cây.

1. Chia (Divide): Tại một cặp nút (u, v), ta có ba lựa chọn: xóa u, chèn v, hoặc thay thế $u \to v$.
2. Trị (Conquer): Mỗi lựa chọn sẽ dẫn đến các lời gọi đệ quy trên các cấu trúc nhỏ hơn.
3. Tổng hợp (Combine): Lấy chi phí nhỏ nhất trong các lựa chọn trên.

Một thuật toán chia để trị thuần túy sẽ tính toán lại cùng một bài toán con nhiều lần, dẫn đến độ phức tạp hàm mũ.

5.5 Phương pháp (d): Quy hoạch động (Dynamic Programming)

5.5.1 Phân tích Thuật toán và Toán học

Đây là phương pháp hiệu quả và kinh điển nhất, chính là phiên bản tối ưu của Chia để trị bằng cách sử dụng ghi nhớ (memoization).

Công thức Quy hoạch động: Công thức chính xác giống như công thức đệ quy, nhưng được tăng cường với việc ghi nhớ.

$$\text{TED}(u,v) = \min \begin{cases} \text{cost\_del}(u) + \dots \\ \text{cost\_ins}(v) + \dots \\ \text{cost\_sub}(u,v) + \text{ForestED}(\text{forest}(u), \text{forest}(v)) \end{cases}$$

Quy hoạch động lồng: Tính ForestED Để tính chi phí biến đổi rừng $F_1 = \{t_{11}, \dots, t_{1n}\}$ thành $F_2 = \{t_{21}, \dots, t_{2m}\}$, ta xây dựng một bảng DP, gọi là sub_dp, kích thước $(n+1) \times (m+1)$. 'sub$_d p[i][j]$'*lchiphnhnhtØbinØiicyØutinca*F_1 thành j cây đầu tiên của F_2. Công thức chuyển đổi cho bảng này tương tự như thuật toán Levenshtein cho chuỗi:

$$\text{sub\_dp}[i][j] = \min \begin{cases} \text{sub\_dp}[i-1][j-1] + \text{TED}(t_{1i}, t_{2j}) & \text{(Thay thế cây)} \\ \text{sub\_dp}[i-1][j] + \text{cost}(\text{del } t_{1i}) & \text{(Xóa cây)} \\ \text{sub\_dp}[i][j-1] + \text{cost}(\text{ins } t_{2j}) & \text{(Chèn cây)} \end{cases}$$

Cấu trúc đệ quy lồng nhau: Lời gọi chính $\text{TED}(u,v)$ sẽ gọi đến ForestED, và ForestED lại gọi đệ quy đến TED cho các cây con. Việc ghi nhớ kết quả của TED là chìa khóa để giảm độ phức tạp.

6 Tổng kết và Kết luận