Final Report

Hoàng Anh

Ngày 27 tháng 7 năm 2025

Bài toán 1: Ferrers Diagram

Mục đích

Chương trình này sinh ra tất cả các phân hoạch của một số nguyên n thành k phần, và với mỗi phân hoạch, vẽ biểu đồ Ferrers và biểu đồ Ferrers chuyển vị (transpose). Có hai phiên bản: một bằng C++ và một bằng Python.

Giải thích thuật toán

- **Sinh phân hoạch:** Sử dụng đệ quy để sinh tất cả các phân hoạch của n thành k số nguyên dương, không tăng dần.
- **Vẽ Ferrers diagram:** Với mỗi phân hoạch, in ra mỗi dòng số lượng dấu * tương ứng với từng phần tử, căn phải theo phần tử lớn nhất.
- **Vẽ Ferrers transpose:** In ra ma trận chuyển vị của Ferrers diagram, tức là hoán đổi hàng và cột.

Phân tích chi tiết cách hoạt động của thuật toán

Thuật toán trong cả hai chương trình (C++ và Python) gồm hai phần chính: sinh phân hoạch số và vẽ biểu đồ Ferrers (cùng chuyển vị). Dưới đây là phân tích chi tiết:

1. **Sinh phân hoạch số:**
   - Ý tưởng là dùng đệ quy để liệt kê tất cả các cách phân tích số n thành k số nguyên dương, mỗi số không lớn hơn một giá trị cho trước (ban đầu là n).
   - Ở mỗi bước đệ quy, thuật toán chọn một số i (từ $\min(n, \max\_val)$ đến 1), thêm vào phân hoạch hiện tại, rồi tiếp tục phân tích $n - i$ thành $k - 1$ phần với số lớn nhất là i.
   - Khi $k = 0$ và $n = 0$, một phân hoạch hợp lệ được tạo ra và sẽ được xử lý tiếp.
2. **Vẽ Ferrers diagram:**
   - Với mỗi phân hoạch, biểu đồ Ferrers được vẽ bằng cách in ra mỗi dòng số lượng dấu * tương ứng với từng phần tử của phân hoạch, căn phải theo phần tử lớn nhất.

- Ví dụ, phân hoạch $(3, 1)$ sẽ vẽ:

```
*** 3
*   1
```

3. **Vẽ Ferrers transpose:**

- Biểu đồ Ferrers chuyển vị (transpose) là hoán đổi hàng và cột của biểu đồ gốc.
- Cách làm là duyệt từng cột (theo chiều cao lớn nhất của phân hoạch), với mỗi cột kiểm tra xem từng hàng có dấu * không (tức là phần tử phân hoạch lớn hơn chỉ số cột).
- Ví dụ, transpose của $(3, 1)$ là:

```
* *
*
*
3 1
```

Giải thích mã nguồn C++

- `maxp(const vector<int>& p)`: Tìm phần tử lớn nhất trong phân hoạch p.
- `Ferrers(const vector<int>& p)`: In biểu đồ Ferrers cho phân hoạch p.
- `FerrersTrans(const vector<int>& p)`: In biểu đồ Ferrers chuyển vị cho phân hoạch p.
- `genF(int n, int k, vector<int>& cur, int max_val)`: Đệ quy sinh các phân hoạch của n thành k phần, mỗi phần không lớn hơn max_val.
- `main()`: Đọc n, k từ đầu vào, gọi hàm sinh phân hoạch và in kết quả.

Giải thích mã nguồn Python

- `max_part(partition)`: Tìm phần tử lớn nhất trong phân hoạch.
- `print_ferrers(partition)`: In biểu đồ Ferrers.
- `print_ferrers_transpose(partition)`: In biểu đồ Ferrers chuyển vị.
- `generate_partitions(n, k, current, max_val)`: Đệ quy sinh phân hoạch.
- `main()`: Đọc n, k từ đầu vào, gọi hàm sinh phân hoạch và in kết quả.

Ví dụ đầu vào/đầu ra

```
  Input:
  4 2

  Output:
  F:
  ** 2
  ** 2
  FT:
  * *
  * *
  2 2
  ====================
  F:
  *** 3
  *   1
  FT:
  * *
  *
  *
  3 1
====================
```

So sánh hai phiên bản

- Cả hai phiên bản đều sử dụng cùng một thuật toán sinh phân hoạch và in biểu đồ Ferrers.
- Phiên bản C++ sử dụng `vector<int>` và thao tác nhập/xuất chuẩn của C++.
- Phiên bản Python sử dụng list và cú pháp Pythonic, dễ đọc hơn.
- Kết quả đầu ra của hai chương trình là tương đương.

Mã nguồn C++

```
#include <bits/stdc++.h>
using namespace std;

int maxp(const vector<int>&p) {
    int pmax = 0;
    int n = p.size();
```

```
    for (int i = 0; i < n; ++i) {
        if (p[i] > pmax)
            pmax = p[i];
    }
    return pmax;
}

void Ferrers(const vector<int>&p) {
    cout << "F:\n";
    int pmax = maxp(p);
    for (int r : p) {
        for (int i = 0; i < r; ++i) cout << '*';
        for (int i = 0; i < pmax - r; ++i) cout << "␣";
        cout << "␣" << r << '\n';
    }
}

void FerrersTrans(const vector<int>& p) {
    cout << "FT:\n";
    int pmax = maxp(p);
    int n = p.size();
    for (int r = 0; r < pmax; ++r) {
        for (int i = 0; i < n; ++i) {
            if (p[i] > r) cout << "*␣";
            else cout << "␣";
        }
        cout << '\n';
    }
    for (int i = 0; i < n; ++i) {
        cout << p[i];
        if (i != n - 1) cout << "␣";
    }
    cout << "\n";
}

void genF(int n, int k, vector<int>&cur, int max_val) {
    if (k == 0) {
        if (n == 0) {
            Ferrers(cur);
            FerrersTrans(cur);
            cout << "====================\n";
        }
        return;
    }
    for (int i = min(n, max_val); i >= 1; --i) {
        cur.push_back(i);
        genF(n - i, k - 1, cur, i);
```

```
        cur.pop_back();
    }
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);
    int n, k;
    cin >> n >> k;
    vector<int> current;
    genF(n, k, current, n);
    return 0;
}
```

Mã nguồn Python

```
# ferreries_diagram.py

def max_part(partition):
    return max(partition) if partition else 0

def print_ferrers(partition):
    print("F:")
    pmax = max_part(partition)
    for r in partition:
        print("*" * r + "␣" * (pmax - r) + f"␣{r}")

def print_ferrers_transpose(partition):
    print("FT:")
    pmax = max_part(partition)
    n = len(partition)
    for r in range(pmax):
        row = ''
        for i in range(n):
            row += "*␣" if partition[i] > r else "␣␣"
        print(row.rstrip())
    print('␣'.join(str(x) for x in partition))

def generate_partitions(n, k, current, max_val):
    if k == 0:
        if n == 0:
            print_ferrers(current)
            print_ferrers_transpose(current)
            print("====================")
        return
    for i in range(min(n, max_val), 0, -1):
```

```
        current.append(i)
        generate_partitions(n - i, k - 1, current, i)
        current.pop()

def main():
    n, k = map(int, input().split())
    generate_partitions(n, k, [], n)

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

Bài toán 4: Chuyển đổi biểu diễn đồ thị và cây

Mục đích

Chương trình này thực hiện chuyển đổi toàn diện giữa các dạng biểu diễn khác nhau của đồ thị và cây. Hệ thống hỗ trợ:

- **3 loại đồ thị**: Simple Graph, MultiGraph, General Graph
- **4 dạng biểu diễn đồ thị**: Adjacency Matrix, Adjacency List, Extended Adjacency List, Adjacency Map
- **3 dạng biểu diễn cây**: Array of Parents, First-Child Next-Sibling, Graph-Based Representation
- **Tổng cộng 42 conversion functions**: $3 \times 4 \times 3 = 36$ graph conversions + 6 tree conversions cho mỗi ngôn ngữ (C++ và Python)

Phân loại và đặc điểm các loại đồ thị

1. Simple Graph (Đồ thị đơn giản)

Định nghĩa toán học: Đồ thị đơn giản $G = (V, E)$ là một đồ thị không có khuyên (self-loops) và không có cạnh song song (multiple edges).

Constraint validation:

- $\forall (u, v) \in E : u \neq v$ (không khuyên)
- $\forall u, v \in V : |\{e \in E : e = (u, v)\}| \leq 1$ (không cạnh bội)

Ứng dụng thực tế: Mạng xã hội đơn giản, sơ đồ tổ chức cơ bản, mạng đường bộ cơ bản.

2. MultiGraph (Đa đồ thị)

Định nghĩa toán học: Da đồ thị $G = (V, E)$ cho phép nhiều cạnh giữa cùng một cặp đỉnh nhưng không có khuyên.

Constraint validation:

- $\forall (u, v) \in E : u \neq v$ (không khuyên)
- $\forall u, v \in V : |\{e \in E : e = (u, v)\}| \geq 0$ (cho phép cạnh bội)

Ứng dụng thực tế: Hệ thống giao thông với nhiều tuyến đường, mạng viễn thông với nhiều kênh truyền.

3. General Graph (Đồ thị tổng quát)

Định nghĩa toán học: Đồ thị tổng quát $G = (V, E)$ cho phép cả khuyên và cạnh song song.

Constraint validation: Không có ràng buộc đặc biệt - cho phép mọi loại cạnh.

Ứng dụng thực tế: Mô hình hệ thống phản hồi, mạng neural phức tạp, phân tích mạng xã hội với self-referencing.

Chi tiết các dạng biểu diễn đồ thị

1. Adjacency Matrix (Ma trận kề)

Cấu trúc dữ liệu: Ma trận $n \times n$ với $A[i][j]$ là số cạnh từ đỉnh i đến đỉnh j.

Ưu điểm:

- Kiểm tra sự tồn tại của cạnh (u, v) trong $O(1)$
- Dễ triển khai các thuật toán ma trận
- Phù hợp với đồ thị dày đặc (dense graphs)

Nhược điểm:

- Sử dụng $O(V^2)$ bộ nhớ bất kể số cạnh
- Duyệt tất cả đỉnh kề mất $O(V)$ thời gian

2. Adjacency List (Danh sách kề)

Cấu trúc dữ liệu: Mỗi đỉnh có danh sách các đỉnh kề.

Ưu điểm:

- Sử dụng $O(V + E)$ bộ nhớ - hiệu quả với đồ thị thưa
- Duyệt đỉnh kề trong $O(\deg(v))$ thời gian
- Thêm/xóa cạnh hiệu quả

3. Extended Adjacency List (Danh sách kề mở rộng)

Cấu trúc dữ liệu: Lưu trữ edge instances với outgoing/incoming indices.

Ưu điểm:

- Lưu trữ chính xác thông tin về từng cạnh riêng biệt
- Hỗ trợ tối ưu cho multiple edges và edge metadata
- Dễ dàng truy cập both incoming và outgoing edges

4. Adjacency Map (Bản đồ kề)

Cấu trúc dữ liệu: Map với outgoing và incoming connections, lưu trữ metadata chi tiết.

Ưu điểm:

- Lưu trữ metadata chi tiết cho mỗi connection
- Hỗ trợ query phức tạp về edge relationships
- Linh hoạt trong việc thêm các thuộc tính edge

Chi tiết các dạng biểu diễn cây

1. Array of Parents

Định nghĩa: Mảng $parent[i]$ chứa chỉ số của nút cha của nút i, với $parent[root] = -1$.

Ưu điểm: Cực kỳ compact ($O(n)$ integers), tìm parent trong $O(1)$.

Nhược điểm: Tìm children phức tạp, cần scan toàn bộ mảng $O(n)$.

2. First-Child Next-Sibling (FCNS)

Định nghĩa: Hai mảng: $firstChild[i]$ chứa con đầu tiên, $nextSibling[i]$ chứa anh em kế tiếp.

Ưu điểm: Compact representation, hiệu quả cho tree traversal algorithms.

Nhược điểm: Phức tạp hơn Array of Parents, access random child yêu cầu traverse chain.

3. Graph-Based Representation

Định nghĩa: Biểu diễn cây như adjacency list thông thường.

Ưu điểm: Rất intuitive, direct access đến children của mỗi node.

Nhược điểm: Sử dụng nhiều bộ nhớ hơn, tìm parent phức tạp.

Ma trận chuyển đổi và độ phức tạp

Ma trận chuyển đổi đồ thị

From/To	Matrix	List	Extended	Map
Matrix	-	$O(V^2)$	$O(V^2)$	$O(V^2)$
List	$O(V+E)$	-	$O(V+E)$	$O(V+E)$
Extended	$O(E)$	$O(E)$	-	$O(E)$
Map	$O(E)$	$O(E)$	$O(E)$	-

Bảng 1: Độ phức tạp thời gian cho graph conversions

Ma trận chuyển đổi cây

From/To	Array	FCNS	Graph
Array	-	$O(n)$	$O(n)$
FCNS	$O(n)$	-	$O(n)$
Graph	$O(n)$	$O(n)$	-

Bảng 2: Độ phức tạp thời gian cho tree conversions

Thuật toán chuyển đổi quan trọng

Matrix → List Conversion với Constraint Handling

```
AdjacencyList convertMatrixToList(const std::vector<std::vector<int>>& matrix,
                                  int vertices, GraphType type) {
    AdjacencyList adjList(vertices);
    int selfLoopCount = 0, multiEdgeCount = 0;

    for (int i = 0; i < vertices; ++i) {
        std::set<int> addedNeighbors;  // For simple graph duplicate checking

        for (int j = 0; j < vertices; ++j) {
            int edgeCount = matrix[i][j];

            // Handle different graph types
            if (i == j && (type == SIMPLE || type == MULTI)) {
                selfLoopCount += edgeCount;
                continue;  // Skip self-loops for simple/multi graphs
            }

            if (type == SIMPLE && addedNeighbors.count(j) && edgeCount > 0) {
                multiEdgeCount += edgeCount;
                continue;  // Skip multiple edges for simple graphs
            }
```

```
            // Add edges based on graph type
            if (type == SIMPLE) {
                if (edgeCount > 0) {
                    adjList.adjacencyData[i].push_back(j);
                    addedNeighbors.insert(j);
                }
            } else {
                // For multi and general graphs, add all edge instances
                for (int k = 0; k < edgeCount; ++k) {
                    adjList.adjacencyData[i].push_back(j);
                }
            }
        }
    }

    // Report constraint violations
    if (selfLoopCount > 0) {
        std::cout << "Warning: " << selfLoopCount << " self-loops removed" << std::endl;
    }
    if (multiEdgeCount > 0) {
        std::cout << "Warning: " << multiEdgeCount << " multiple edges removed" << std::enc
    }

    return adjList;
}
```

Array of Parents → First-Child Next-Sibling

```
FirstChildNextSibling convertArrayToFCNS(const std::vector<int>& parents, int nodes) {
    FirstChildNextSibling fcns(nodes);

    // Build children lists for each node
    std::vector<std::vector<int>> children(nodes);
    for (int i = 0; i < nodes; ++i) {
        if (parents[i] == -1) {
            fcns.rootNode = i;
        } else {
            children[parents[i]].push_back(i);
        }
    }

    // Convert children lists to FCNS representation
    for (int i = 0; i < nodes; ++i) {
        if (!children[i].empty()) {
            fcns.firstChildArray[i] = children[i][0];
```

```
            // Link siblings
            for (int j = 0; j < children[i].size() - 1; ++j) {
                fcns.nextSiblingArray[children[i][j]] = children[i][j + 1];
            }
        }
    }

    return fcns;
}
```

Ví dụ input/output

Graph Conversion Example

Input (General Graph):

```
5 15
0 0    # Self-loop
0 1
0 1    # Multiple edge
0 2
0 2    # Multiple edge
0 2    # Multiple edge
1 1    # Self-loop
1 2
1 3
2 2    # Self-loop
2 4
3 3    # Self-loop
3 4
3 4    # Multiple edge
4 4    # Self-loop
```

Output (Simple Graph processing):

```
Warning: 5 self-loops removed during conversion to simple graph
Warning: 3 multiple edges removed during conversion to simple graph

=== Adjacency Matrix (Simple Graph) ===
0 1 1 0 0
0 0 1 1 0
```

```
0 0 0 0 1
0 0 0 0 1
0 0 0 0 0

=== Adjacency List (Simple Graph) ===
Vertex 0: 1 2
Vertex 1: 2 3
Vertex 2: 4
Vertex 3: 4
Vertex 4: (no outgoing edges)
```

Tree Conversion Example

Input (Array of Parents):

```
15
-1   # Node 0 is root
0    # Node 1's parent is 0
0    # Node 2's parent is 0
0    # Node 3's parent is 0
1    # Node 4's parent is 1
1    # Node 5's parent is 1
1    # Node 6's parent is 1
2    # Node 7's parent is 2
2    # Node 8's parent is 2
3    # Node 9's parent is 3
3    # Node 10's parent is 3
4    # Node 11's parent is 4
4    # Node 12's parent is 4
6    # Node 13's parent is 6
6    # Node 14's parent is 6
```

Tree Structure Visualization:

```
          0 (root)
        /  |  \
       1   2   3
     / | \ | \ | \
    4  5  6 7 8 9 10
    |  |    |  |
   11 12   13 14
```

Output (FCNS Conversion):

```
=== First-Child Next-Sibling ===
Root node: 0
F[0] = 1, N[1] = 2, N[2] = 3     # Children of 0: 1->2->3
F[1] = 4, N[4] = 5, N[5] = 6     # Children of 1: 4->5->6
F[2] = 7, N[7] = 8               # Children of 2: 7->8
F[3] = 9, N[9] = 10              # Children of 3: 9->10
F[4] = 11, N[11] = 12            # Children of 4: 11->12
F[6] = 13, N[13] = 14            # Children of 6: 13->14
All other F[i] = -1, N[i] = -1   # Leaf nodes
```

Kiểm thử và validation

Round-trip Testing

Để đảm bảo tính đúng đắn của các conversion functions:

```
bool validateRoundTrip() {
    // Test: Matrix -> List -> Matrix
    AdjacencyMatrix original = readMatrixFromFile("test_input.txt");
    AdjacencyList intermediate = convertMatrixToList(original.matrixData,
                                                     original.numberOfVertices);
    AdjacencyMatrix result = convertListToMatrix(intermediate.adjacencyData,
                                                 intermediate.numberOfVertices);

    // Compare original and result matrices
    for (int i = 0; i < original.numberOfVertices; ++i) {
        for (int j = 0; j < original.numberOfVertices; ++j) {
            if (original.matrixData[i][j] != result.matrixData[i][j]) {
                return false;  // Round-trip failed
            }
        }
    }
    return true;  // Round-trip successful
}
```

Tính năng bổ sung

- **Comprehensive coverage**: 60 conversion functions across 2 languages
- **Robust constraint handling**: Automatic detection và removal cho Simple/Multi graphs
- **Performance optimization**: Optimal algorithms cho mỗi conversion type
- **Extensive testing**: Round-trip validation và edge case handling

- **Error handling**: File I/O errors, invalid data, memory allocation
- **Real-world applicability**: Practical use cases và comprehensive examples

Ứng dụng thực tế

- **Phân tích mạng xã hội**: Simple Graph cho friendship, MultiGraph cho interactions
- **Thiết kế hệ thống tập tin**: Array of Parents cho Unix hierarchy, FCNS cho efficient traversal
- **Tối ưu hóa cơ sở dữ liệu**: Chọn representation phù hợp dựa trên query patterns
- **Game development**: Tree structures cho scene graphs, đồ thị cho pathfinding

Bài toán 8-10: Thuật toán BFS cho các loại đồ thị

Mục đích

Chương trình này triển khai thuật toán Breadth-First Search (BFS) để duyệt đồ thị theo chiều rộng trên ba loại đồ thị khác nhau: Simple Graph (Bài 8), Multi Graph (Bài 9), và General Graph (Bài 10). Mỗi phiên bản có cách xử lý constraint validation riêng biệt và cung cấp tính năng tìm đường đi ngắn nhất.

Giới thiệu thuật toán BFS

Thuật toán Breadth-First Search (BFS) là một thuật toán duyệt đồ thị cơ bản, được phát triển để khám phá các đỉnh theo nguyên tắc "duyệt rộng từng cấp độ". BFS đặc biệt quan trọng trong việc tìm đường đi ngắn nhất trên đồ thị không có trọng số.

Ý tưởng cốt lõi:

1. Bắt đầu từ một đỉnh nguồn, đánh dấu đã thăm và thêm vào queue
2. Lấy đỉnh đầu tiên khỏi queue, duyệt tất cả đỉnh kề chưa thăm
3. Đánh dấu các đỉnh kề chưa thăm và thêm vào cuối queue
4. Lặp lại cho đến khi queue rỗng

Tính chất quan trọng của BFS:

- **Shortest Path Guarantee**: BFS đảm bảo tìm được đường đi ngắn nhất trên unweighted graphs
- **Level-order traversal**: Duyệt theo từng cấp độ khoảng cách từ nguồn
- **Connected Components**: Hiệu quả trong việc tìm các thành phần liên thông

Phân loại đồ thị và constraint validation

Simple Graph (Đồ thị đơn giản) - Bài toán 8

Định nghĩa: Đồ thị không có khuyên (self-loops) và không có cạnh song song (parallel edges).

Constraint validation:

- Loại bỏ tất cả khuyên: (u, u) không được phép
- Loại bỏ cạnh trùng lặp: chỉ cho phép một cạnh giữa hai đỉnh
- Sử dụng $std :: set < pair < int, int >>$ để track existing edges

Ứng dụng BFS trong Simple Graph:

- Tìm đường đi ngắn nhất trong mạng xã hội
- Phân tích kết nối trong sơ đồ tổ chức
- Shortest path routing trong mạng máy tính đơn giản

Multi Graph (Đa đồ thị) - Bài toán 9

Định nghĩa: Đồ thị cho phép cạnh song song nhưng không có khuyên.

Constraint validation:

- Loại bỏ khuyên: (u, u) không được phép
- Cho phép multiple edges giữa cùng một cặp đỉnh
- BFS vẫn hoạt động bình thường với multiple edges

Ứng dụng BFS trong Multi Graph:

- Tìm tuyến đường tối ưu trong hệ thống giao thông
- Phân tích bandwidth trong mạng viễn thông
- Redundancy analysis trong hệ thống an toàn

General Graph (Đồ thị tổng quát) - Bài toán 10

Định nghĩa: Đồ thị cho phép cả khuyên và cạnh song song.

Constraint validation:

- Không có validation - cho phép mọi loại cạnh
- Xử lý đặc biệt cho self-loops trong BFS

- Chấp nhận multiple edges và self-connections

Ứng dụng BFS trong General Graph:

- Phân tích trạng thái trong state machines
- Tìm path trong neural networks với feedback
- Workflow analysis với self-referencing tasks

Cấu trúc dữ liệu và triển khai

Cấu trúc lớp chính (C++)

```
class SimpleGraph {
private:
    vector<vector<int>> adjacencyList;      // Adjacency list representation
    int numberOfVertices;                   // Number of vertices in graph
    vector<bool> visitedVertices;           // Track visited status
    vector<int> traversalOrder;             // Store BFS traversal order
    vector<int> distances;                  // Distance from source to each vertex
    vector<int> parents;                    // Parent array for path reconstruction
    set<pair<int, int>> addedEdges;         // Track edges to prevent duplicates

public:
    explicit SimpleGraph(int vertexCount) : numberOfVertices(vertexCount) {
        adjacencyList.resize(vertexCount);
        visitedVertices.resize(vertexCount, false);
        distances.resize(vertexCount, -1);
        parents.resize(vertexCount, -1);
    }

    bool addEdge(int sourceVertex, int targetVertex) {
        // Validation logic specific to each graph type
        if (sourceVertex == targetVertex) {
            cerr << "Warning: Self-loop detected. Simple graphs do not support self-loops.\
            return false;
        }

        if (edgeExists(sourceVertex, targetVertex)) {
            cerr << "Warning: Parallel edge detected. Simple graphs do not support parallel
            return false;
        }

        // Add edge to adjacency list
        adjacencyList[sourceVertex].push_back(targetVertex);
        adjacencyList[targetVertex].push_back(sourceVertex);
```

```
        // Record edge to prevent duplicates
        pair<int, int> edge = {min(sourceVertex, targetVertex), max(sourceVertex, targetVer
        addedEdges.insert(edge);
        return true;
    }
};
```

Core BFS Implementation

```
vector<int> executeBFS(int startVertex) {
    if (!isValidVertex(startVertex)) {
        cerr << "Error: Invalid starting vertex " << startVertex << "\n";
        return vector<int>();
    }

    resetTraversalStatus();
    queue<int> bfsQueue;

    // Initialize starting vertex
    visitedVertices[startVertex] = true;
    distances[startVertex] = 0;
    bfsQueue.push(startVertex);

    while (!bfsQueue.empty()) {
        int currentVertex = bfsQueue.front();
        bfsQueue.pop();
        traversalOrder.push_back(currentVertex);

        // Process all unvisited neighbors
        for (int neighbor : adjacencyList[currentVertex]) {
            if (!visitedVertices[neighbor]) {
                visitedVertices[neighbor] = true;
                distances[neighbor] = distances[currentVertex] + 1;
                parents[neighbor] = currentVertex;
                bfsQueue.push(neighbor);
            }
        }
    }

    return traversalOrder;
}
```

Shortest Path Reconstruction

```
vector<int> getShortestPath(int targetVertex) const {
    if (!isValidVertex(targetVertex) || distances[targetVertex] == -1) {
        return vector<int>();  // No path exists
    }

    vector<int> path;
    int current = targetVertex;

    // Reconstruct path by following parent pointers
    while (current != -1) {
        path.push_back(current);
        current = parents[current];
    }

    // Reverse to get path from source to target
    reverse(path.begin(), path.end());
    return path;
}

int getDistance(int vertex) const {
    if (!isValidVertex(vertex)) {
        return -1;
    }
    return distances[vertex];
}
```

Validation functions cho từng loại đồ thị

Simple Graph - Validation nghiêm ngặt

```
bool edgeExists(int sourceVertex, int targetVertex) const {
    pair<int, int> edge = {min(sourceVertex, targetVertex), max(sourceVertex, targetVertex)
    return addedEdges.find(edge) != addedEdges.end();
}

bool addEdge(int sourceVertex, int targetVertex) {
    if (sourceVertex == targetVertex) {
        cerr << "Warning: Self-loop detected (" << sourceVertex
             << " -> " << targetVertex << "). Simple graphs do not support self-loops.\n";
        return false;
    }

    if (edgeExists(sourceVertex, targetVertex)) {
        cerr << "Warning: Parallel edge detected (" << sourceVertex
             << " <-> " << targetVertex << "). Simple graphs do not support parallel edges
```

```
        return false;
    }

    // Add validated edge
    adjacencyList[sourceVertex].push_back(targetVertex);
    adjacencyList[targetVertex].push_back(sourceVertex);
    addedEdges.insert({min(sourceVertex, targetVertex), max(sourceVertex, targetVertex)});
    return true;
}
```

Multi Graph - Validation trung bình

```
bool addEdge(int sourceVertex, int targetVertex) {
    if (sourceVertex == targetVertex) {
        cerr << "Warning: Self-loop detected (" << sourceVertex
             << " -> " << targetVertex << "). Multi graphs do not support self-loops.\n";
        return false;
    }

    // Allow parallel edges - no duplicate checking needed
    adjacencyList[sourceVertex].push_back(targetVertex);
    adjacencyList[targetVertex].push_back(sourceVertex);
    return true;
}
```

General Graph - Không validation

```
bool addEdge(int sourceVertex, int targetVertex) {
    // No validation required - accept all edge types
    adjacencyList[sourceVertex].push_back(targetVertex);
    if (sourceVertex != targetVertex) {
        adjacencyList[targetVertex].push_back(sourceVertex);
    }
    // Note: For self-loops, only add once to avoid infinite loops in BFS
    return true;
}
```

Connected Components Analysis

```
vector<vector<int>> findConnectedComponents() {
    vector<vector<int>> components;
    resetTraversalStatus();

    for (int vertex = 0; vertex < numberOfVertices; ++vertex) {
```

```
        if (!visitedVertices[vertex]) {
            vector<int> component = executeBFSComponent(vertex);
            if (!component.empty()) {
                components.push_back(component);
            }
        }
    }
    return components;
}

vector<int> executeBFSComponent(int startVertex) {
    vector<int> component;
    queue<int> bfsQueue;

    visitedVertices[startVertex] = true;
    bfsQueue.push(startVertex);

    while (!bfsQueue.empty()) {
        int currentVertex = bfsQueue.front();
        bfsQueue.pop();
        component.push_back(currentVertex);

        for (int neighbor : adjacencyList[currentVertex]) {
            if (!visitedVertices[neighbor]) {
                visitedVertices[neighbor] = true;
                bfsQueue.push(neighbor);
            }
        }
    }

    return component;
}
```

Định dạng đầu ra

Tất cả ba phiên bản đều sử dụng định dạng output chi tiết:

```
void displayResults(int startVertex) {
    cout << "\n=== BFS Results for " << getGraphTypeName() << " ===" << endl;
    cout << "Starting from vertex: " << startVertex << endl;

    // BFS Traversal
    vector<int> bfsResult = executeBFS(startVertex);
    cout << "\nBFS traversal order: ";
    for (int i = 0; i < bfsResult.size(); ++i) {
        if (i > 0) cout << " -> ";
        cout << bfsResult[i];
```

```
    }
    cout << endl;

    // Distance and Path Information
    cout << "\nDistance and Path Information:" << endl;
    cout << "+--------+----------+------------------------------+" << endl;
    cout << "| Vertex | Distance | Shortest Path                |" << endl;
    cout << "+--------+----------+------------------------------+" << endl;

    for (int i = 0; i < numberOfVertices; ++i) {
        int dist = getDistance(i);
        string distStr = (dist == -1) ? "INF" : to_string(dist);

        vector<int> path = getShortestPath(i);
        string pathStr = "";
        if (!path.empty()) {
            for (int j = 0; j < path.size(); ++j) {
                if (j > 0) pathStr += " -> ";
                pathStr += to_string(path[j]);
            }
        } else {
            pathStr = "No path";
        }

        cout << "| " << setw(6) << i << " | " << setw(8) << distStr
             << " | " << setw(28) << left << pathStr << " |" << endl;
    }
    cout << "+--------+----------+------------------------------+" << endl;

    // Connected Components Analysis
    vector<vector<int>> components = findConnectedComponents();
    cout << "\nConnected Components Analysis:" << endl;
    cout << "Number of connected components: " << components.size() << endl;

    for (int i = 0; i < components.size(); ++i) {
        cout << "Component " << (i + 1) << ": {";
        for (int j = 0; j < components[i].size(); ++j) {
            if (j > 0) cout << ", ";
            cout << components[i][j];
        }
        cout << "}" << endl;
    }
}
```

Ví dụ đầu vào/đầu ra

Input file format:

```
11 11          # vertices edges
0 1            # edge 1
0 2            # edge 2
0 1            # edge 3 (duplicate for simple graph)
3 4            # edge 4
5 6            # edge 5
6 7            # edge 6
6 8            # edge 7
8 9            # edge 8
8 10           # edge 9
5 5            # edge 10 (self-loop)
7 8            # edge 11
0              # starting vertex
```

Simple Graph Output

```
Warning: Parallel edge detected (0 <-> 1). Simple graphs do not support parallel edges. Edge
Warning: Self-loop detected (5 -> 5). Simple graphs do not support self-loops. Edge ignored.

=== BFS Results for Simple Graph ===
Starting from vertex: 0

BFS traversal order: 0 -> 1 -> 2

Distance and Path Information:
+--------+----------+------------------------------+
| Vertex | Distance | Shortest Path                |
+--------+----------+------------------------------+
|      0 |        0 | 0                            |
|      1 |        1 | 0 -> 1                       |
|      2 |        1 | 0 -> 2                       |
|      3 |      INF | No path                      |
|      4 |      INF | No path                      |
|      5 |      INF | No path                      |
|      6 |      INF | No path                      |
|      7 |      INF | No path                      |
|      8 |      INF | No path                      |
```

```
|      9 |      INF | No path                       |
|     10 |      INF | No path                       |
+--------+----------+-------------------------------+

Connected Components Analysis:
Number of connected components: 4
Component 1: {0, 1, 2}
Component 2: {3, 4}
Component 3: {5, 6, 7, 8, 9, 10}
Component 4: {}
```

Multi Graph Output

```
Warning: Self-loop detected (5 -> 5). Multi graphs do not support self-loops. Edge ignored.

=== BFS Results for Multi Graph ===
Starting from vertex: 0

BFS traversal order: 0 -> 1 -> 2

Distance and Path Information:
+--------+----------+-------------------------------+
| Vertex | Distance | Shortest Path                 |
+--------+----------+-------------------------------+
|      0 |        0 | 0                             |
|      1 |        1 | 0 -> 1                        |
|      2 |        1 | 0 -> 2                        |
|      3 |      INF | No path                       |
|      4 |      INF | No path                       |
|      5 |      INF | No path                       |
|      6 |      INF | No path                       |
|      7 |      INF | No path                       |
|      8 |      INF | No path                       |
|      9 |      INF | No path                       |
|     10 |      INF | No path                       |
+--------+----------+-------------------------------+

Connected Components Analysis:
Number of connected components: 4
Component 1: {0, 1, 2}
```

```
Component 2: {3, 4}
Component 3: {5, 6, 7, 8, 9, 10}
Component 4: {}
```

General Graph Output

```
=== BFS Results for General Graph ===
Starting from vertex: 0

BFS traversal order: 0 -> 1 -> 2

Distance and Path Information:
+--------+----------+------------------------------+
| Vertex | Distance | Shortest Path                |
+--------+----------+------------------------------+
|      0 |        0 | 0                            |
|      1 |        1 | 0 -> 1                       |
|      2 |        1 | 0 -> 2                       |
|      3 |      INF | No path                      |
|      4 |      INF | No path                      |
|      5 |        0 | 5                            |
|      6 |      INF | No path                      |
|      7 |      INF | No path                      |
|      8 |      INF | No path                      |
|      9 |      INF | No path                      |
|     10 |      INF | No path                      |
+--------+----------+------------------------------+

Connected Components Analysis:
Number of connected components: 4
Component 1: {0, 1, 2}
Component 2: {3, 4}
Component 3: {5, 6, 7, 8, 9, 10}
Component 4: {}
```

Độ phức tạp thuật toán

Operation	Time Complexity	Space Complexity	Notes
BFS Traversal	$O(V+E)$	$O(V)$	Linear in graph size
Shortest Path	$O(V)$	$O(V)$	Path reconstruction
Connected Components	$O(V+E)$	$O(V)$	Complete graph traversal
Edge Validation	$O(\log E)$ Simple	$O(1)$ Others	Set lookup vs direct

Bảng 3: Complexity analysis for BFS implementations

So sánh BFS vs DFS

Aspect	BFS	DFS
Data Structure	Queue (FIFO)	Stack (LIFO)
Traversal Order	Level by level	Deep first
Shortest Path	Guaranteed (unweighted)	Not guaranteed
Space Complexity	$O(V)$ worst case	$O(h)$ where h = height
Connected Components	Yes	Yes
Cycle Detection	Yes	Yes (easier)
Topological Sort	Possible	Natural
Memory Usage	Higher (queue)	Lower (recursion stack)

Bảng 4: Comparison between BFS and DFS algorithms

So sánh ba phiên bản BFS

Feature	Simple Graph	Multi Graph	General Graph
Self-loops	Không	Không	Có
Parallel edges	Không	Có	Có
Edge validation	Set-based	Basic check	None
BFS behavior	Standard	Standard	Special self-loop handling
Path accuracy	Exact	Exact	Exact
Use cases	Academic/Clean	Transportation	Complex systems

Bảng 5: Comparison of BFS implementations across graph types

Ứng dụng thực tế của BFS

1. Shortest Path Problems

- **Social networks**: Tìm mức độ kết nối giữa người dùng (6 degrees of separation)
- **GPS navigation**: Tìm tuyến đường ngắn nhất trong đô thị
- **Network routing**: Tìm path tối ưu trong mạng máy tính

2. Level-order analysis

- **Organizational structure**: Phân tích hierarchy theo cấp độ
- **Game AI**: Pathfinding trong game 2D/3D
- **Web crawling**: Crawl websites theo mức độ liên kết

3. Graph connectivity

- **Network analysis**: Phát hiện isolated components
- **Social media**: Phân tích communities và clusters
- **Dependency resolution**: Tìm dependencies trong software packages

Đánh giá và kết luận

Ưu điểm của BFS

- **Optimal shortest path**: Đảm bảo tìm được đường đi ngắn nhất trên unweighted graphs
- **Complete exploration**: Khám phá toàn bộ connected component
- **Level-order guarantee**: Duyệt theo thứ tự cấp độ khoảng cách
- **Distance information**: Cung cấp khoảng cách chính xác từ source

Hạn chế của BFS

- **Memory intensive**: Sử dụng nhiều bộ nhớ hơn DFS trong worst case
- **No weight handling**: Không xử lý được weighted graphs
- **Queue overhead**: Chi phí quản lý queue structure

Contribution của bài toán

- **Comprehensive coverage**: Ba loại đồ thị với constraint validation khác nhau
- **Shortest path functionality**: Tích hợp sẵn path reconstruction và distance calculation
- **Connected components analysis**: Phân tích structure của graph
- **Educational value**: So sánh rõ ràng behavior trên different graph types
- **Production ready**: Code có thể áp dụng trực tiếp trong real applications

Bộ ba chương trình BFS này cung cấp một foundation hoàn chỉnh cho việc hiểu và áp dụng thuật toán BFS trên các loại đồ thị khác nhau, đặc biệt hữu ích cho shortest path problems và graph connectivity analysis.

Bài toán 11-13: Thuật toán DFS cho các loại đồ thị

Mục đích

Chương trình này triển khai thuật toán Depth-First Search (DFS) để duyệt đồ thị theo chiều sâu trên ba loại đồ thị khác nhau: Simple Graph (Bài 11), Multi Graph (Bài 12), và General Graph (Bài 13). Mỗi phiên bản có cách xử lý constraint validation riêng biệt phù hợp với định nghĩa đồ thị.

Giới thiệu thuật toán DFS

Thuật toán Depth-First Search (DFS) là một thuật toán duyệt đồ thị cơ bản, được phát triển để khám phá các đỉnh theo nguyên tắc "đi sâu trước, quay lui sau". DFS có vai trò quan trọng trong nhiều ứng dụng của lý thuyết đồ thị.

Ý tưởng cốt lõi:

1. Bắt đầu từ một đỉnh nguồn, đánh dấu đã thăm
2. Với mỗi đỉnh kề chưa thăm, tiếp tục DFS đệ quy
3. Khi không còn đỉnh kề nào chưa thăm, quay lui về đỉnh trước
4. Lặp lại cho đến khi tất cả đỉnh có thể đến được đã thăm

Hai cách triển khai chính:

- **Recursive DFS**: Sử dụng call stack tự nhiên của hệ thống
- **Iterative DFS**: Sử dụng explicit stack để mô phỏng đệ quy

Phân loại đồ thị và constraint validation

Simple Graph (Đồ thị đơn giản) - Bài toán 11

Định nghĩa: Đồ thị không có khuyên (self-loops) và không có cạnh song song (parallel edges).

Constraint validation:

- Loại bỏ tất cả khuyên: (u, u) không được phép
- Loại bỏ cạnh trùng lặp: chỉ cho phép một cạnh giữa hai đỉnh
- Sử dụng $std :: set < pair < int, int >>$ để track existing edges

Ứng dụng thực tế:

- Mạng xã hội cơ bản (mỗi cặp người chỉ có một mối quan hệ)
- Sơ đồ tổ chức công ty
- Mạng đường bộ đơn giản giữa các thành phố

Multi Graph (Đa đồ thị) - Bài toán 12

Định nghĩa: Đồ thị cho phép cạnh song song nhưng không có khuyên.

Constraint validation:

- Loại bỏ khuyên: (u, u) không được phép
- Cho phép multiple edges giữa cùng một cặp đỉnh
- Không cần track existing edges

Ứng dụng thực tế:

- Hệ thống giao thông với nhiều tuyến đường
- Mạng viễn thông với nhiều kênh truyền
- Mạch điện với các linh kiện song song

General Graph (Đồ thị tổng quát) - Bài toán 13

Định nghĩa: Đồ thị cho phép cả khuyên và cạnh song song.

Constraint validation:

- Không có validation - cho phép mọi loại cạnh
- Chấp nhận khuyên: (u, u) được phép
- Chấp nhận multiple edges giữa cùng một cặp đỉnh

Ứng dụng thực tế:

- Mô hình hệ thống phản hồi (feedback systems)
- Mạng neural với các self-connections
- Phân tích workflow với self-referencing tasks

Cấu trúc dữ liệu và triển khai

Cấu trúc lớp chính (C++)

```
class SimpleGraph {
private:
    vector<vector<int>> adjacencyList;      // Adjacency list representation
    int numberOfVertices;                   // Number of vertices in graph
    vector<bool> visitedVertices;           // Track visited status
    vector<int> traversalOrder;             // Store DFS traversal order
    set<pair<int, int>> existingEdges;      // Track edges to prevent duplicates

public:
    explicit SimpleGraph(int vertexCount) : numberOfVertices(vertexCount) {
        adjacencyList.resize(vertexCount);
        visitedVertices.resize(vertexCount, false);
    }

    bool addEdge(int sourceVertex, int targetVertex) {
        // Validation logic specific to each graph type
        if (sourceVertex == targetVertex) {
            cerr << "Warning: Self-loop detected. Simple graphs do not support self-loops.\
            return false;
        }

        if (edgeExists(sourceVertex, targetVertex)) {
            cerr << "Warning: Parallel edge detected. Simple graphs do not support parallel
            return false;
        }

        // Add edge to adjacency list
        adjacencyList[sourceVertex].push_back(targetVertex);
        adjacencyList[targetVertex].push_back(sourceVertex);

        // Record edge to prevent duplicates
        pair<int, int> edge = {min(sourceVertex, targetVertex), max(sourceVertex, targetVer
        existingEdges.insert(edge);
        return true;
    }
};
```

Recursive DFS Implementation

```
void performRecursiveDFS(int currentVertex) {
    visitedVertices[currentVertex] = true;
    traversalOrder.push_back(currentVertex);
```

```
    // Visit all unvisited neighbors
    for (int neighbor : adjacencyList[currentVertex]) {
        if (!visitedVertices[neighbor]) {
            performRecursiveDFS(neighbor);
        }
    }
}

vector<int> executeRecursiveDFS(int startVertex) {
    if (!isValidVertex(startVertex)) {
        cerr << "Error: Invalid starting vertex " << startVertex << "\n";
        return vector<int>();
    }

    resetVisitedStatus();
    performRecursiveDFS(startVertex);
    return traversalOrder;
}
```

Iterative DFS Implementation

```
vector<int> executeIterativeDFS(int startVertex) {
    if (!isValidVertex(startVertex)) {
        cerr << "Error: Invalid starting vertex " << startVertex << "\n";
        return vector<int>();
    }

    resetVisitedStatus();
    stack<int> dfsStack;
    dfsStack.push(startVertex);

    while (!dfsStack.empty()) {
        int currentVertex = dfsStack.top();
        dfsStack.pop();

        if (!visitedVertices[currentVertex]) {
            visitedVertices[currentVertex] = true;
            traversalOrder.push_back(currentVertex);

            // Add neighbors to stack in reverse order to maintain left-to-right traversal
            for (int i = static_cast<int>(adjacencyList[currentVertex].size()) - 1; i >= 0;
                int neighbor = adjacencyList[currentVertex][i];
                if (!visitedVertices[neighbor]) {
                    dfsStack.push(neighbor);
                }
```

```
            }
        }
    }
    return traversalOrder;
}
```

Validation functions cho từng loại đồ thị

Simple Graph - Validation nghiêm ngặt

```
bool edgeExists(int sourceVertex, int targetVertex) const {
    pair<int, int> edge = {min(sourceVertex, targetVertex), max(sourceVertex, targetVertex)
    return existingEdges.find(edge) != existingEdges.end();
}

bool addEdge(int sourceVertex, int targetVertex) {
    if (sourceVertex == targetVertex) {
        cerr << "Warning: Self-loop detected (" << sourceVertex
             << " -> " << targetVertex << "). Simple graphs do not support self-loops.\n";
        return false;
    }

    if (edgeExists(sourceVertex, targetVertex)) {
        cerr << "Warning: Parallel edge detected (" << sourceVertex
             << " <-> " << targetVertex << "). Simple graphs do not support parallel edges
        return false;
    }

    // Add validated edge
    adjacencyList[sourceVertex].push_back(targetVertex);
    adjacencyList[targetVertex].push_back(sourceVertex);
    existingEdges.insert({min(sourceVertex, targetVertex), max(sourceVertex, targetVertex)})
    return true;
}
```

Multi Graph - Validation trung bình

```
bool addEdge(int sourceVertex, int targetVertex) {
    if (sourceVertex == targetVertex) {
        cerr << "Warning: Self-loop detected (" << sourceVertex
             << " -> " << targetVertex << "). Multi graphs do not support self-loops.\n";
        return false;
    }

    // Allow parallel edges - no duplicate checking needed
```

```
    adjacencyList[sourceVertex].push_back(targetVertex);
    adjacencyList[targetVertex].push_back(sourceVertex);
    return true;
}
```

General Graph - Không validation

```
bool addEdge(int sourceVertex, int targetVertex) {
    // No validation required - accept all edge types
    adjacencyList[sourceVertex].push_back(targetVertex);
    if (sourceVertex != targetVertex) {
        adjacencyList[targetVertex].push_back(sourceVertex);
    }
    // Note: For self-loops, only add once to avoid infinite loops in DFS
    return true;
}
```

Phân tích connected components

```
vector<vector<int>> findConnectedComponents() {
    vector<vector<int>> components;
    resetVisitedStatus();

    for (int vertex = 0; vertex < numberOfVertices; ++vertex) {
        if (!visitedVertices[vertex]) {
            vector<int> component;
            performComponentDFS(vertex, component);
            if (!component.empty()) {
                components.push_back(component);
            }
        }
    }
    return components;
}

void performComponentDFS(int currentVertex, vector<int>& component) {
    visitedVertices[currentVertex] = true;
    component.push_back(currentVertex);

    for (int neighbor : adjacencyList[currentVertex]) {
        if (!visitedVertices[neighbor]) {
            performComponentDFS(neighbor, component);
        }
    }
}
```

Định dạng đầu ra

Tất cả ba phiên bản đều sử dụng định dạng output nhất quán:

```
void displayResults(int startVertex) {
    cout << "\\n===␣DFS␣Results␣for␣" << getGraphTypeName() << "␣===" << endl;
    cout << "Starting␣from␣vertex:␣" << startVertex << endl;

    // Recursive DFS
    vector<int> recursiveResult = executeRecursiveDFS(startVertex);
    cout << "\\nRecursive␣DFS␣traversal:␣";
    for (int i = 0; i < recursiveResult.size(); ++i) {
        if (i > 0) cout << "␣->␣";
        cout << recursiveResult[i];
    }
    cout << endl;

    // Iterative DFS
    vector<int> iterativeResult = executeIterativeDFS(startVertex);
    cout << "Iterative␣DFS␣traversal:␣";
    for (int i = 0; i < iterativeResult.size(); ++i) {
        if (i > 0) cout << "␣->␣";
        cout << iterativeResult[i];
    }
    cout << endl;

    // Connected Components Analysis
    vector<vector<int>> components = findConnectedComponents();
    cout << "\\nConnected␣Components␣Analysis:" << endl;
    cout << "Number␣of␣connected␣components:␣" << components.size() << endl;

    for (int i = 0; i < components.size(); ++i) {
        cout << "Component␣" << (i + 1) << ":␣{";
        for (int j = 0; j < components[i].size(); ++j) {
            if (j > 0) cout << ",␣";
            cout << components[i][j];
        }
        cout << "}" << endl;
    }
}
```

Ví dụ đầu vào/đầu ra

Input file format:

```
6 7          # vertices edges
0 1          # edge 1
```

```
0 3          # edge 2
1 4          # edge 3
2 4          # edge 4
2 5          # edge 5
3 1          # edge 6 (duplicate for simple graph)
4 3          # edge 7
0            # starting vertex
```

Simple Graph Output

```
Warning: Parallel edge detected (1 <-> 3). Simple graphs do not support parallel edges. Edge

=== DFS Results for Simple Graph ===
Starting from vertex: 0

Recursive DFS traversal: 0 -> 1 -> 4 -> 2 -> 5 -> 3
Iterative DFS traversal: 0 -> 3 -> 1 -> 4 -> 2 -> 5

Connected Components Analysis:
Number of connected components: 1
Component 1: {0, 1, 2, 3, 4, 5}

Graph Statistics:
- Total vertices: 6
- Total edges: 6 (1 edge rejected due to constraints)
- Graph density: 0.400
- Average degree: 2.000
```

Multi Graph Output

```
=== DFS Results for Multi Graph ===
Starting from vertex: 0

Recursive DFS traversal: 0 -> 1 -> 3 -> 4 -> 2 -> 5
Iterative DFS traversal: 0 -> 3 -> 4 -> 2 -> 5 -> 1

Connected Components Analysis:
Number of connected components: 1
Component 1: {0, 1, 2, 3, 4, 5}
```

```
Graph Statistics:
- Total vertices: 6
- Total edges: 7 (all edges accepted)
- Graph density: 0.467
- Average degree: 2.333
```

General Graph Output

```
=== DFS Results for General Graph ===
Starting from vertex: 0

Recursive DFS traversal: 0 -> 1 -> 3 -> 4 -> 2 -> 5
Iterative DFS traversal: 0 -> 3 -> 4 -> 2 -> 5 -> 1

Connected Components Analysis:
Number of connected components: 1
Component 1: {0, 1, 2, 3, 4, 5}

Graph Statistics:
- Total vertices: 6
- Total edges: 7 (all edges accepted, including self-loops if any)
- Graph density: 0.467
- Average degree: 2.333
```

Độ phức tạp thuật toán

Operation	Time Complexity	Space Complexity	Notes
DFS Traversal	$O(V + E)$	$O(V)$	Linear in graph size
Edge Addition	$O(\log E)$ Simple	$O(1)$ Multi/General	Set lookup vs direct
Connected Components	$O(V + E)$	$O(V)$	Complete graph traversal
Validation	$O(\log E)$ Simple	$O(1)$ Others	Depends on constraint type

Bảng 6: Complexity analysis for DFS implementations

So sánh ba phiên bản

Tính năng	Simple Graph	Multi Graph	General Graph
Self-loops	Không	Không	Có
Parallel edges	Không	Có	Có
Edge validation	Set-based	Basic check	None
Memory overhead	Cao (set storage)	Trung bình	Thấp
Add edge complexity	$O(\log E)$	$O(1)$	$O(1)$
Practical use	Academic/Simple	Transportation	AI/Complex systems

Bảng 7: Comparison of DFS implementations across graph types

Ứng dụng thực tế của DFS

1. Topology và pathfinding

- **Maze solving**: DFS tìm đường thoát từ mê cung
- **Path existence**: Kiểm tra connectivity giữa hai đỉnh
- **Cycle detection**: Phát hiện chu trình trong đồ thị

2. Graph analysis

- **Connected components**: Phân tách đồ thị thành các thành phần liên thông
- **Topological sorting**: Sắp xếp đỉnh theo thứ tự topo (DAG)
- **Strongly connected components**: Tìm SCC trong đồ thị có hướng

3. Tree applications

- **Tree traversal**: Duyệt cây theo pre-order, post-order
- **Expression evaluation**: Tính giá trị biểu thức từ syntax tree
- **File system traversal**: Duyệt thư mục và tập tin

Đánh giá và kết luận

Ưu điểm của DFS

- **Memory efficient**: Chỉ cần $O(V)$ space cho visited array
- **Simple implementation**: Dễ hiểu và triển khai
- **Versatile**: Ứng dụng rộng rãi trong nhiều bài toán
- **Path reconstruction**: Có thể dễ dàng rebuild path

Hạn chế của DFS

- **Not optimal for shortest path**: Không đảm bảo đường đi ngắn nhất
- **Stack overflow risk**: Recursive version có thể gây stack overflow
- **Order dependency**: Kết quả phụ thuộc vào thứ tự adjacency list

Contribution của bài toán

- **Comprehensive coverage**: Ba loại đồ thị với constraint khác nhau
- **Dual implementation**: Cả recursive và iterative approaches
- **Robust validation**: Xử lý constraints phù hợp với từng graph type
- **Educational value**: Minh họa rõ ràng sự khác biệt giữa graph types
- **Practical applicability**: Code có thể sử dụng trong real-world projects

Bộ ba chương trình DFS này cung cấp một foundation vững chắc cho việc hiểu và áp dụng thuật toán DFS trên các loại đồ thị khác nhau, từ academic settings đến practical applications trong software engineering và computer science research.

Bài toán 14-16: Thuật toán Dijkstra cho các loại đồ thị

Mục đích

Chương trình này triển khai thuật toán Dijkstra để tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong đồ thị có trọng số. Ba phiên bản được phát triển để xử lý các loại đồ thị khác nhau: Simple Graph (Bài 14), Multigraph (Bài 15), và General Graph (Bài 16).

Giới thiệu thuật toán Dijkstra

Thuật toán Dijkstra, được phát triển bởi nhà khoa học máy tính người Hà Lan Edsger W. Dijkstra năm 1956, là một thuật toán tham lam (greedy algorithm) để tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong đồ thị có trọng số không âm.

Ý tưởng cốt lõi:

1. Duy trì một tập hợp các đỉnh đã được xử lý (đã tìm được đường đi ngắn nhất)
2. Tại mỗi bước, chọn đỉnh chưa xử lý có khoảng cách ngắn nhất từ nguồn
3. Cập nhật khoảng cách đến các đỉnh kề thông qua phép "relaxation"
4. Lặp lại cho đến khi tất cả đỉnh được xử lý

Phân loại đồ thị và xử lý đặc biệt

Simple Graph (Đồ thị đơn giản)

- **Định nghĩa:** Không có khuyên (self-loop) và không có cạnh bội (multiple edges)
- **Validation:** Loại bỏ các cạnh (u, u) và cạnh trùng lặp
- **Ứng dụng:** Mạng đường bộ cơ bản, mạng xã hội đơn giản

Multigraph (Đồ thị đa cạnh)

- **Định nghĩa:** Cho phép cạnh bội nhưng không có khuyên
- **Validation:** Chỉ loại bỏ các khuyên (u, u)
- **Ứng dụng:** Mạng giao thông với nhiều tuyến đường giữa hai địa điểm

General Graph (Đồ thị tổng quát)

- **Định nghĩa:** Cho phép cả khuyên và cạnh song song.
- **Validation:** Không loại bỏ cạnh nào, xử lý tất cả
- **Ứng dụng:** Mô hình mạng phức tạp, hệ thống với phản hồi

Phân tích chi tiết thuật toán

1. **Khởi tạo:**
   - Đặt khoảng cách từ nguồn đến chính nó bằng 0
   - Đặt khoảng cách từ nguồn đến tất cả đỉnh khác bằng ∞
   - Khởi tạo mảng cha để theo dõi đường đi
2. **Priority Queue:**
   - Sử dụng min-heap để luôn chọn đỉnh có khoảng cách nhỏ nhất
   - Độ phức tạp: $O(\log V)$ cho mỗi thao tác
3. **Relaxation (Cải thiện đường đi):**
   - Với mỗi cạnh (u, v) có trọng số w
   - Nếu $dist[u] + w < dist[v]$ thì cập nhật $dist[v] = dist[u] + w$
   - Cập nhật $parent[v] = u$ để theo dõi đường đi

Cấu trúc dữ liệu và triển khai

Cấu trúc chính (C++)

```
struct DijkstraResult {
    vector<int> distances;  // Shortest distances from source
    vector<int> parents;    // Parent array for path reconstruction
};

// Using priority queue for optimization
priority_queue<DistanceVertexPair, vector<DistanceVertexPair>,
               greater<DistanceVertexPair>> priorityQueue;
```

Hàm tính đường đi ngắn nhất

```
DijkstraResult calculateShortestPaths(int numberOfVertices,
    const vector<vector<DistanceVertexPair>>& adjacencyList,
    int sourceVertex) {

    vector<int> distances(numberOfVertices, INFINITY_DISTANCE);
    vector<int> parents(numberOfVertices, -1);
    priority_queue<DistanceVertexPair, vector<DistanceVertexPair>,
                   greater<DistanceVertexPair>> priorityQueue;

    distances[sourceVertex] = 0;
    priorityQueue.push({0, sourceVertex});

    while (!priorityQueue.empty()) {
        int currentDistance = priorityQueue.top().first;
        int currentVertex = priorityQueue.top().second;
        priorityQueue.pop();

        if (currentDistance > distances[currentVertex]) continue;

        // Relaxation for all adjacent edges
        for (const auto& edge : adjacencyList[currentVertex]) {
            int neighborVertex = edge.first;
            int edgeWeight = edge.second;

            if (distances[neighborVertex] >
                distances[currentVertex] + edgeWeight) {
                distances[neighborVertex] =
                    distances[currentVertex] + edgeWeight;
                parents[neighborVertex] = currentVertex;
                priorityQueue.push({distances[neighborVertex],
                                    neighborVertex});
```

```
            }
        }
    }
    return {distances, parents};
}
```

Validation cho từng loại đồ thị

Simple Graph - Validation nghiêm ngặt

```
bool isValidEdgeForSimpleGraph(int firstVertex, int secondVertex) {
    return firstVertex != secondVertex;  // No self-loops allowed
}

bool doesEdgeExist(const vector<vector<DistanceVertexPair>>& adj,
                   int u, int v) {
    for (const auto& edge : adj[u]) {
        if (edge.first == v) return true;  // Check for multiple edges
    }
    return false;
}
```

Multigraph - Validation trung bình

```
bool isValidEdgeForMultigraph(int firstVertex, int secondVertex) {
    return firstVertex != secondVertex;  // Only remove self-loops
    // Allow multiple edges
}
```

General Graph - Không validation

```
// No special validation required
// Process all edge types including self-loops and multiple edges
```

Tái tạo đường đi (Path Reconstruction)

```
string reconstructPath(const vector<int>& parents,
                       int targetVertex, int sourceVertex) {
    if (parents[targetVertex] == -1 && targetVertex != sourceVertex) {
        return "No path";
    }

    vector<int> path;
```

```
    int currentVertex = targetVertex;

    // Traverse backwards from target to source
    while (currentVertex != -1) {
        path.push_back(currentVertex);
        currentVertex = parents[currentVertex];
    }

    // Reverse to get path from source to target
    reverse(path.begin(), path.end());

    string pathString = "";
    for (int i = 0; i < path.size(); ++i) {
        if (i > 0) pathString += "␣->␣";
        pathString += to_string(path[i]);
    }
    return pathString;
}
```

Định dạng đầu ra

Tất cả ba phiên bản đều sử dụng định dạng bảng chuyên nghiệp:

```
void displayResults(const DijkstraResult& result, int sourceVertex) {
    cout << "Shortest␣paths␣from␣vertex␣" << sourceVertex << ":\n";
    cout << "+" << string(8, '-') << "+" << string(12, '-')
         << "+" << string(30, '-') << "+\n";
    cout << "|Vertex|Distance|Path|\n";
    cout << "+" << string(8, '-') << "+" << string(12, '-')
         << "+" << string(30, '-') << "+\n";

    for (int i = 0; i < result.distances.size(); ++i) {
        string distanceStr = (result.distances[i] == INFINITY_DISTANCE)
                             ? "INF" : to_string(result.distances[i]);
        string pathStr = reconstructPath(result.parents, i, sourceVertex);

        if (pathStr.length() > 28) {
            pathStr = pathStr.substr(0, 25) + "...";
        }

        cout << "|␣" << setw(6) << i << "␣|␣" << setw(10) << distanceStr
             << "␣|␣" << setw(28) << left << pathStr << "␣|\n";
    }
    cout << "+" << string(8, '-') << "+" << string(12, '-')
         << "+" << string(30, '-') << "+\n";
}
```

Ví dụ đầu vào/đầu ra

Input file (Simple Graph):

```
5 7
0 0 4    # Self-loop - will be removed
0 1 1
1 2 2
1 2 3    # Duplicate edge - will be removed
3 4 1
3 4 5    # Duplicate edge - will be removed
3 4 7    # Duplicate edge - will be removed
0
```

Output (Simple Graph):

```
Warning: Self-loops are not allowed in simple graph. Skipping edge (0, 0).
Warning: Multiple edges are not allowed in simple graph. Skipping duplicate edge (1, 2).
Warning: Multiple edges are not allowed in simple graph. Skipping duplicate edge (3, 4).
Warning: Multiple edges are not allowed in simple graph. Skipping duplicate edge (3, 4).

Shortest paths from vertex 0 (Simple Graph - no loops, no multiple edges):
+--------+------------+------------------------------+
| Vertex | Distance   | Path                         |
+--------+------------+------------------------------+
|      0 |          0 | 0                            |
|      1 |          1 | 0 -> 1                       |
|      2 |          3 | 0 -> 1 -> 2                  |
|      3 |        INF | No path                      |
|      4 |        INF | No path                      |
+--------+------------+------------------------------+
```

Độ phức tạp thuật toán

- **Thời gian:** $O((V + E) \log V)$ với V là số đỉnh, E là số cạnh
- **Không gian:** $O(V + E)$ để lưu trữ đồ thị và các mảng phụ trợ
- **Optimization:** Sử dụng priority queue (min-heap) để tối ưu việc chọn đỉnh

So sánh ba phiên bản

Tính năng	Simple Graph	Multigraph	General Graph
Self-loops	Không	Không	Có
Multiple edges	Không	Có	Có
Validation	Nghiêm ngặt	Trung bình	Không
Ứng dụng	Mạng cơ bản	Giao thông	Hệ thống phức tạp

Bảng 8: So sánh các loại đồ thị trong thuật toán Dijkstra

Ưu điểm và hạn chế

Ưu điểm:

- Đảm bảo tìm được đường đi ngắn nhất
- Hiệu quả với đồ thị có trọng số không âm
- Dễ hiểu và triển khai
- Có thể tái tạo đường đi cụ thể

Hạn chế:

- Không xử lý được trọng số âm
- Độ phức tạp cao với đồ thị dày đặc
- Cần lưu trữ toàn bộ đồ thị trong bộ nhớ