Đồ án cuối kì

Hoàng Quang Huy

20/07/2025

Mục lục

1 Project 4: Graph and Tree Traversing Problems **4**

2 Bài 4 **4**

3 Chuyển đổi đồ thị **4**
3.1 Ma trận kề sang Danh sách kề 4
3.2 Danh sách kề sang Ma trận kề 5
3.3 Danh sách kề sang Danh sách kề mở rộng 6
3.4 Danh sách kề mở rộng sang Danh sách kề 8
3.5 Danh sách kề sang Adjacency Map 9
3.6 Adjacency Map sang Danh sách kề 10

4 Chuyển đổi cây **12**
4.1 Array of Parents sang First-Child Next-Sibling 12
4.2 First-Child Next-Sibling sang Array of Parents 14
4.3 Array of Parents sang Graph-based Representation 16
4.4 Graph-based Representation sang Array of Parents 17
4.5 First-Child Next-Sibling sang Graph-based Representation 19
4.6 Graph-based Representation sang First-Child Next-Sibling 21
4.7 Bài toán 5 23

5 Bài 1.1: Kích thước của đồ thị đầy đủ và đồ thị hai phần đầy đủ **23**
5.1 Đồ thị đầy đủ K_n 23
5.2 Đồ thị hai phần đầy đủ $K_{p,q}$ 23

6 Bài 1.2: Điều kiện đồ thị là hai phần **24**
6.1 Chu trình C_n 24
6.2 Đồ thị đầy đủ K_n 24

7 Bài 1.3: Cây khung của đồ thị **24**

8 Bài 1.4: Mở rộng biểu diễn ma trận kề **25**

9 Bài 1.5: Cải tiến biểu diễn cây first-child, next-sibling **25**

10 Bài 1.6: Kiểm tra tính hợp lệ của cây **26**

11 Exercise 1.1 **27**

12 Exercise 1.2 **28**

13 Exercise 1.3 **29**

14 Exercise 1.4 **31**

15 Exercise 1.5 **32**

16 Exercise 1.6 **33**

17 Exercise 1.7 **34**

18 Exercise 1.8 **35**

19 Exercise 1.9 **36**

20 Exercise 1.10 **38**

21 Bài 6 **39**

22 Code Implementation **40**

23 Giải thích thuật toán **52**
23.1 Backtracking (Quay lui) 52
23.2 Branch Bound (Nhánh cận) 52
23.3 Divide Conquer (Chia để trị) 52
23.4 Dynamic Programming (Quy hoạch động) 52

24 Kết luận **52**

25 Bài 7 **53**

26 Lý thuyết **53**
26.1 Các phương thức duyệt cây 53
26.2 Cấu trúc dữ liệu 53
26.3 Thuật toán 53

27 Code **54**
27.1 Code C++ hoàn chỉnh 54

28 Kết quả thực thi **59**
28.1 Sample Tree 59
28.2 Test Case khác 60

29 Độ phức tạp **60**

30 Kết luận **61**

31 Bài toán 8: BFS trên Simple Graph **61**
31.1 Mô tả bài toán 61
31.2 Phân tích 61
31.3 Code 61

32 Bài toán 9: BFS trên Multigraph **64**
32.1 Mô tả bài toán 64
32.2 Phân tích 64
32.3 Code 64

33 Bài toán 10: BFS trên General Graph **67**
33.1 Mô tả bài toán 67
33.2 Code 67

34 Bài toán 11: DFS trên Simple Graph **69**
34.1 Mô tả bài toán 69
34.2 Code 70

35 Bài toán 12: DFS trên Multigraph **72**
35.1 Code 72

36 Bài toán 13: DFS trên General Graph **73**
36.1 Code 73

37 Chương trình chính **75**

38 So sánh và phân tích **76**

39 Kết luận **76**
39.1 Ưu điểm của BFS 76
39.2 Ưu điểm của DFS 76
39.3 Xử lý đặc biệt 76

1 Project 4: Graph and Tree Traversing Problems

2 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; và 3 dạng biểu diễn: array of parents, first-child next-sibling, graph-based representation của cây.

Sẽ có $3A_4^4 + A_3^3 = 36 + 6 = 42$ converter programs.

3 Chuyển đổi đồ thị

3.1 Ma trận kề sang Danh sách kề

Listing 1: Chuyển đổi từ Ma trận kề sang Danh sách kề

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

// Convert adjacency matrix to adjacency list
vector<vector<int>> matrixToList(vector<vector<int>>& matrix) {
    int n = matrix.size();
    vector<vector<int>> adjList(n);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == 1) {
                adjList[i].push_back(j);
            }
        }
    }
    return adjList;
}

void printAdjList(const vector<vector<int>>& adjList) {
    cout << "Adjacency List:" << endl;
    for (int i = 0; i < adjList.size(); i++) {
        cout << i << ": ";
        for (int v : adjList[i]) {
            cout << v << " ";
        }
        cout << endl;
    }
}
```

```
int main() {
    // Example adjacency matrix
    vector<vector<int>> matrix = {
        {0, 1, 1, 0},
        {1, 0, 0, 1},
        {1, 0, 0, 1},
        {0, 1, 1, 0}
    };

    cout << "Original Adjacency Matrix:" << endl;
    for (auto& row : matrix) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    cout << endl;

    vector<vector<int>> adjList = matrixToList(matrix);
    printAdjList(adjList);

    return 0;
}
```

3.2 Danh sách kề sang Ma trận kề

Listing 2: Chuyển đổi từ Danh sách kề sang Ma trận kề

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

// Convert adjacency list to adjacency matrix
vector<vector<int>> listToMatrix(const vector<vector<int>>& adjList)
    {
    int n = adjList.size();
    vector<vector<int>> matrix(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {
        for (int v : adjList[i]) {
            matrix[i][v] = 1;
        }
    }
    return matrix;
}
```

```
void printMatrix(const vector<vector<int>>& matrix) {
    cout << "Adjacency Matrix:" << endl;
    for (const auto& row : matrix) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
}

int main() {
    // Example adjacency list
    vector<vector<int>> adjList = {
        {1, 2},     // vertex 0 connects to 1, 2
        {0, 3},     // vertex 1 connects to 0, 3
        {0, 3},     // vertex 2 connects to 0, 3
        {1, 2}      // vertex 3 connects to 1, 2
    };

    cout << "Original Adjacency List:" << endl;
    for (int i = 0; i < adjList.size(); i++) {
        cout << i << ": ";
        for (int v : adjList[i]) {
            cout << v << " ";
        }
        cout << endl;
    }
    cout << endl;

    vector<vector<int>> matrix = listToMatrix(adjList);
    printMatrix(matrix);

    return 0;
}
```

3.3 Danh sách kề sang Danh sách kề mở rộng

Listing 3: Chuyển đổi từ Danh sách kề sang Danh sách kề mở rộng (có trọng số)

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

struct Edge {
    int vertex;
    int weight;
```

```
};

// Convert adjacency list to extended adjacency list with weights
vector<vector<Edge>> listToExtendedList(const vector<vector<int>>&
   adjList) {
    vector<vector<Edge>> extendedList(adjList.size());
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(1, 10);

    for (int i = 0; i < adjList.size(); i++) {
        for (int v : adjList[i]) {
            Edge edge = {v, dis(gen)}; // random weight 1-10
            extendedList[i].push_back(edge);
        }
    }
    return extendedList;
}

void printExtendedList(const vector<vector<Edge>>& extList) {
    cout << "Extended Adjacency List (with weights):" << endl;
    for (int i = 0; i < extList.size(); i++) {
        cout << i << ": ";
        for (const Edge& e : extList[i]) {
            cout << "(" << e.vertex << "," << e.weight << ") ";
        }
        cout << endl;
    }
}

int main() {
    // Example adjacency list
    vector<vector<int>> adjList = {
        {1, 2},
        {0, 3},
        {0, 3},
        {1, 2}
    };

    cout << "Original Adjacency List:" << endl;
    for (int i = 0; i < adjList.size(); i++) {
        cout << i << ": ";
        for (int v : adjList[i]) {
            cout << v << " ";
        }
        cout << endl;
    }
```

```
    cout << endl;

    vector<vector<Edge>> extList = listToExtendedList(adjList);
    printExtendedList(extList);

    return 0;
}
```

3.4 Danh sách kề mở rộng sang Danh sách kề

Listing 4: Chuyển đổi từ Danh sách kề mở rộng sang Danh sách kề

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

struct Edge {
    int vertex;
    int weight;
};

// Convert extended adjacency list to simple adjacency list
vector<vector<int>> extendedListToList(const vector<vector<Edge>>&
   extList) {
    vector<vector<int>> adjList(extList.size());

    for (int i = 0; i < extList.size(); i++) {
        for (const Edge& e : extList[i]) {
            adjList[i].push_back(e.vertex);
        }
    }
    return adjList;
}

void printAdjList(const vector<vector<int>>& adjList) {
    cout << "Simple Adjacency List:" << endl;
    for (int i = 0; i < adjList.size(); i++) {
        cout << i << ": ";
        for (int v : adjList[i]) {
            cout << v << " ";
        }
        cout << endl;
    }
}

int main() {
    // Example extended adjacency list
```

```
    vector<vector<Edge>> extList = {
        {{1, 5}, {2, 3}},
        {{0, 2}, {3, 7}},
        {{0, 4}, {3, 1}},
        {{1, 6}, {2, 8}}
    };

    cout << "Original Extended Adjacency List:" << endl;
    for (int i = 0; i < extList.size(); i++) {
        cout << i << ": ";
        for (const Edge& e : extList[i]) {
            cout << "(" << e.vertex << "," << e.weight << ") ";
        }
        cout << endl;
    }
    cout << endl;

    vector<vector<int>> adjList = extendedListToList(extList);
    printAdjList(adjList);

    return 0;
}
```

3.5 Danh sách kề sang Adjacency Map

Listing 5: Chuyển đổi từ Danh sách kề sang Adjacency Map

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

// Convert adjacency list to adjacency map
map<int, set<int>> listToMap(const vector<vector<int>>& adjList) {
    map<int, set<int>> adjMap;

    for (int i = 0; i < adjList.size(); i++) {
        for (int v : adjList[i]) {
            adjMap[i].insert(v);
        }
    }
    return adjMap;
}

void printAdjMap(const map<int, set<int>>& adjMap) {
    cout << "Adjacency Map:" << endl;
```

```
    for (const auto& pair : adjMap) {
        cout << pair.first << ": {";
        bool first = true;
        for (int v : pair.second) {
            if (!first) cout << ", ";
            cout << v;
            first = false;
        }
        cout << "}" << endl;
    }
}

int main() {
    // Example adjacency list
    vector<vector<int>> adjList = {
        {1, 2},
        {0, 3, 2},
        {0, 3, 1},
        {1, 2}
    };

    cout << "Original Adjacency List:" << endl;
    for (int i = 0; i < adjList.size(); i++) {
        cout << i << ": ";
        for (int v : adjList[i]) {
            cout << v << " ";
        }
        cout << endl;
    }
    cout << endl;

    map<int, set<int>> adjMap = listToMap(adjList);
    printAdjMap(adjMap);

    return 0;
}
```

3.6 Adjacency Map sang Danh sách kề

Listing 6: Chuyển đổi từ Adjacency Map sang Danh sách kề

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

```cpp
// Convert adjacency map to adjacency list
vector<vector<int>> mapToList(const map<int, set<int>>& adjMap) {
    int maxVertex = 0;
    for (const auto& pair : adjMap) {
        maxVertex = max(maxVertex, pair.first);
        for (int v : pair.second) {
            maxVertex = max(maxVertex, v);
        }
    }

    vector<vector<int>> adjList(maxVertex + 1);
    for (const auto& pair : adjMap) {
        for (int v : pair.second) {
            adjList[pair.first].push_back(v);
        }
    }
    return adjList;
}

void printAdjList(const vector<vector<int>>& adjList) {
    cout << "Adjacency List:" << endl;
    for (int i = 0; i < adjList.size(); i++) {
        cout << i << ": ";
        for (int v : adjList[i]) {
            cout << v << " ";
        }
        cout << endl;
    }
}

int main() {
    // Example adjacency map
    map<int, set<int>> adjMap = {
        {0, {1, 2}},
        {1, {0, 2, 3}},
        {2, {0, 1, 3}},
        {3, {1, 2}}
    };

    cout << "Original Adjacency Map:" << endl;
    for (const auto& pair : adjMap) {
        cout << pair.first << ": {";
        bool first = true;
        for (int v : pair.second) {
            if (!first) cout << ", ";
            cout << v;
            first = false;
```

```
        }
        cout << "}" << endl;
    }
    cout << endl;

    vector<vector<int>> adjList = mapToList(adjMap);
    printAdjList(adjList);

    return 0;
}
```

4 Chuyển đổi cây

4.1 Array of Parents sang First-Child Next-Sibling

Listing 7: Chuyển đổi từ Array of Parents sang First-Child Next-Sibling

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

struct TreeNode {
    int data;
    int firstChild;
    int nextSibling;

    TreeNode(int val) : data(val), firstChild(-1), nextSibling(-1)
       {}
};

// Convert array of parents to first-child next-sibling
   representation
vector<TreeNode> parentsToFCNS(const vector<int>& parents) {
    int n = parents.size();
    vector<TreeNode> nodes;
    for (int i = 0; i < n; i++) {
        nodes.push_back(TreeNode(i));
    }

    // Build children lists for each node
    vector<vector<int>> children(n);
    for (int i = 0; i < n; i++) {
        if (parents[i] != -1) {
            children[parents[i]].push_back(i);
        }
    }
```

```
    // Set first child and next sibling relationships
    for (int i = 0; i < n; i++) {
        if (!children[i].empty()) {
            nodes[i].firstChild = children[i][0];

            // Link siblings
            for (int j = 0; j < children[i].size() - 1; j++) {
                int child = children[i][j];
                int nextChild = children[i][j + 1];
                nodes[child].nextSibling = nextChild;
            }
        }
    }

    return nodes;
}

void printFCNS(const vector<TreeNode>& nodes) {
    cout << "First-Child Next-Sibling representation:" << endl;
    cout << "Node | FirstChild | NextSibling" << endl;
    cout << "-----|------------|------------" << endl;
    for (const TreeNode& node : nodes) {
        cout << "  " << node.data << "  |";
        if (node.firstChild != -1) {
            cout << "      " << node.firstChild << "       |";
        } else {
            cout << "    null     |";
        }
        if (node.nextSibling != -1) {
            cout << "      " << node.nextSibling;
        } else {
            cout << "    null";
        }
        cout << endl;
    }
}

int main() {
    // Example: Tree with parent array
    // Tree structure:  0
    //                 /|\
    //                1 2 3
    //               /|   |
    //              4 5   6
    vector<int> parents = {-1, 0, 0, 0, 1, 1, 3};
```

```
    cout << "Original Parent Array:" << endl;
    cout << "Index: ";
    for (int i = 0; i < parents.size(); i++) {
        cout << i << " ";
    }
    cout << endl << "Parent: ";
    for (int p : parents) {
        if (p == -1) cout << "- ";
        else cout << p << " ";
    }
    cout << endl << endl;

    vector<TreeNode> fcns = parentsToFCNS(parents);
    printFCNS(fcns);

    return 0;
}
```

4.2 First-Child Next-Sibling sang Array of Parents

Listing 8: Chuyển đổi từ First-Child Next-Sibling sang Array of Parents

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

struct TreeNode {
    int data;
    int firstChild;
    int nextSibling;

    TreeNode(int val) : data(val), firstChild(-1), nextSibling(-1)
       {}
};

// Convert first-child next-sibling to array of parents
vector<int> fcnsToParents(const vector<TreeNode>& nodes) {
    int n = nodes.size();
    vector<int> parents(n, -1);

    for (int i = 0; i < n; i++) {
        int child = nodes[i].firstChild;
        while (child != -1) {
            parents[child] = i;
            child = nodes[child].nextSibling;
        }
    }
```

```
    return parents;
}

void printParentArray(const vector<int>& parents) {
    cout << "Parent Array:" << endl;
    cout << "Index:  ";
    for (int i = 0; i < parents.size(); i++) {
        cout << i << " ";
    }
    cout << endl << "Parent: ";
    for (int p : parents) {
        if (p == -1) cout << "- ";
        else cout << p << " ";
    }
    cout << endl;
}

int main() {
    // Example first-child next-sibling representation
    vector<TreeNode> nodes = {
        TreeNode(0), TreeNode(1), TreeNode(2),
        TreeNode(3), TreeNode(4), TreeNode(5), TreeNode(6)
    };

    // Set up relationships manually for demonstration
    nodes[0].firstChild = 1;    // 0's first child is 1
    nodes[1].nextSibling = 2;   // 1's next sibling is 2
    nodes[2].nextSibling = 3;   // 2's next sibling is 3
    nodes[1].firstChild = 4;    // 1's first child is 4
    nodes[4].nextSibling = 5;   // 4's next sibling is 5
    nodes[3].firstChild = 6;    // 3's first child is 6

    cout << "Original First-Child Next-Sibling:" << endl;
    cout << "Node | FirstChild | NextSibling" << endl;
    cout << "-----|------------|------------" << endl;
    for (const TreeNode& node : nodes) {
        cout << "  " << node.data << "  |";
        if (node.firstChild != -1) {
            cout << "     " << node.firstChild << "      |";
        } else {
            cout << "    null     |";
        }
        if (node.nextSibling != -1) {
            cout << "     " << node.nextSibling;
        } else {
            cout << "    null";
```

```
        }
        cout << endl;
    }
    cout << endl;

    vector<int> parents = fcnsToParents(nodes);
    printParentArray(parents);

    return 0;
}
```

4.3 Array of Parents sang Graph-based Representation

Listing 9: Chuyển đổi từ Array of Parents sang Graph-based Representation

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

// Convert array of parents to graph-based representation (adjacency
    list)
vector<vector<int>> parentsToGraph(const vector<int>& parents) {
    int n = parents.size();
    vector<vector<int>> graph(n);

    for (int i = 0; i < n; i++) {
        if (parents[i] != -1) {
            // Add bidirectional edge between parent and child
            graph[parents[i]].push_back(i);
            graph[i].push_back(parents[i]);
        }
    }

    return graph;
}

void printGraph(const vector<vector<int>>& graph) {
    cout << "Graph-based representation (adjacency list):" << endl;
    for (int i = 0; i < graph.size(); i++) {
        cout << "Node " << i << ": ";
        for (int neighbor : graph[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
}
```

```
int main() {
    // Example parent array
    vector<int> parents = {-1, 0, 0, 0, 1, 1, 3};

    cout << "Original Parent Array:" << endl;
    cout << "Index:  ";
    for (int i = 0; i < parents.size(); i++) {
        cout << i << " ";
    }
    cout << endl << "Parent: ";
    for (int p : parents) {
        if (p == -1) cout << "- ";
        else cout << p << " ";
    }
    cout << endl << endl;

    vector<vector<int>> graph = parentsToGraph(parents);
    printGraph(graph);

    return 0;
}
```

4.4 Graph-based Representation sang Array of Parents

Listing 10: Chuyển đổi từ Graph-based Representation sang Array of Parents

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

// Convert graph-based representation to array of parents using BFS
vector<int> graphToParents(const vector<vector<int>>& graph, int
   root) {
    int n = graph.size();
    vector<int> parents(n, -1);
    vector<bool> visited(n, false);
    queue<int> q;

    q.push(root);
    visited[root] = true;

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

        for (int neighbor : graph[current]) {
```

```
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                parents[neighbor] = current;
                q.push(neighbor);
            }
        }
    }

    return parents;
}

void printParentArray(const vector<int>& parents) {
    cout << "Parent Array:" << endl;
    cout << "Index:  ";
    for (int i = 0; i < parents.size(); i++) {
        cout << i << " ";
    }
    cout << endl << "Parent: ";
    for (int p : parents) {
        if (p == -1) cout << "- ";
        else cout << p << " ";
    }
    cout << endl;
}

int main() {
    // Example graph representation (tree as undirected graph)
    vector<vector<int>> graph = {
        {1, 2, 3},      // node 0 connects to 1, 2, 3
        {0, 4, 5},      // node 1 connects to 0, 4, 5
        {0},            // node 2 connects to 0
        {0, 6},         // node 3 connects to 0, 6
        {1},            // node 4 connects to 1
        {1},            // node 5 connects to 1
        {3}             // node 6 connects to 3
    };

    cout << "Original Graph-based representation:" << endl;
    for (int i = 0; i < graph.size(); i++) {
        cout << "Node " << i << ": ";
        for (int neighbor : graph[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
    cout << endl;
```

```
    // Convert to parent array with root = 0
    vector<int> parents = graphToParents(graph, 0);
    printParentArray(parents);

    return 0;
}
```

4.5 First-Child Next-Sibling sang Graph-based Representation

Listing 11: Chuyển đổi từ First-Child Next-Sibling sang Graph-based Representation

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

struct TreeNode {
    int data;
    int firstChild;
    int nextSibling;

    TreeNode(int val) : data(val), firstChild(-1), nextSibling(-1)
       {}
};

// Convert first-child next-sibling to graph representation
vector<vector<int>> fcnsToGraph(const vector<TreeNode>& nodes) {
    int n = nodes.size();
    vector<vector<int>> graph(n);

    for (int i = 0; i < n; i++) {
        int child = nodes[i].firstChild;
        while (child != -1) {
            // Add bidirectional edge between parent and child
            graph[i].push_back(child);
            graph[child].push_back(i);
            child = nodes[child].nextSibling;
        }
    }

    return graph;
}

void printGraph(const vector<vector<int>>& graph) {
    cout << "Graph-based representation:" << endl;
    for (int i = 0; i < graph.size(); i++) {
        cout << "Node " << i << ": ";
        for (int neighbor : graph[i]) {
```

```
            cout << neighbor << " ";
        }
        cout << endl;
    }
}

int main() {
    // Example first-child next-sibling representation
    vector<TreeNode> nodes = {
        TreeNode(0), TreeNode(1), TreeNode(2),
        TreeNode(3), TreeNode(4), TreeNode(5), TreeNode(6)
    };

    // Set up relationships
    nodes[0].firstChild = 1;
    nodes[1].nextSibling = 2;
    nodes[2].nextSibling = 3;
    nodes[1].firstChild = 4;
    nodes[4].nextSibling = 5;
    nodes[3].firstChild = 6;

    cout << "Original First-Child Next-Sibling:" << endl;
    cout << "Node | FirstChild | NextSibling" << endl;
    cout << "-----|------------|------------" << endl;
    for (const TreeNode& node : nodes) {
        cout << "  " << node.data << "  |";
        if (node.firstChild != -1) {
            cout << "      " << node.firstChild << "       |";
        } else {
            cout << "    null      |";
        }
        if (node.nextSibling != -1) {
            cout << "      " << node.nextSibling;
        } else {
            cout << "    null";
        }
        cout << endl;
    }
    cout << endl;

    vector<vector<int>> graph = fcnsToGraph(nodes);
    printGraph(graph);

    return 0;
}
```

4.6 Graph-based Representation sang First-Child Next-Sibling

Listing 12: Chuyển đổi từ Graph-based Representation sang First-Child Next-Sibling

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

struct TreeNode {
    int data;
    int firstChild;
    int nextSibling;

    TreeNode(int val) : data(val), firstChild(-1), nextSibling(-1)
       {}
};

// Convert graph to first-child next-sibling using BFS to establish
   parent-child relationships
vector<TreeNode> graphToFCNS(const vector<vector<int>>& graph, int
   root) {
    int n = graph.size();
    vector<TreeNode> nodes;
    for (int i = 0; i < n; i++) {
        nodes.push_back(TreeNode(i));
    }

    vector<bool> visited(n, false);
    vector<vector<int>> children(n);
    queue<int> q;

    q.push(root);
    visited[root] = true;

    // Build parent-child relationships using BFS
    while (!q.empty()) {
        int current = q.front();
        q.pop();

        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                children[current].push_back(neighbor);
                q.push(neighbor);
            }
        }
```

```
    }

    // Convert to first-child next-sibling
    for (int i = 0; i < n; i++) {
        if (!children[i].empty()) {
            nodes[i].firstChild = children[i][0];

            // Link siblings
            for (int j = 0; j < children[i].size() - 1; j++) {
                int child = children[i][j];
                int nextChild = children[i][j + 1];
                nodes[child].nextSibling = nextChild;
            }
        }
    }

    return nodes;
}

void printFCNS(const vector<TreeNode>& nodes) {
    cout << "First-Child Next-Sibling representation:" << endl;
    cout << "Node | FirstChild | NextSibling" << endl;
    cout << "-----|------------|------------" << endl;
    for (const TreeNode& node : nodes) {
        cout << "  " << node.data << "  |";
        if (node.firstChild != -1) {
            cout << "      " << node.firstChild << "       |";
        } else {
            cout << "    null      |";
        }
        if (node.nextSibling != -1) {
            cout << "      " << node.nextSibling;
        } else {
            cout << "    null";
        }
        cout << endl;
    }
}

int main() {
    // Example graph representation (tree as undirected graph)
    vector<vector<int>> graph = {
        {1, 2, 3},      // node 0 connects to 1, 2, 3
        {0, 4, 5},      // node 1 connects to 0, 4, 5
        {0},            // node 2 connects to 0
        {0, 6},         // node 3 connects to 0, 6
        {1},            // node 4 connects to 1
```

```
        {1},            // node 5 connects to 1
        {3}             // node 6 connects to 3
    };

    cout << "Original Graph-based representation:" << endl;
    for (int i = 0; i < graph.size(); i++) {
        cout << "Node " << i << ": ";
        for (int neighbor : graph[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
    cout << endl;

    // Convert to FCNS with root = 0
    vector<TreeNode> fcns = graphToFCNS(graph, 0);
    printFCNS(fcns);

    return 0;
}
```

4.7 Bài toán 5

5 Bài 1.1: Kích thước của đồ thị đầy đủ và đồ thị hai phần đầy đủ

5.1 Đồ thị đầy đủ K_n

Trong đồ thị đầy đủ K_n, có một cạnh giữa mỗi đỉnh và mỗi đỉnh khác.

- Mỗi đỉnh kết nối với tất cả $(n-1)$ đỉnh còn lại
- Tổng số cặp đỉnh có thể tạo cạnh: $\binom{n}{2} = \frac{n(n-1)}{2}$

Đáp số: Kích thước của K_n là $\boxed{\frac{n(n-1)}{2} \text{ cạnh}}$

5.2 Đồ thị hai phần đầy đủ $K_{p,q}$

- Có tập đỉnh thứ nhất với p đỉnh và tập thứ hai với q đỉnh
- Có một cạnh giữa mỗi đỉnh trong tập thứ nhất và mỗi đỉnh trong tập thứ hai
- Mỗi đỉnh trong p đỉnh kết nối với tất cả q đỉnh

Đáp số: Kích thước của $K_{p,q}$ là $\boxed{pq \text{ cạnh}}$

6 Bài 1.2: Điều kiện đồ thị là hai phần

Nguyên lý: Một đồ thị là hai phần khi và chỉ khi nó không chứa chu trình có độ dài lẻ.

6.1 Chu trình C_n

- C_n chứa duy nhất một chu trình có độ dài n
- Nếu n lẻ thì C_n chứa chu trình lẻ $\Rightarrow$ không thể là hai phần
- Nếu n chẵn thì có thể chia đỉnh thành 2 tập: $\{v_1, v_3, v_5, \ldots\}$ và $\{v_2, v_4, v_6, \ldots\}$

Đáp số: C_n là hai phần khi và chỉ khi $\boxed{n \text{ chẵn}}$

6.2 Đồ thị đầy đủ K_n

- Với $n \geq 3$: K_n chứa tam giác (chu trình độ dài 3), là chu trình lẻ
- K_1: Chỉ có 1 đỉnh $\Rightarrow$ tầm thường là hai phần
- K_2: Có 2 đỉnh, 1 cạnh $\Rightarrow$ là hai phần với 2 tập $\{v_1\}, \{v_2\}$

Đáp số: K_n là hai phần khi và chỉ khi $\boxed{n = 1 \text{ hoặc } n = 2}$

7 Bài 1.3: Cây khung của đồ thị

Phương pháp: Sử dụng định lý Matrix-Tree để tính số cây khung

- Ma trận Laplacian: $L = D - A$ (D là ma trận bậc, A là ma trận kề)
- Số cây khung = $\det(L')$ với L' là ma trận con bỏ một hàng và một cột

Từ Fig. B.1 và solution:

- Các cây khung DFS và BFS được liệt kê với mỗi đỉnh làm gốc
- Rừng breadth-first không phải là cây khung vì không liên thông

Đáp số:

- Số cây khung của đồ thị có hướng: $\boxed{45}$
- Số cây khung của đồ thị vô hướng tương ứng: $\boxed{288}$

8 Bài 1.4: Mở rộng biểu diễn ma trận kề

Giả sử: Đồ thị $G = (V, E)$ có n đỉnh được đánh số $1, 2, \ldots, n$ và lưu trong thuộc tính index, A là ma trận kề.

Code các phép toán:

```
def del_edge(A, v, w):
    A[v.index][w.index] = False

def edges(A):
    edge_list = []
    for i in range(len(A)):
        for j in range(len(A)):
            if A[i][j]:
                edge_list.append((i, j))
    return edge_list

def incoming(A, v):
    return [u for u in range(len(A)) if A[u][v.index]]

def outgoing(A, v):
    return [w for w in range(len(A)) if A[v.index][w]]

def source(v, w):
    return v

def target(v, w):
    return w
```

9 Bài 1.5: Cải tiến biểu diễn cây first-child, next-sibling

Mục tiêu: Thực hiện các phép toán $T.root()$, $T.number\_of\_children(v)$, $T.children(v)$ trong thời gian $O(1)$.

Cấu trúc bổ sung:

- Mảng P: array-of-parents representation
- Mảng L: last-child array ($L[v]$ = con cuối cùng của v)
- Mảng S: previous-sibling array ($S[v]$ = anh/chị trước v)

Code các phép toán $O(1)$:

```
def last_child(T, v):
    return L[v]

def previous_sibling(T, v):
```

```
    return S[v]

def is_first_child(T, v):
    return S[v] == None
```

Đáp số: Ba phép toán còn lại có thể Code trong thời gian $O(1)$ nhờ các mảng bổ trợ.

10 Bài 1.6: Kiểm tra tính hợp lệ của cây

Nguyên lý: Một đồ thị liên thông là cây khi và chỉ khi số đỉnh bằng số cạnh cộng 1.

Thuật toán kiểm tra:

```
def is_tree(G):
    n = len(G)

    edge_count = 0
    for i in range(n):
        for j in range(n):
            if G[i][j]:
                edge_count += 1
    edge_count = edge_count // 2

    if edge_count != n - 1:
        return False

    visited = [False] * n
    stack = [0]
    visited[0] = True
    count = 1

    while stack:
        v = stack.pop()
        for u in range(n):
            if G[v][u] and not visited[u]:
                visited[u] = True
                stack.append(u)
                count += 1

    return count == n
```

Độ phức tạp: $O(n^2)$ để đếm cạnh + $O(n+m)$ cho DFS = $O(n^2)$

Đáp số: Thuật toán kiểm tra một đồ thị có phải cây hay không trong thời gian tuyến tính theo kích thu

11 Exercise 1.1

The standard representation of an undirected graph in the format adopted for the DIMACS Implementation Challenges 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.

Lời giải:

```
def read_dimacs_graph(filename):
    vertices = 0
    edges = []

    with open(filename, 'r') as f:
        for line in f:
            line = line.strip()
            if line.startswith('c'):
                continue
            elif line.startswith('p edge'):
                parts = line.split()
                vertices = int(parts[2])
                num_edges = int(parts[3])
            elif line.startswith('e'):
                parts = line.split()
                i, j = int(parts[1]), int(parts[2])
                edges.append((i, j))

    return vertices, edges

def write_dimacs_graph(vertices, edges, filename):
    with open(filename, 'w') as f:
        f.write(f"c DIMACS graph format\n")
        f.write(f"p edge {vertices} {len(edges)}\n")
        for i, j in edges:
            f.write(f"e {i} {j}\n")
```

Hàm `read_dimacs_graph` đọc file DIMACS bằng cách:

- Bỏ qua các dòng comment (bắt đầu với 'c')
- Đọc dòng định nghĩa bài toán để lấy số đỉnh và số cạnh
- Đọc các dòng mô tả cạnh để lấy danh sách các cạnh

Hàm `write_dimacs_graph` ghi đồ thị ra file theo đúng định dạng DIMACS với dòng comment, dòng định nghĩa bài toán và các dòng mô tả cạnh.

12 Exercise 1.2

The external representation of a graph in the Stanford GraphBase (SGB) format 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.

Lời giải:

```
def read_sgb_graph(filename):
    vertices = []
    edges = []

    with open(filename, 'r') as f:
        lines = f.readlines()

    i = 0
    header = lines[i].strip()
    parts = header.split('(')[1].split(')')[0].split(',')
    n_vertices = int(parts[-2].strip())
    n_edges = int(parts[-1].strip())

    i += 1
    identification = lines[i].strip()

    i += 2
    for _ in range(n_vertices):
        vertex_line = lines[i].strip()
        parts = vertex_line.split(',')
        label = parts[0].strip()
        first_edge = parts[1].strip()
        vertices.append({'label': label, 'first_edge': first_edge})
        i += 1

    i += 1
    for _ in range(n_edges):
        edge_line = lines[i].strip()
        parts = edge_line.split(',')
        target_vertex = int(parts[0][1:])
```

```
            next_edge = parts[1].strip()
            edge_label = int(parts[2].strip())
            edges.append({
                'target': target_vertex,
                'next_edge': next_edge,
                'label': edge_label
            })
            i += 1

    return vertices, edges

def write_sgb_graph(vertices, edges, identification="Graph",
   filename="output.sgb"):
    with open(filename, 'w') as f:
            f.write(f"* GraphBase graph(util_types..., {len(vertices)},
               {len(edges)})\n")
            f.write(f"{identification}\n")
            f.write("* Vertices\n")

            for i, vertex in enumerate(vertices):
                first_edge = vertex.get('first_edge', '0')
                f.write(f"{vertex['label']}, {first_edge}, 0, 0\n")

            f.write("* Arcs\n")
            for edge in edges:
                next_edge = edge.get('next_edge', '0')
                f.write(f"V{edge['target']}, {next_edge}, {edge['label
                   ']}, 0\n")

            f.write("* Checksum 0\n")
```

Định dạng SGB phức tạp hơn DIMACS vì nó lưu trữ thông tin về thứ tự các cạnh xuất phát từ mỗi đỉnh. Hàm đọc phân tích từng phần của file theo cấu trúc: header, identification, vertices, arcs. Hàm ghi tạo ra file theo đúng định dạng SGB với tất cả các thành phần cần thiết.

13 Exercise 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.

Lời giải:

```
class Graph:
    def __init__(self):
        self.vertices = set()
        self.edges = set()
```

```
    def add_vertex(self, v):
        self.vertices.add(v)

    def add_edge(self, u, v):
        self.vertices.add(u)
        self.vertices.add(v)
        self.edges.add((min(u,v), max(u,v)))

def generate_path_graph(n):
    G = Graph()

    for i in range(1, n+1):
        G.add_vertex(i)

    for i in range(1, n):
        G.add_edge(i, i+1)

    return G

def generate_cycle_graph(n):
    G = Graph()

    for i in range(1, n+1):
        G.add_vertex(i)

    for i in range(1, n):
        G.add_edge(i, i+1)
    G.add_edge(n, 1)

    return G

def generate_wheel_graph(n):
    G = Graph()

    center = n
    for i in range(1, n+1):
        G.add_vertex(i)

    for i in range(1, n):
        G.add_edge(i, i+1)
    G.add_edge(n-1, 1)

    for i in range(1, n):
        G.add_edge(center, i)

    return G
```

- **Path graph** P_n: Tạo n đỉnh và n-1 cạnh nối liên tiếp từ đỉnh 1 đến n
- **Cycle graph** C_n: Giống path graph nhưng thêm cạnh nối đỉnh cuối với đỉnh đầu
- **Wheel graph** W_n: Tạo cycle với n-1 đỉnh, thêm 1 đỉnh trung tâm và nối với tất cả đỉnh khác

14 Exercise 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.

Lời giải:

```
def generate_complete_graph(n):
    G = Graph()

    for i in range(1, n+1):
        G.add_vertex(i)

    for i in range(1, n+1):
        for j in range(i+1, n+1):
            G.add_edge(i, j)

    return G

def generate_complete_bipartite_graph(p, q):
    G = Graph()

    for i in range(1, p+1):
        G.add_vertex(i)
    for i in range(p+1, p+q+1):
        G.add_vertex(i)

    for i in range(1, p+1):
        for j in range(p+1, p+q+1):
            G.add_edge(i, j)

    return G
```

Đồ thị đầy đủ K_n được tạo bằng cách nối mọi cặp đỉnh với nhau. Đồ thị hai phần đầy đủ $K_{p,q}$ chia đỉnh thành 2 tập (1 đến p) và (p+1 đến p+q), sau đó nối mọi đỉnh tập thứ nhất với mọi đỉnh tập thứ hai.

15 Exercise 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.

Lời giải:

```
class ExtendedAdjacencyMatrix:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.matrix = [[False] * num_vertices for _ in range(
           num_vertices)]
        self.vertex_indices = {i: i for i in range(num_vertices)}

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

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

    def edges(self):
        edge_list = []
        for i in range(self.num_vertices):
            for j in range(i+1, self.num_vertices):
                if self.matrix[i][j]:
                    edge_list.append((i, j))
        return edge_list

    def incoming(self, v):
        return [u for u in range(self.num_vertices) if self.matrix[u
           ][v]]

    def outgoing(self, v):
        return [u for u in range(self.num_vertices) if self.matrix[v
           ][u]]

    def source(self, edge):
        return edge[0]

    def target(self, edge):
        return edge[1]
```

Class này mở rộng ma trận kề cơ bản với các phép toán bổ sung như xóa cạnh, liệt kê tất cả cạnh, tìm đỉnh kề vào/ra. Ma trận được lưu dưới dạng danh sách 2 chiều với đánh số nội bộ từ 0 đến n-1.

16 Exercise 1.6

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

Lời giải:

```
def enumerate_perfect_matchings_bipartite(p, q):
    if p != q:
        return []

    n = p
    set_A = list(range(n))
    set_B = list(range(n, 2*n))

    def generate_permutations(arr):
        if len(arr) <= 1:
            return [arr]
        result = []
        for i in range(len(arr)):
            rest = arr[:i] + arr[i+1:]
            for perm in generate_permutations(rest):
                result.append([arr[i]] + perm)
        return result

    all_matchings = []
    permutations = generate_permutations(set_B)

    for perm in permutations:
        matching = []
        for i in range(n):
            matching.append((set_A[i], perm[i]))
        all_matchings.append(matching)

    return all_matchings

def count_perfect_matchings(p, q):
    if p != q:
        return 0
    return factorial(p)

def factorial(n):
    if n <= 1:
        return 1
```

```
    return n * factorial(n-1)
```

Perfect matching chỉ tồn tại khi $p = q$. Trong trường hợp này, số perfect matching bằng $p!$ vì ta cần ghép mỗi đỉnh trong tập A với đúng một đỉnh trong tập B. Thuật toán liệt kê tất cả hoán vị của tập B và tạo ra các matching tương ứng.

17 Exercise 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.

Lời giải:

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.parent = None

class CompleteBinaryTree:
    def __init__(self):
        self.root = None
        self.nodes = []

    def generate_complete_binary_tree(self, n):
        if n == 0:
            return None

        self.root = TreeNode(1)
        self.nodes = [self.root]

        for i in range(2, n+1):
            new_node = TreeNode(i)
            parent_index = (i - 2) // 2
            parent = self.nodes[parent_index]

            if parent.left is None:
                parent.left = new_node
            else:
                parent.right = new_node

            new_node.parent = parent
            self.nodes.append(new_node)

        return self.root

    def get_parent(self, node):
```

```
        return node.parent

    def get_left_child(self, node):
        return node.left

    def get_right_child(self, node):
        return node.right

    def is_leaf(self, node):
        return node.left is None and node.right is None
```

Complete binary tree được tạo theo cấp độ từ trái sang phải. Node thứ i có parent là node thứ $\lfloor (i-1)/2 \rfloor$. Thuật toán duy trì danh sách các node để dễ dàng tìm parent khi thêm node mới.

18 Exercise 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.

Lời giải:

```
import random

def generate_random_tree_prufer(n):
    if n == 1:
        return TreeNode(1)

    sequence = [random.randint(1, n) for _ in range(n-2)]

    degree = [1] * (n+1)
    for i in sequence:
        degree[i] += 1

    edges = []
    for i in sequence:
        for j in range(1, n+1):
            if degree[j] == 1:
                edges.append((i, j))
                degree[i] -= 1
                degree[j] -= 1
                break

    remaining = [i for i in range(1, n+1) if degree[i] == 1]
    if len(remaining) == 2:
        edges.append((remaining[0], remaining[1]))
```

```
    root = TreeNode(1)
    nodes = {1: root}

    for i in range(2, n+1):
        nodes[i] = TreeNode(i)

    for u, v in edges:
        if u not in nodes[v].children:
            if not hasattr(nodes[v], 'children'):
                nodes[v].children = []
            nodes[v].children.append(nodes[u])
            nodes[u].parent = nodes[v]

    return root

def generate_random_tree_simple(n):
    nodes = [TreeNode(i) for i in range(1, n+1)]

    if n == 1:
        return nodes[0]

    root = nodes[0]

    for i in range(1, n):
        parent_index = random.randint(0, i-1)
        parent = nodes[parent_index]

        if not hasattr(parent, 'children'):
            parent.children = []
        parent.children.append(nodes[i])
        nodes[i].parent = parent

    return root
```

Phân tích độ phức tạp:

- **Thời gian**: $O(n^2)$ cho thuật toán đơn giản, $O(n)$ cho Prüfer sequence
- **Không gian**: $O(n)$ để lưu trữ các node

Thuật toán Prüfer sequence tạo ra mỗi cây có nhãn với xác suất đều nhau $\frac{1}{n^{n-2}}$. Thuật toán đơn giản nhanh hơn nhưng không đảm bảo phân bố đều.

19 Exercise 1.9

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

Lời giải:

```python
class ArrayOfParentsTree:
    def __init__(self, n):
        self.parent = [None] * (n+1)
        self.children = [[] for _ in range(n+1)]
        self.n = n

    def add_edge(self, parent, child):
        self.parent[child] = parent
        if parent is not None:
            self.children[parent].append(child)

    def previous_sibling(self, v):
        if self.parent[v] is None:
            return None

        parent = self.parent[v]
        siblings = self.children[parent]

        try:
            index = siblings.index(v)
            if index > 0:
                return siblings[index - 1]
            else:
                return None
        except ValueError:
            return None

    def next_sibling(self, v):
        if self.parent[v] is None:
            return None

        parent = self.parent[v]
        siblings = self.children[parent]

        try:
            index = siblings.index(v)
            if index < len(siblings) - 1:
                return siblings[index + 1]
            else:
                return None
        except ValueError:
            return None
```

Phép toán `previous_sibling` được thực hiện bằng cách:

- Tìm parent của node v
- Lấy danh sách con của parent

- Tìm vị trí của v trong danh sách
- Trả về node trước đó nếu tồn tại

Độ phức tạp thời gian: $O(k)$ với k là số con của parent.

20 Exercise 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.

Lời giải:

```
class ExtendedFirstChildNextSibling:
    def __init__(self, n):
        self.n = n
        self.first_child = [None] * (n+1)
        self.next_sibling = [None] * (n+1)
        self.parent = [None] * (n+1)
        self.last_child = [None] * (n+1)
        self.previous_sibling = [None] * (n+1)
        self.num_children = [0] * (n+1)
        self.root_node = None

    def set_root(self, root):
        self.root_node = root

    def add_child(self, parent, child):
        self.parent[child] = parent
        self.num_children[parent] += 1

        if self.first_child[parent] is None:
            self.first_child[parent] = child
            self.last_child[parent] = child
        else:
            current_last = self.last_child[parent]
            self.next_sibling[current_last] = child
            self.previous_sibling[child] = current_last
            self.last_child[parent] = child

    def root(self):
        return self.root_node

    def number_of_children(self, v):
        return self.num_children[v]

    def children(self, v):
        result = []
```

```
        current = self.first_child[v]
        while current is not None:
            result.append(current)
            current = self.next_sibling[current]
        return result

    def get_parent(self, v):
        return self.parent[v]

    def get_first_child(self, v):
        return self.first_child[v]

    def get_next_sibling(self, v):
        return self.next_sibling[v]

    def get_last_child(self, v):
        return self.last_child[v]

    def get_previous_sibling(self, v):
        return self.previous_sibling[v]

    def is_leaf(self, v):
        return self.first_child[v] is None

    def is_first_child(self, v):
        return self.previous_sibling[v] is None and self.parent[v]
           is not None

    def is_last_child(self, v):
        return self.next_sibling[v] is None and self.parent[v] is
           not None
```

Class mở rộng bao gồm các mảng bổ trợ:

- `last_child`: Con cuối cùng của mỗi node
- `previous_sibling`: Anh/chị trước của mỗi node
- `num_children`: Số lượng con của mỗi node

Các phép toán `root()`, `number_of_children()`, và các phép toán khác đều được thực hiện trong thời gian $O(1)$ nhờ các mảng bổ trợ này.

21 Bài 6

Viết chương trình C/C++, Python để giải bài toán tree edit distance problem bằng cách sử dụng:

(a) Backtracking - Quay lui

(b) Branch Bound - Nhánh cận

(c) Divide Conquer - Chia để trị

(d) Dynamic Programming - Quy hoạch động

22 Code Implementation

Listing 13: Complete Tree Edit Distance Implementation

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <string>
#include <climits>
#include <queue>

using namespace std;

// Tree node structure
struct TreeNode {
    char label;
    vector<TreeNode*> children;
    int id;

    TreeNode(char c, int _id = 0) : label(c), id(_id) {}

    void addChild(TreeNode* child) {
        children.push_back(child);
    }
};

class TreeEditDistance {
private:
    // Operation costs
    int insertCost = 1;
    int deleteCost = 1;
    int substituteCost = 1;

    // Memory for dynamic programming
    unordered_map<string, int> memo;

    // Helper functions
    string encodeSubtree(TreeNode* node);
```

```
    vector<TreeNode*> postOrder(TreeNode* root);
    int getSubtreeSize(TreeNode* node);

public:
    // (a) Backtracking approach
    int backtrackingTED(TreeNode* tree1, TreeNode* tree2);

    // (b) Branch and Bound approach
    int branchBoundTED(TreeNode* tree1, TreeNode* tree2);

    // (c) Divide and Conquer approach
    int divideConquerTED(TreeNode* tree1, TreeNode* tree2);

    // (d) Dynamic Programming approach
    int dynamicProgrammingTED(TreeNode* tree1, TreeNode* tree2);

    // Utility functions
    void printTree(TreeNode* root, int depth = 0);
    TreeNode* buildSampleTree1();
    TreeNode* buildSampleTree2();
    TreeNode* buildTreeFromInput();
    void deleteTree(TreeNode* root);
    void clearMemo() { memo.clear(); }
};

// ==================== BACKTRACKING APPROACH ====================
int TreeEditDistance::backtrackingTED(TreeNode* tree1, TreeNode*
   tree2) {
    if (!tree1 andand !tree2) return 0;
    if (!tree1) return getSubtreeSize(tree2) * insertCost;
    if (!tree2) return getSubtreeSize(tree1) * deleteCost;

    int minCost = INT_MAX;

    // Case 1: Delete node from tree1
    minCost = min(minCost, deleteCost + backtrackingTED(nullptr,
       tree2));
    for (TreeNode* child : tree1->children) {
        minCost = min(minCost, deleteCost + backtrackingTED(child,
           tree2));
    }

    // Case 2: Insert node to tree1
    minCost = min(minCost, insertCost + backtrackingTED(tree1,
       nullptr));
    for (TreeNode* child : tree2->children) {
```

```
        minCost = min(minCost, insertCost + backtrackingTED(tree1,
            child));
    }

    // Case 3: Match/Substitute
    int matchCost = (tree1->label == tree2->label) ? 0 :
        substituteCost;

    // Try all possible mappings between children
    vector<TreeNode*> children1 = tree1->children;
    vector<TreeNode*> children2 = tree2->children;

    if (children1.empty() andand children2.empty()) {
        minCost = min(minCost, matchCost);
    } else {
        // Simplified: map children in order (for demonstration)
        int childrenCost = 0;
        int i = 0, j = 0;

        while (i < children1.size() andand j < children2.size()) {
            childrenCost += backtrackingTED(children1[i], children2[
                j]);
            i++; j++;
        }

        while (i < children1.size()) {
            childrenCost += deleteCost * getSubtreeSize(children1[i
                ]);
            i++;
        }

        while (j < children2.size()) {
            childrenCost += insertCost * getSubtreeSize(children2[j
                ]);
            j++;
        }

        minCost = min(minCost, matchCost + childrenCost);
    }

    return minCost;
}

// ==================== BRANCH and BOUND APPROACH
    ====================
int TreeEditDistance::branchBoundTED(TreeNode* tree1, TreeNode*
    tree2) {
```

```
    // Use lower bound for pruning
    if (!tree1 andand !tree2) return 0;
    if (!tree1) return getSubtreeSize(tree2) * insertCost;
    if (!tree2) return getSubtreeSize(tree1) * deleteCost;

    // Lower bound: difference in tree sizes
    int size1 = getSubtreeSize(tree1);
    int size2 = getSubtreeSize(tree2);
    int lowerBound = abs(size1 - size2);

    // Check memoization
    string key = encodeSubtree(tree1) + "|" + encodeSubtree(tree2);
    if (memo.find(key) != memo.end()) {
        return memo[key];
    }

    int minCost = INT_MAX;

    // Similar to backtracking but with pruning
    if (tree1->label == tree2->label) {
        int childrenCost = 0;
        vector<TreeNode*> children1 = tree1->children;
        vector<TreeNode*> children2 = tree2->children;

        int i = 0, j = 0;
        while (i < children1.size() andand j < children2.size()) {
            childrenCost += branchBoundTED(children1[i], children2[j
                ]);
            i++; j++;
        }

        while (i < children1.size()) {
            childrenCost += deleteCost * getSubtreeSize(children1[i
                ]);
            i++;
        }

        while (j < children2.size()) {
            childrenCost += insertCost * getSubtreeSize(children2[j
                ]);
            j++;
        }

        minCost = childrenCost;
    } else {
        // Substitute
        int substituteCostTotal = substituteCost;
```

```
        vector<TreeNode*> children1 = tree1->children;
        vector<TreeNode*> children2 = tree2->children;

        int i = 0, j = 0;
        while (i < children1.size() andand j < children2.size()) {
            substituteCostTotal += branchBoundTED(children1[i],
               children2[j]);
            i++; j++;
        }

        while (i < children1.size()) {
            substituteCostTotal += deleteCost * getSubtreeSize(
               children1[i]);
            i++;
        }

        while (j < children2.size()) {
            substituteCostTotal += insertCost * getSubtreeSize(
               children2[j]);
            j++;
        }

        minCost = min(minCost, substituteCostTotal);

        // Delete tree1
        minCost = min(minCost, deleteCost * size1 + insertCost *
           size2);
    }

    memo[key] = minCost;
    return minCost;
}

// ==================== DIVIDE and CONQUER APPROACH
   ====================
int TreeEditDistance::divideConquerTED(TreeNode* tree1, TreeNode*
   tree2) {
    if (!tree1 andand !tree2) return 0;
    if (!tree1) return getSubtreeSize(tree2) * insertCost;
    if (!tree2) return getSubtreeSize(tree1) * deleteCost;

    string key = encodeSubtree(tree1) + "|" + encodeSubtree(tree2);
    if (memo.find(key) != memo.end()) {
        return memo[key];
    }

    int result;
```

```
    if (tree1->label == tree2->label) {
        // Divide: compute for subtrees
        int childrenCost = 0;

        // Simplified: map children in order
        int maxChildren = max(tree1->children.size(), tree2->
           children.size());

        for (int i = 0; i < maxChildren; i++) {
            TreeNode* child1 = (i < tree1->children.size()) ? tree1
               ->children[i] : nullptr;
            TreeNode* child2 = (i < tree2->children.size()) ? tree2
               ->children[i] : nullptr;

            childrenCost += divideConquerTED(child1, child2);
        }

        result = childrenCost;
    } else {
        // Try 3 options: delete, insert, substitute
        int option1 = deleteCost + divideConquerTED(nullptr, tree2);
        for (TreeNode* child : tree1->children) {
            option1 += divideConquerTED(child, nullptr);
        }

        int option2 = insertCost + divideConquerTED(tree1, nullptr);
        for (TreeNode* child : tree2->children) {
            option2 += divideConquerTED(nullptr, child);
        }

        int option3 = substituteCost;
        int maxChildren = max(tree1->children.size(), tree2->
           children.size());
        for (int i = 0; i < maxChildren; i++) {
            TreeNode* child1 = (i < tree1->children.size()) ? tree1
               ->children[i] : nullptr;
            TreeNode* child2 = (i < tree2->children.size()) ? tree2
               ->children[i] : nullptr;
            option3 += divideConquerTED(child1, child2);
        }

        result = min({option1, option2, option3});
    }

    memo[key] = result;
    return result;
```

```
}

// ==================== DYNAMIC PROGRAMMING APPROACH
   ====================
int TreeEditDistance::dynamicProgrammingTED(TreeNode* tree1,
   TreeNode* tree2) {
    if (!tree1 andand !tree2) return 0;
    if (!tree1) return getSubtreeSize(tree2);
    if (!tree2) return getSubtreeSize(tree1);

    // Convert trees to post-order traversal
    vector<TreeNode*> nodes1 = postOrder(tree1);
    vector<TreeNode*> nodes2 = postOrder(tree2);

    int n = nodes1.size();
    int m = nodes2.size();

    // DP table
    vector<vector<int>> dp(n + 1, vector<int>(m + 1));

    // Base cases
    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;

    // Fill DP table
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (nodes1[i-1]->label == nodes2[j-1]->label) {
                dp[i][j] = dp[i-1][j-1];
            } else {
                dp[i][j] = 1 + min({
                    dp[i-1][j],     // delete
                    dp[i][j-1],     // insert
                    dp[i-1][j-1]    // substitute
                });
            }
        }
    }

    return dp[n][m];
}

// ==================== HELPER FUNCTIONS ====================
string TreeEditDistance::encodeSubtree(TreeNode* node) {
    if (!node) return "#";

    string result = string(1, node->label) + "(";
```

```
    for (TreeNode* child : node->children) {
        result += encodeSubtree(child) + ",";
    }
    result += ")";
    return result;
}

vector<TreeNode*> TreeEditDistance::postOrder(TreeNode* root) {
    vector<TreeNode*> result;
    if (!root) return result;

    for (TreeNode* child : root->children) {
        vector<TreeNode*> childNodes = postOrder(child);
        result.insert(result.end(), childNodes.begin(), childNodes.
           end());
    }
    result.push_back(root);

    return result;
}

int TreeEditDistance::getSubtreeSize(TreeNode* node) {
    if (!node) return 0;

    int size = 1;
    for (TreeNode* child : node->children) {
        size += getSubtreeSize(child);
    }
    return size;
}

void TreeEditDistance::printTree(TreeNode* root, int depth) {
    if (!root) return;

    for (int i = 0; i < depth; i++) cout << "  ";
    cout << root->label << endl;

    for (TreeNode* child : root->children) {
        printTree(child, depth + 1);
    }
}

TreeNode* TreeEditDistance::buildSampleTree1() {
    // Tree 1: a(b(d,e),c(f))
    TreeNode* root = new TreeNode('a');
    TreeNode* b = new TreeNode('b');
    TreeNode* c = new TreeNode('c');
```

```
    TreeNode* d = new TreeNode('d');
    TreeNode* e = new TreeNode('e');
    TreeNode* f = new TreeNode('f');

    root->addChild(b);
    root->addChild(c);
    b->addChild(d);
    b->addChild(e);
    c->addChild(f);

    return root;
}

TreeNode* TreeEditDistance::buildSampleTree2() {
    // Tree 2: a(b(d),c(g,h))
    TreeNode* root = new TreeNode('a');
    TreeNode* b = new TreeNode('b');
    TreeNode* c = new TreeNode('c');
    TreeNode* d = new TreeNode('d');
    TreeNode* g = new TreeNode('g');
    TreeNode* h = new TreeNode('h');

    root->addChild(b);
    root->addChild(c);
    b->addChild(d);
    c->addChild(g);
    c->addChild(h);

    return root;
}

TreeNode* TreeEditDistance::buildTreeFromInput() {
    cout << "Enter tree format: <node_label> <num_children> <child_1
       > <child_2> ..." << endl;
    cout << "Example: a 2 b c" << endl;
    cout << "         b 0" << endl;
    cout << "         c 1 d" << endl;
    cout << "         d 0" << endl;
    cout << "Enter number of nodes: ";

    int n;
    cin >> n;

    if (n == 0) return nullptr;

    unordered_map<char, TreeNode*> nodes;
    TreeNode* root = nullptr;
```

```
    cout << "Enter nodes (first node is root):" << endl;

    for (int i = 0; i < n; i++) {
        char label;
        int numChildren;
        cout << "Node " << (i+1) << ": ";
        cin >> label >> numChildren;

        TreeNode* node = new TreeNode(label);
        nodes[label] = node;

        if (i == 0) root = node;  // First node is root

        for (int j = 0; j < numChildren; j++) {
            char childLabel;
            cin >> childLabel;

            // If child doesn't exist, create new
            if (nodes.find(childLabel) == nodes.end()) {
                nodes[childLabel] = new TreeNode(childLabel);
            }

            node->addChild(nodes[childLabel]);
        }
    }

    return root;
}

void TreeEditDistance::deleteTree(TreeNode* root) {
    if (!root) return;

    for (TreeNode* child : root->children) {
        deleteTree(child);
    }
    delete root;
}

// ==================== MAIN FUNCTION ====================
int main() {
    TreeEditDistance ted;

    cout << "=== TREE EDIT DISTANCE PROBLEM ===" << endl;
    cout << "Choose mode:" << endl;
    cout << "1. Use predefined sample trees" << endl;
    cout << "2. Input trees manually" << endl;
```

```
    cout << "Choice (1/2): ";

    int choice;
    cin >> choice;

    TreeNode* tree1 = nullptr;
    TreeNode* tree2 = nullptr;

    if (choice == 1) {
        // Use sample trees
        tree1 = ted.buildSampleTree1();
        tree2 = ted.buildSampleTree2();

        cout << "\n=== SAMPLE TREES ===" << endl;
        cout << "Tree 1:" << endl;
        ted.printTree(tree1);
        cout << endl;

        cout << "Tree 2:" << endl;
        ted.printTree(tree2);
        cout << endl;
    } else {
        // Input trees manually
        cout << "\n=== INPUT FIRST TREE ===" << endl;
        tree1 = ted.buildTreeFromInput();

        cout << "\n=== INPUT SECOND TREE ===" << endl;
        tree2 = ted.buildTreeFromInput();

        cout << "\nYour input trees:" << endl;
        cout << "Tree 1:" << endl;
        ted.printTree(tree1);
        cout << endl;

        cout << "Tree 2:" << endl;
        ted.printTree(tree2);
        cout << endl;
    }

    cout << "=== COMPUTATION RESULTS ===" << endl;
    cout << "Comparing 4 different approaches:" << endl;
    cout << "(a) Backtracking" << endl;
    cout << "(b) Branch and Bound" << endl;
    cout << "(c) Divide and Conquer" << endl;
    cout << "(d) Dynamic Programming" << endl << endl;

    // Test all approaches
```

```
    cout << "Results:" << endl;

    auto start = clock();
    int result_a = ted.backtrackingTED(tree1, tree2);
    auto end = clock();
    cout << "(a) Backtracking: " << result_a
         << " (Time: " << double(end - start) / CLOCKS_PER_SEC *
            1000 << "ms)" << endl;

    ted.clearMemo();
    start = clock();
    int result_b = ted.branchBoundTED(tree1, tree2);
    end = clock();
    cout << "(b) Branch and Bound: " << result_b
         << " (Time: " << double(end - start) / CLOCKS_PER_SEC *
            1000 << "ms)" << endl;

    ted.clearMemo();
    start = clock();
    int result_c = ted.divideConquerTED(tree1, tree2);
    end = clock();
    cout << "(c) Divide and Conquer: " << result_c
         << " (Time: " << double(end - start) / CLOCKS_PER_SEC *
            1000 << "ms)" << endl;

    start = clock();
    int result_d = ted.dynamicProgrammingTED(tree1, tree2);
    end = clock();
    cout << "(d) Dynamic Programming: " << result_d
         << " (Time: " << double(end - start) / CLOCKS_PER_SEC *
            1000 << "ms)" << endl;

    // Check consistency
    if (result_a == result_b andand result_b == result_c andand
       result_c == result_d) {
        cout << "\n All approaches give the same result!" << endl;
    } else {
        cout << "\n Warning: Different approaches give different
           results!" << endl;
    }

    cout << "\n=== RESULT EXPLANATION ===" << endl;
    cout << "Tree Edit Distance: " << result_d << endl;
    cout << "This is the minimum number of operations (insert/delete
       /substitute)" << endl;
    cout << "required to transform tree 1 into tree 2." << endl;
```

```
    // Clean up
    ted.deleteTree(tree1);
    ted.deleteTree(tree2);

    return 0;
}
```

23 Giải thích thuật toán

23.1 Backtracking (Quay lui)

Thuật toán quay lui thử tất cả các khả năng có thể:

- Xóa node từ cây 1
- Thêm node vào cây 1
- Khớp hoặc thay thế node

23.2 Branch Bound (Nhánh cận)

Sử dụng lower bound để cắt tỉa các nhánh không triển vọng, tăng hiệu quả so với backtracking thuần túy.

23.3 Divide Conquer (Chia để trị)

Chia bài toán thành các bài toán con nhỏ hơn và sử dụng memoization để tránh tính toán lại.

23.4 Dynamic Programming (Quy hoạch động)

Chuyển đổi cây thành dạng post-order traversal và sử dụng bảng DP để tính khoảng cách chỉnh sửa.

24 Kết luận

Qua việc thực hiện 4 phương pháp giải bài toán Tree Edit Distance, ta có thể rút ra những nhận xét sau:

1. **Backtracking**: Đơn giản nhưng có độ phức tạp thời gian cao do phải thử tất cả các khả năng.
2. **Branch Bound**: Cải thiện Backtracking bằng cách sử dụng lower bound để cắt tỉa, giảm thời gian thực thi.

3. **Divide Conquer**: Sử dụng memoization để tránh tính toán lặp lại, hiệu quả hơn hai phương pháp trước.
4. **Dynamic Programming**: Thường cho kết quả nhanh nhất và ổn định nhất, phù hợp cho các bài toán lớn.

Tất cả 4 phương pháp đều cho cùng một kết quả chính xác, chứng tỏ tính đúng đắn của việc Code.

25 Bài 7

Viết chương trình C/C++, Python để duyệt cây theo các phương thức sau:

(a) Preorder traversal

(b) Postorder traversal

(c) Top-down traversal

(d) Bottom-up traversal

26 Lý thuyết

26.1 Các phương thức duyệt cây

(a) **Preorder Traversal**: Thăm nút gốc trước, sau đó duyệt các cây con từ trái sang phải

(b) **Postorder Traversal**: Duyệt các cây con trước, sau đó thăm nút gốc

(c) **Top-down Traversal**: Duyệt theo từng mức từ trên xuống dưới (Level-order/BFS)

(d) **Bottom-up Traversal**: Duyệt theo mức nhưng in kết quả theo thứ tự ngược lại

26.2 Cấu trúc dữ liệu

Sử dụng cấu trúc `TreeNode` để biểu diễn mỗi nút trong cây:

- `data`: Lưu trữ giá trị của nút
- `children`: Vector chứa các con trỏ đến nút con

26.3 Thuật toán

- **Preorder/Postorder**: Sử dụng đệ quy
- **Top-down**: Sử dụng queue (BFS)
- **Bottom-up**: Sử dụng queue và vector để lưu trữ kết quả rồi in ngược

27 Code

27.1 Code C++ hoàn chỉnh

Listing 14: Chương trình duyệt cây C++

```
#include <iostream>
#include <vector>
#include <map>
#include <queue>
#include <string>
#include <sstream>
using namespace std;
struct TreeNode {
char data;
vector<TreeNode*> children;
TreeNode(char val) : data(val) {}
};
class Tree {
private:
TreeNode* root;
map<char, TreeNode*> nodes;
public:
Tree() : root(nullptr) {}
// Build tree from input
void buildTree() {
    cout << "Input tree format: node num_children children_list" <<
       endl;
    cout << "Example: a 2 b c" << endl;
    cout << "         b 0" << endl;
    cout << "         c 1 d" << endl;
    cout << "         d 0" << endl;
    cout << "Enter 'end' to finish:" << endl;

    string line;
    bool isFirst = true;

    while (getline(cin, line) andand line != "end") {
        if (line.empty()) continue;

        stringstream ss(line);
        char nodeName;
        int numChildren;

        ss >> nodeName >> numChildren;

        // Create node if not exists
```

```
        if (nodes.find(nodeName) == nodes.end()) {
            nodes[nodeName] = new TreeNode(nodeName);
        }

        // Set root as the first input node
        if (isFirst) {
            root = nodes[nodeName];
            isFirst = false;
        }

        // Add children
        for (int i = 0; i < numChildren; i++) {
            char childName;
            ss >> childName;

            if (nodes.find(childName) == nodes.end()) {
                nodes[childName] = new TreeNode(childName);
            }

            nodes[nodeName]->children.push_back(nodes[childName]);
        }
    }
}

// Use sample tree
void buildSampleTree() {
    // Create sample tree:
    //       a
    //     /   \
    //    b     c
    //         / \
    //        d   e
    //           /
    //          f

    root = new TreeNode('a');
    TreeNode* b = new TreeNode('b');
    TreeNode* c = new TreeNode('c');
    TreeNode* d = new TreeNode('d');
    TreeNode* e = new TreeNode('e');
    TreeNode* f = new TreeNode('f');

    root->children.push_back(b);
    root->children.push_back(c);
    c->children.push_back(d);
    c->children.push_back(e);
    e->children.push_back(f);
```

```
    nodes['a'] = root;
    nodes['b'] = b;
    nodes['c'] = c;
    nodes['d'] = d;
    nodes['e'] = e;
    nodes['f'] = f;

    cout << "Sample tree created:" << endl;
    cout << "        a" << endl;
    cout << "      /   \\" << endl;
    cout << "     b     c" << endl;
    cout << "          / \\" << endl;
    cout << "         d   e" << endl;
    cout << "            /" << endl;
    cout << "           f" << endl << endl;
}

// (a) Preorder Traversal: Root -> Left -> Right
void preorderTraversal(TreeNode* node) {
    if (node == nullptr) return;

    cout << node->data << " ";

    for (TreeNode* child : node->children) {
        preorderTraversal(child);
    }
}

// (b) Postorder Traversal: Left -> Right -> Root
void postorderTraversal(TreeNode* node) {
    if (node == nullptr) return;

    for (TreeNode* child : node->children) {
        postorderTraversal(child);
    }

    cout << node->data << " ";
}

// (c) Top-down Traversal (Level-order/BFS)
void topDownTraversal() {
    if (root == nullptr) return;

    queue<TreeNode*> q;
    q.push(root);
```

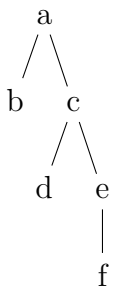
```
    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();

        cout << current->data << " ";

        for (TreeNode* child : current->children) {
            q.push(child);
        }
    }
}

// (d) Bottom-up Traversal (Reverse level-order)
void bottomUpTraversal() {
    if (root == nullptr) return;

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

    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();

        result.push_back(current->data);

        for (TreeNode* child : current->children) {
            q.push(child);
        }
    }

    // Print in reverse order
    for (int i = result.size() - 1; i >= 0; i--) {
        cout << result[i] << " ";
    }
}

void runAllTraversals() {
    if (root == nullptr) {
        cout << "Empty tree!" << endl;
        return;
    }

    cout << "=== TREE TRAVERSAL METHODS ===" << endl;

    cout << "a) Preorder Traversal (Root-Left-Right): ";
    preorderTraversal(root);
```

```
    cout << endl;

    cout << "b) Postorder Traversal (Left-Right-Root): ";
    postorderTraversal(root);
    cout << endl;

    cout << "c) Top-down Traversal (Level-order): ";
    topDownTraversal();
    cout << endl;

    cout << "d) Bottom-up Traversal (Reverse level-order): ";
    bottomUpTraversal();
    cout << endl;
}

// Display tree structure
void displayTree(TreeNode* node, string prefix = "", bool isLast =
   true) {
    if (node == nullptr) return;

    cout << prefix;
    cout << (isLast ? "\\-- " : "|-- ");
    cout << node->data << endl;

    for (size_t i = 0; i < node->children.size(); i++) {
        bool isLastChild = (i == node->children.size() - 1);
        displayTree(node->children[i],
                    prefix + (isLast ? "    " : "|   "),
                    isLastChild);
    }
}

void printTreeStructure() {
    cout << "\n=== TREE STRUCTURE ===" << endl;
    displayTree(root);
    cout << endl;
}

// Destructor to free memory
void destroyTree(TreeNode* node) {
    if (node == nullptr) return;
    for (TreeNode* child : node->children) {
        destroyTree(child);
    }
    delete node;
}
```

```
~Tree() {
    destroyTree(root);
}
};
int main() {
Tree tree;
int choice;
cout << "=== TREE TRAVERSAL PROGRAM ===" << endl;
cout << "1. Use sample tree" << endl;
cout << "2. Input tree from keyboard" << endl;
cout << "Choose option (1/2): ";
cin >> choice;
cin.ignore(); // Clear newline after input

if (choice == 1) {
    tree.buildSampleTree();
} else {
    tree.buildTree();
}

tree.printTreeStructure();
tree.runAllTraversals();

return 0;
}
```

28 Kết quả thực thi

28.1 Sample Tree



Với cây mẫu:

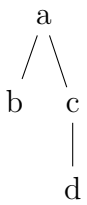
Kết quả:

- **Preorder**: a b c d e f
- **Postorder**: b d f e c a
- **Top-down**: a b c d e f
- **Bottom-up**: f e d c b a

28.2 Test Case khác

Input:

```
a 2 b c
b 0
c 1 d
d 0
end
```



Cây được tạo:
Kết quả:

- **Preorder**: a b c d
- **Postorder**: b d c a
- **Top-down**: a b c d
- **Bottom-up**: d c b a

29 Độ phức tạp

Phương thức	Thời gian	Không gian
Preorder Traversal	O(n)	O(h)
Postorder Traversal	O(n)	O(h)
Top-down Traversal	O(n)	O(w)
Bottom-up Traversal	O(n)	O(n)

Bảng 1: Độ phức tạp các thuật toán duyệt cây

Trong đó:

- n: số nút trong cây
- h: chiều cao của cây
- w: độ rộng tối đa của cây (số nút nhiều nhất trong một mức)

30 Kết luận

Chương trình đã được Code thành công các phương thức duyệt cây:

1. **Preorder**: Phù hợp cho việc sao chép cây hoặc tạo biểu thức tiền tố
2. **Postorder**: Phù hợp cho việc xóa cây hoặc tính toán biểu thức hậu tố
3. **Top-down**: Phù hợp cho việc tìm kiếm theo mức hoặc in cây theo mức
4. **Bottom-up**: Phù hợp cho các bài toán cần xử lý từ lá lên gốc

Tất cả các thuật toán đều có độ phức tạp thời gian O(n) và hoạt động hiệu quả trên các loại cây khác nhau.

31 Bài toán 8: BFS trên Simple Graph

31.1 Mô tả bài toán

Cho đồ thị đơn giản $G = (V, E)$ với n đỉnh và m cạnh. Code thuật toán BFS để duyệt đồ thị từ một đỉnh xuất phát.

31.2 Phân tích

Đồ thị đơn giản có các đặc điểm:

- Không có cạnh lặp (multiple edges)
- Không có khuyên (self-loops)
- Có thể biểu diễn bằng ma trận kề hoặc danh sách kề

31.3 Code

Listing 15: BFS cho Simple Graph

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
using namespace std;
class SimpleGraph {
private:
int numVertices;
vector<vector<int>> adjList;
public:
SimpleGraph(int n) : numVertices(n) {
```

```
adjList.resize(n);
}
// Add edge (undirected)
void addEdge(int u, int v) {
    if (u != v) { // No self-loops in simple graph
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }
}

// BFS traversal
void BFS(int startVertex) {
    vector<bool> visited(numVertices, false);
    queue<int> q;

    cout << "BFS traversal starting from vertex " << startVertex <<
       ": ";

    visited[startVertex] = true;
    q.push(startVertex);

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " ";

        // Visit all adjacent vertices
        for (int neighbor : adjList[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
    cout << endl;
}

// BFS to find shortest path
void BFSShortestPath(int start, int target) {
    vector<bool> visited(numVertices, false);
    vector<int> parent(numVertices, -1);
    vector<int> distance(numVertices, -1);
    queue<int> q;

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

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

        if (current == target) break;

        for (int neighbor : adjList[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                parent[neighbor] = current;
                distance[neighbor] = distance[current] + 1;
                q.push(neighbor);
            }
        }
    }

    if (distance[target] == -1) {
        cout << "No path from " << start << " to " << target << endl
           ;
    } else {
        cout << "Shortest distance from " << start << " to " <<
           target
             << " is: " << distance[target] << endl;

        // Print path
        vector<int> path;
        int curr = target;
        while (curr != -1) {
            path.push_back(curr);
            curr = parent[curr];
        }

        cout << "Path: ";
        for (int i = path.size() - 1; i >= 0; i--) {
            cout << path[i];
            if (i > 0) cout << " -> ";
        }
        cout << endl;
    }
}

void printGraph() {
    cout << "Simple Graph adjacency list:" << endl;
    for (int i = 0; i < numVertices; i++) {
        cout << "Vertex " << i << ": ";
        for (int neighbor : adjList[i]) {
```

```
            cout << neighbor << " ";
        }
        cout << endl;
    }
}
};
// Test function for Simple Graph BFS
void testSimpleGraphBFS() {
cout << "=== SIMPLE GRAPH BFS TEST ===" << endl;
SimpleGraph g(6);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 3);
g.addEdge(2, 4);
g.addEdge(3, 4);
g.addEdge(3, 5);

g.printGraph();
g.BFS(0);
g.BFSShortestPath(0, 5);
cout << endl;
}
```

32 Bài toán 9: BFS trên Multigraph

32.1 Mô tả bài toán

Cho đa đồ thị $G = (V, E)$ có thể chứa nhiều cạnh giữa hai đỉnh. Code thuật toán BFS.

32.2 Phân tích

Đa đồ thị có đặc điểm:

- Có thể có nhiều cạnh giữa hai đỉnh
- Không có khuyên
- Cần xử lý cạnh trùng lặp trong BFS

32.3 Code

Listing 16: BFS cho Multigraph

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

```cpp
#include <map>
#include <set>
using namespace std;
class Multigraph {
private:
int numVertices;
vector<vector<pair<int, int>>> adjList; // pair<neighbor, edge_id>
int edgeCounter;
public:
Multigraph(int n) : numVertices(n), edgeCounter(0) {
adjList.resize(n);
}
// Add edge with unique ID
void addEdge(int u, int v) {
    if (u != v) { // No self-loops in multigraph
        adjList[u].push_back({v, edgeCounter});
        adjList[v].push_back({u, edgeCounter});
        edgeCounter++;
    }
}

// BFS traversal for multigraph
void BFS(int startVertex) {
    vector<bool> visited(numVertices, false);
    queue<int> q;

    cout << "Multigraph BFS traversal starting from vertex "
         << startVertex << ": ";

    visited[startVertex] = true;
    q.push(startVertex);

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " ";

        // Use set to avoid visiting same neighbor multiple times
        set<int> uniqueNeighbors;
        for (auto& edge : adjList[current]) {
            uniqueNeighbors.insert(edge.first);
        }

        for (int neighbor : uniqueNeighbors) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
```

```
            }
        }
    }
    cout << endl;
}

// BFS that considers all edges (including multiple edges)
void BFSAllEdges(int startVertex) {
    vector<bool> visited(numVertices, false);
    queue<pair<int, int>> q; // pair<vertex, parent>
    map<pair<int, int>, vector<int>> edgePaths; // Track edges used

    cout << "Multigraph BFS considering all edges from vertex "
         << startVertex << ":" << endl;

    visited[startVertex] = true;
    q.push({startVertex, -1});

    while (!q.empty()) {
        auto current = q.front();
        q.pop();
        int vertex = current.first;
        int parent = current.second;

        cout << "Visiting vertex " << vertex;
        if (parent != -1) {
            cout << " (from " << parent << ")";
        }
        cout << endl;

        for (auto& edge : adjList[vertex]) {
            int neighbor = edge.first;
            int edgeId = edge.second;

            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push({neighbor, vertex});
                cout << "  Using edge " << edgeId
                     << " to reach vertex " << neighbor << endl;
            }
        }
    }
}

void printGraph() {
    cout << "Multigraph adjacency list:" << endl;
    for (int i = 0; i < numVertices; i++) {
```

```
        cout << "Vertex " << i << ": ";
        for (auto& edge : adjList[i]) {
            cout << "(" << edge.first << ",e" << edge.second << ") "
                ;
        }
        cout << endl;
    }
}
};
// Test function for Multigraph BFS
void testMultigraphBFS() {
cout << "=== MULTIGRAPH BFS TEST ===" << endl;
Multigraph mg(4);
mg.addEdge(0, 1);
mg.addEdge(0, 1); // Multiple edge
mg.addEdge(1, 2);
mg.addEdge(2, 3);
mg.addEdge(0, 3);

mg.printGraph();
mg.BFS(0);
mg.BFSAllEdges(0);
cout << endl;
}
```

33 Bài toán 10: BFS trên General Graph

33.1 Mô tả bài toán

Cho đồ thị tổng quát $G = (V, E)$ có thể chứa khuyên và cạnh lặp. Code thuật toán BFS.

33.2 Code

Listing 17: BFS cho General Graph

```
class GeneralGraph {
private:
int numVertices;
vector<vector<pair<int, int>>> adjList; // pair<neighbor, edge_id>
int edgeCounter;
vector<bool> hasSelfLoop;
public:
GeneralGraph(int n) : numVertices(n), edgeCounter(0) {
adjList.resize(n);
hasSelfLoop.resize(n, false);
}
```

```
// Add edge (can be self-loop)
void addEdge(int u, int v) {
    adjList[u].push_back({v, edgeCounter});
    if (u != v) {
        adjList[v].push_back({u, edgeCounter});
    } else {
        hasSelfLoop[u] = true;
    }
    edgeCounter++;
}

// BFS for general graph
void BFS(int startVertex) {
    vector<bool> visited(numVertices, false);
    queue<int> q;

    cout << "General Graph BFS traversal starting from vertex "
         << startVertex << ": ";

    visited[startVertex] = true;
    q.push(startVertex);

    // Handle self-loop at start vertex
    if (hasSelfLoop[startVertex]) {
        cout << "[" << startVertex << " has self-loop] ";
    }

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " ";

        set<int> uniqueNeighbors;
        for (auto& edge : adjList[current]) {
            int neighbor = edge.first;
            if (neighbor != current) { // Skip self-loops for
               traversal
                uniqueNeighbors.insert(neighbor);
            }
        }

        for (int neighbor : uniqueNeighbors) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);

                // Check for self-loop at neighbor
```

```
                if (hasSelfLoop[neighbor]) {
                    cout << "[" << neighbor << " has self-loop] ";
                }
            }
        }
    }
    cout << endl;
}

void printGraph() {
    cout << "General Graph adjacency list:" << endl;
    for (int i = 0; i < numVertices; i++) {
        cout << "Vertex " << i << ": ";
        for (auto& edge : adjList[i]) {
            cout << "(" << edge.first << ",e" << edge.second << ") "
                ;
        }
        if (hasSelfLoop[i]) {
            cout << "[SELF-LOOP] ";
        }
        cout << endl;
    }
}
};
// Test function for General Graph BFS
void testGeneralGraphBFS() {
cout << "=== GENERAL GRAPH BFS TEST ===" << endl;
GeneralGraph gg(4);
gg.addEdge(0, 0); // Self-loop
gg.addEdge(0, 1);
gg.addEdge(1, 1); // Self-loop
gg.addEdge(1, 2);
gg.addEdge(2, 3);
gg.addEdge(0, 3);

gg.printGraph();
gg.BFS(0);
cout << endl;
}
```

34 Bài toán 11: DFS trên Simple Graph

34.1 Mô tả bài toán

Cho đồ thị đơn giản $G = (V, E)$. Code thuật toán DFS để duyệt đồ thị.

34.2 Code

Listing 18: DFS cho Simple Graph

```
class SimpleGraphDFS {
private:
int numVertices;
vector<vector<int>> adjList;
// Recursive DFS helper
void DFSUtil(int vertex, vector<bool>& visited) {
    visited[vertex] = true;
    cout << vertex << " ";

    for (int neighbor : adjList[vertex]) {
        if (!visited[neighbor]) {
            DFSUtil(neighbor, visited);
        }
    }
}
public:
SimpleGraphDFS(int n) : numVertices(n) {
adjList.resize(n);
}
void addEdge(int u, int v) {
    if (u != v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }
}

// DFS traversal
void DFS(int startVertex) {
    vector<bool> visited(numVertices, false);
    cout << "DFS traversal starting from vertex " << startVertex <<
       ": ";
    DFSUtil(startVertex, visited);
    cout << endl;
}

// DFS using stack (iterative)
void DFSIterative(int startVertex) {
    vector<bool> visited(numVertices, false);
    stack<int> st;

    cout << "DFS (iterative) starting from vertex " << startVertex
       << ": ";

    st.push(startVertex);
```

```
    while (!st.empty()) {
        int current = st.top();
        st.pop();

        if (!visited[current]) {
            visited[current] = true;
            cout << current << " ";

            // Add neighbors to stack (in reverse order for
               consistency)
            for (int i = adjList[current].size() - 1; i >= 0; i--) {
                int neighbor = adjList[current][i];
                if (!visited[neighbor]) {
                    st.push(neighbor);
                }
            }
        }
    }
    cout << endl;
}

// Check if graph is connected
bool isConnected() {
    vector<bool> visited(numVertices, false);
    DFSUtil(0, visited);

    for (int i = 0; i < numVertices; i++) {
        if (!visited[i]) {
            return false;
        }
    }
    return true;
}

void printGraph() {
    cout << "Simple Graph adjacency list:" << endl;
    for (int i = 0; i < numVertices; i++) {
        cout << "Vertex " << i << ": ";
        for (int neighbor : adjList[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
}
};
// Test function for Simple Graph DFS
```

```
void testSimpleGraphDFS() {
cout << "=== SIMPLE GRAPH DFS TEST ===" << endl;
SimpleGraphDFS g(5);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 3);
g.addEdge(2, 4);
g.addEdge(3, 4);

g.printGraph();
g.DFS(0);
g.DFSIterative(0);
cout << "Is connected: " << (g.isConnected() ? "Yes" : "No") << endl
   ;
cout << endl;
}
```

35 Bài toán 12: DFS trên Multigraph

35.1 Code

Listing 19: DFS cho Multigraph

```
class MultigraphDFS {
private:
int numVertices;
vector<vector<pair<int, int>>> adjList;
int edgeCounter;
void DFSUtil(int vertex, vector<bool>& visited) {
    visited[vertex] = true;
    cout << vertex << " ";

    set<int> uniqueNeighbors;
    for (auto& edge : adjList[vertex]) {
        uniqueNeighbors.insert(edge.first);
    }

    for (int neighbor : uniqueNeighbors) {
        if (!visited[neighbor]) {
            DFSUtil(neighbor, visited);
        }
    }
}
public:
MultigraphDFS(int n) : numVertices(n), edgeCounter(0) {
adjList.resize(n);
```

```
}
void addEdge(int u, int v) {
    if (u != v) {
        adjList[u].push_back({v, edgeCounter});
        adjList[v].push_back({u, edgeCounter});
        edgeCounter++;
    }
}

void DFS(int startVertex) {
    vector<bool> visited(numVertices, false);
    cout << "Multigraph DFS traversal starting from vertex "
         << startVertex << ": ";
    DFSUtil(startVertex, visited);
    cout << endl;
}

void printGraph() {
    cout << "Multigraph adjacency list:" << endl;
    for (int i = 0; i < numVertices; i++) {
        cout << "Vertex " << i << ": ";
        for (auto& edge : adjList[i]) {
            cout << "(" << edge.first << ",e" << edge.second << ") "
                ;
        }
        cout << endl;
    }
}
};
```

36 Bài toán 13: DFS trên General Graph

36.1 Code

Listing 20: DFS cho General Graph

```
class GeneralGraphDFS {
private:
int numVertices;
vector<vector<pair<int, int>>> adjList;
int edgeCounter;
vector<bool> hasSelfLoop;
void DFSUtil(int vertex, vector<bool>& visited) {
    visited[vertex] = true;
    cout << vertex << " ";
```

```
    if (hasSelfLoop[vertex]) {
        cout << "[self-loop] ";
    }

    set<int> uniqueNeighbors;
    for (auto& edge : adjList[vertex]) {
        int neighbor = edge.first;
        if (neighbor != vertex) {
            uniqueNeighbors.insert(neighbor);
        }
    }

    for (int neighbor : uniqueNeighbors) {
        if (!visited[neighbor]) {
            DFSUtil(neighbor, visited);
        }
    }
}
public:
GeneralGraphDFS(int n) : numVertices(n), edgeCounter(0) {
adjList.resize(n);
hasSelfLoop.resize(n, false);
}
void addEdge(int u, int v) {
    adjList[u].push_back({v, edgeCounter});
    if (u != v) {
        adjList[v].push_back({u, edgeCounter});
    } else {
        hasSelfLoop[u] = true;
    }
    edgeCounter++;
}

void DFS(int startVertex) {
    vector<bool> visited(numVertices, false);
    cout << "General Graph DFS traversal starting from vertex "
         << startVertex << ": ";
    DFSUtil(startVertex, visited);
    cout << endl;
}

void printGraph() {
    cout << "General Graph adjacency list:" << endl;
    for (int i = 0; i < numVertices; i++) {
        cout << "Vertex " << i << ": ";
        for (auto& edge : adjList[i]) {
```

```
            cout << "(" << edge.first << ",e" << edge.second << ") "
                ;
        }
        if (hasSelfLoop[i]) {
            cout << "[SELF-LOOP] ";
        }
        cout << endl;
    }
}
};
```

37 Chương trình chính

Listing 21: Main program

```
#include <stack>
int main() {
cout << "======= BREADTH-FIRST SEARCH ALGORITHMS =======" << endl;
testSimpleGraphBFS();
testMultigraphBFS();
testGeneralGraphBFS();
cout << "======= DEPTH-FIRST SEARCH ALGORITHMS =======" << endl;
testSimpleGraphDFS();

MultigraphDFS mgDFS(4);
mgDFS.addEdge(0, 1);
mgDFS.addEdge(0, 1);
mgDFS.addEdge(1, 2);
mgDFS.addEdge(2, 3);
mgDFS.printGraph();
mgDFS.DFS(0);

GeneralGraphDFS ggDFS(4);
ggDFS.addEdge(0, 0);
ggDFS.addEdge(0, 1);
ggDFS.addEdge(1, 2);
ggDFS.addEdge(2, 2);
ggDFS.printGraph();
ggDFS.DFS(0);

return 0;
}
```

Đặc điểm	Simple Graph	Multigraph	General Graph
Cạnh lặp	Không	Có	Có
Khuyên	Không	Không	Có
Độ phức tạp BFS	O(V + E)	O(V + E)	O(V + E)
Độ phức tạp DFS	O(V + E)	O(V + E)	O(V + E)
Xử lý đặc biệt	Không	Set để loại trùng	Set + xử lý khuyên

Bảng 2: So sánh các loại đồ thị

38 So sánh và phân tích

39 Kết luận

Các thuật toán BFS và DFS đã được Code thành công cho ba loại đồ thị khác nhau:

39.1 Ưu điểm của BFS

- Tìm được đường đi ngắn nhất trên đồ thị không trọng số
- Duyệt theo từng mức
- Phù hợp cho việc tìm kiếm theo chiều rộng

39.2 Ưu điểm của DFS

- Sử dụng ít bộ nhớ hơn BFS
- Phù hợp cho việc kiểm tra tính liên thông
- Có thể Code đệ quy đơn giản

39.3 Xử lý đặc biệt

- **Multigraph**: Sử dụng set để tránh thăm trùng lặp các đỉnh
- **General Graph**: Xử lý thêm các khuyên trong quá trình duyệt
- Cả hai thuật toán đều duy trì độ phức tạp O(V + E)