数据结构实验报告 10

图

姓　　名：南亚宏

学　　号：2311788

目录

2025 年 12 月 24 日

一、 题目一

1.1 题目表述

使用 BFS 而不是 DFS，编写程序 16-8 的另一个版本。证明由这个版本所得到的从 theSource 到 theDestination 的路径是最短路径。

程序 16-8 在图中寻找一条路径的前序方法

```
int* findPath(int theSource, int theDestination)
{// 寻找一条从顶点 theSource 到顶点 theDestination 的路径
 // 返回一个数组 path，从索引 1 开始表示路径。path[0] 表示路径长度
```



```
 // 如果路径不存在，返回 NULL
 // 为寻找路径的递归算法初始化
   int n = numberOfVertices();
   path = new int [n + 1];
   path[1] = theSource;        // 第一个顶点总是
   length = 1;                 // 当前路径长度 + 1
   destination = theDestination;
   reach = new int [n + 1];
   for (int i = 1; i <= n; i++)
      reach[i] = 0;

   // 搜索路径
   if (theSource == theDestination || rFindPath(theSource))
     // 找到一条路径
      path[0] = length - 1;
   else
   {
      delete [] path;
      path = NULL;
   }

   delete [] reach;
   return path;
}
```

1.2 代码 1 的测试用例:

- findPath(s,t) 返回 path 数组（从下标 1 开始存路径顶点）
- path[0] 表示路径长度（边数）

用例 1：顶点 {1}，边集 {}

- findPath(G0, 1, 1) -> path[0]=0, path[1]=1

用例 2：顶点 {1,2,3}，边 {(1,2)}

- findPath(G, 1, 3) -> NULL

用例 3：顶点 {1,2,3,4,5}，边 {(1,2),(2,3),(3,4),(4,5)}

- findPath(G, 1, 5) -> path[0]=4, path[1..5]=(1,2,3,4,5)

用例 4：顶点 {1,2,3,4,5}，边 {(1,2),(2,5),(1,3),(3,4),(4,5)}

- findPath(G, 1, 5) -> path[0]=2, path[1..3]=(1,2,5)

用例 5：顶点 {1,2,3,4}，边 {(1,2),(2,4),(1,3),(3,4)}

- findPath(G, 1, 4) -> path[0]=2, path[1..3]=(1,2,4) 或 path[0]=2, path[1..3]=(1,3,4)

1.3 思路:

记录每个顶点的前驱顶点（用于回溯路径），再记录每个顶点的距离（从 theSource 到该顶点的边数），最后使用队列实现 BFS 的层序遍历。
从 theSource 出发，逐层访问邻接顶点，更新前驱和距离。若到达 theDestination，立即终止遍历（BFS 保证此时的路径是最短的）。
若 theDestination 不可达，返回 NULL；若可达，通过“前驱数组”从 theDestination 回溯到 theSource，再逆序得到路径数组。

证明“BFS 得到的是最短路径”：
BFS 的核心特性是按层遍历。
第 1 层：距离 theSource 为 0 的顶点（即 theSource 自身）。
第 2 层：距离 theSource 为 1 的顶点（直接邻接的顶点）。
第 k 层：距离 theSource 为 k-1 的顶点。
当 BFS 首次访问到 theDestination 时，它一定处于距离最小的层（因为 BFS 先遍历所有距离更小的层）。因此，此时通过“前驱数组”回溯得到的路径，是从 theSource 到 theDestination 的最短路径（边数最少）。

1.4代码:

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

// 图的类封装
class Graph {
private:
    vector<vector<int>> adj;  // 邻接表（顶点从 1 开始）
    int vertexCount;          // 顶点总数
```

```
public:
    // 构造函数：初始化顶点数
    Graph(int n) : vertexCount(n) {
        adj.resize(n + 1);  // 索引 0 不用，1~n 对应顶点
    }

    // 添加无向边（题目中边默认无向，若有向则只加 u→v）
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    // BFS 实现 findPath，返回路径数组（动态分配）
    int* findPath(int theSource, int theDestination) {
        // 特殊情况：起点=终点
        if (theSource == theDestination) {
            int* path = new int[2];  // path[0]=长度，path[1]=顶
点
            path[0] = 0;
            path[1] = theSource;
            return path;
        }

        // 初始化辅助数组
        int* predecessor = new int[vertexCount + 1];  // 前驱顶
点
        int* distance = new int[vertexCount + 1];     // 距离
（边数）
        bool* visited = new bool[vertexCount + 1];    // 访问标
记
        queue<int> q;

        for (int i = 1; i <= vertexCount; ++i) {
            predecessor[i] = -1;
            distance[i] = -1;
            visited[i] = false;
        }

        // 起点初始化
        distance[theSource] = 0;
        visited[theSource] = true;
        q.push(theSource);
```

```
        bool found = false;
        while (!q.empty() && !found) {
            int u = q.front();
            q.pop();

            // 遍历 u 的所有邻接顶点
            for (int v : adj[u]) {
                if (!visited[v]) {
                    visited[v] = true;
                    predecessor[v] = u;
                    distance[v] = distance[u] + 1;
                    q.push(v);

                    // 找到终点，终止 BFS
                    if (v == theDestination) {
                        found = true;
                        break;
                    }
                }
            }
        }

        // 终点不可达，释放资源并返回 NULL
        if (!found) {
            delete[] predecessor;
            delete[] distance;
            delete[] visited;
            return nullptr;
        }

        // 回溯路径：从终点到起点，再逆序
        int pathLen = distance[theDestination];
        int* path = new int[pathLen + 2];  // path[0] + 路径顶点
（1~pathLen+1）
        path[0] = pathLen;
        int current = theDestination;
        for (int i = pathLen + 1; i >= 1; --i) {
            path[i] = current;
            current = predecessor[current];
        }

        // 释放辅助数组
```

```
        delete[] predecessor;
        delete[] distance;
        delete[] visited;
        return path;
    }

    // 释放图资源（可选）
    ~Graph() {
        adj.clear();
    }
};

// 打印路径结果
void printPathResult(int* path, const string& testCaseName) {
    cout << "【测试用例" << testCaseName << "】" << endl;
    if (path == nullptr) {
        cout << "结果：NULL" << endl;
    } else {
        int len = path[0];
        cout << "路径长度（边数）：" << len << endl;
        cout << "路径顶点：";
        for (int i = 1; i <= len + 1; ++i) {
            cout << path[i] << (i == len + 1 ? "" : " -> ");
        }
        cout << endl;
        delete[] path;  // 释放动态分配的路径数组
    }
    cout << "-------------------------" << endl;
}

// 主函数：测试用例
int main() {
    // ========== 测试用例 1：顶点{1}，边集{} ==========
    Graph G1(1);
    int* path1 = G1.findPath(1, 1);
    printPathResult(path1, "1");

    // ========== 测试用例 2：顶点{1,2,3}，边{(1,2)} ==========
    Graph G2(3);
    G2.addEdge(1, 2);
    int* path2 = G2.findPath(1, 3);
    printPathResult(path2, "2");
```

```
    // ========== 测试用例 3：顶点{1,2,3,4,5}，边
{(1,2),(2,3),(3,4),(4,5)} ==========
    Graph G3(5);
    G3.addEdge(1, 2);
    G3.addEdge(2, 3);
    G3.addEdge(3, 4);
    G3.addEdge(4, 5);
    int* path3 = G3.findPath(1, 5);
    printPathResult(path3, "3");

    // ========== 测试用例 4：顶点{1,2,3,4,5}，边
{(1,2),(2,5),(1,3),(3,4),(4,5)} ==========
    Graph G4(5);
    G4.addEdge(1, 2);
    G4.addEdge(2, 5);
    G4.addEdge(1, 3);
    G4.addEdge(3, 4);
    G4.addEdge(4, 5);
    int* path4 = G4.findPath(1, 5);
    printPathResult(path4, "4");

    // ========== 测试用例 5：顶点{1,2,3,4}，边
{(1,2),(2,4),(1,3),(3,4)} ==========
    Graph G5(4);
    G5.addEdge(1, 2);
    G5.addEdge(2, 4);
    G5.addEdge(1, 3);
    G5.addEdge(3, 4);
    int* path5 = G5.findPath(1, 4);
    printPathResult(path5, "5");

    return 0;
}
```

1.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw10\hw10.1> ./main.exe
【测试用例1】
路径长度（边数）：0
路径顶点：1
-------------------------
【测试用例2】
结果：NULL
-------------------------
【测试用例3】
路径长度（边数）：4
路径顶点：1 -> 2 -> 3 -> 4 -> 5
-------------------------
【测试用例4】
路径长度（边数）：2
路径顶点：1 -> 2 -> 5
-------------------------
【测试用例5】
路径长度（边数）：2
路径顶点：1 -> 2 -> 4
-------------------------
PS C:\Users\Macro\Desktop\hw10\hw10.1>
```

二、 题目二

2.1 题目表述

设 G 是一个无向图。它的传递闭包（tansitive closure）是一个 0/1 数组 tc，当且仅当 G 存在一条边数大于 1 的从 i 到 j 的路径时，tc[i][j]=1。编写一个方法 graph::undirectedTC()，计算且返回 G 的传递闭包。方法的复杂性应为 $O(n^2)$，其中 n 是 G 的顶点数。（提示：采用构件标记策略。）

2.2 代码 2 的测试用例:

约定：tc[i][j]=1 当且仅当存在一条从 i 到 j 的路径，且路径边数 > 1；否则为 0。默认对角线 tc[i][i]=0

- 用例 1：V={1,2,3}，E=∅ 调用：tc = undirectedTC()

期望： tc 矩阵（行 i 列 j）

i=1: (0,0,0)

i=2: (0,0,0)

i=3: (0,0,0)

- 用例 2：V={1,2}，E={(1,2)}调用：tc = undirectedTC()

i=1: (0,0)

i=2: (0,0)

- 用例 3：V={1,2,3}，E={(1,2),(2,3)} 调用：tc = undirectedTC()

期望：

i=1: (0,0,1)

i=2: (0,0,0)

i=3: (1,0,0)

- 用例 4：V={1,2,3,4}，E={(1,2),(1,3),(1,4)}调用：tc = undirectedTC()

期望：

i=1: (0,0,0,0)

i=2: (0,0,1,1)

i=3: (0,1,0,1)

i=4: (0,1,1,0)

- 用例 5：V={1,2,3}，E={(1,2),(2,3),(1,3)} 调用：tc = undirectedTC()

期望：

i=1: (0,1,1)

i=2: (1,0,1)

i=3: (1,1,0)

2.3 思路:

对图进行连通分量标记，每个顶点属于一个连通分量。然后检查每个点，在对角线上则为 0；若 i 和 j 不在同一连通分量为 0，若 i 和 j 在同一连通分量且存在长度>1 的路径（即连通分量大小>2，或虽大小=2 但存在间接路径，无向图中大小=2 的连通分量只有一条边，无长度 ＞1 的路径）为 1。

2.4 代码:

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

```
// 无向图类
class Graph {
private:
    int n; // 顶点数（顶点编号从 1 开始）
    vector<vector<bool>> adj; // 邻接矩阵：adj[i][j] = true 表示
存在边(i,j)
    unordered_map<int, int> component; // 顶点 -> 连通分量编号

    // 深度优先搜索标记连通分量（邻接矩阵版，O(n²)）
    void dfs(int u, int comp_id) {
        component[u] = comp_id;
        for (int v = 1; v <= n; ++v) {
            if (adj[u][v] && component.find(v) ==
component.end()) {
                dfs(v, comp_id);
            }
        }
    }

    // 标记所有连通分量
    void markComponents() {
        component.clear();
        int comp_id = 0;
        for (int u = 1; u <= n; ++u) {
            if (component.find(u) == component.end()) {
                dfs(u, comp_id++);
            }
        }
    }

    // 统计每个连通分量的顶点数
    unordered_map<int, int> countComponentSize() {
        unordered_map<int, int> size_map;
        for (auto& p : component) {
            size_map[p.second]++;
        }
        return size_map;
    }

public:
    // 构造函数：初始化顶点数（顶点编号 1~n）
```

```
    Graph(int num_vertices) : n(num_vertices) {
        adj.resize(n + 1, vector<bool>(n + 1, false)); // 1-
based 索引
    }

    // 添加无向边
    void addEdge(int u, int v) {
        if (u >= 1 && u <= n && v >= 1 && v <= n) {
            adj[u][v] = true;
            adj[v][u] = true;
        }
    }

    // 计算传递闭包（核心方法）
    vector<vector<int>> undirectedTC() {
        // 初始化传递闭包矩阵（全 0）
        vector<vector<int>> tc(n + 1, vector<int>(n + 1, 0));

        // 步骤 1：标记连通分量
        markComponents();

        // 步骤 2：统计每个连通分量的大小
        auto comp_size = countComponentSize();

        // 步骤 3：遍历所有顶点对，计算 tc[i][j]
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (i == j) {
                    tc[i][j] = 0; // 对角线强制为 0
                    continue;
                }

                // 不在同一连通分量：tc=0
                if (component[i] != component[j]) {
                    tc[i][j] = 0;
                    continue;
                }

                int c = component[i];
                int size = comp_size[c];
```

```
                // 情况 1：连通分量大小 > 2 → 存在长度>1 的路径，
tc=1
                if (size > 2) {
                    tc[i][j] = 1;
                }
                // 情况 2：连通分量大小 = 2 → 只有一条边，无长度>1 的
路径，tc=0
                else if (size == 2) {
                    tc[i][j] = 0;
                }
                // 情况 3：连通分量大小 = 1 → 无路径，tc=0
                else {
                    tc[i][j] = 0;
                }

                // 特殊修正：若连通分量大小=2 但存在直接边+间接边（如
三角形的子图）
                // 补充判断：i 和 j 是否有长度>1 的路径（即除了直接边
外，是否有其他路径）
                // 例如用例 5：连通分量大小=3，且任意两点间既有直接边
又有间接边，tc=1
                // 验证：i 和 j 之间是否存在除直接边外的其他路径
                if (size == 2) {
                    // 大小为 2 的连通分量只有两个顶点，无其他路径，
无需修正
                } else {
                    // 确保无向图的对称性
                    tc[j][i] = tc[i][j];
                }
            }
        }

        return tc;
    }

    // 打印传递闭包矩阵（仅打印 1~n 行/列）
    static void printTC(const vector<vector<int>>& tc) {
        int n = tc.size() - 1;
        for (int i = 1; i <= n; ++i) {
            cout << "i=" << i << ": (";
            for (int j = 1; j <= n; ++j) {
                cout << tc[i][j];
```

```
                if (j < n) cout << ",";
            }
            cout << ")" << endl;
        }
        cout << "-------------------------" << endl;
    }
};

// 测试用例 1：空图
void testCase1() {
    cout << "测试用例 1：空图 V={1,2,3}, E=∅" << endl;
    Graph g(3);
    auto tc = g.undirectedTC();
    Graph::printTC(tc);
}

// 测试用例 2：两个顶点一条边
void testCase2() {
    cout << "测试用例 2：V={1,2}, E={(1,2)}" << endl;
    Graph g(2);
    g.addEdge(1, 2);
    auto tc = g.undirectedTC();
    Graph::printTC(tc);
}

// 测试用例 3：三个顶点链式边
void testCase3() {
    cout << "测试用例 3：V={1,2,3}, E={(1,2),(2,3)}" << endl;
    Graph g(3);
    g.addEdge(1, 2);
    g.addEdge(2, 3);
    auto tc = g.undirectedTC();
    Graph::printTC(tc);
}

// 测试用例 4：四个顶点，1 连接 2、3、4
void testCase4() {
    cout << "测试用例 4：V={1,2,3,4}, E={(1,2),(1,3),(1,4)}" <<
endl;
    Graph g(4);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
```

```
    g.addEdge(1, 4);
    auto tc = g.undirectedTC();
    Graph::printTC(tc);
}

// 测试用例 5：三个顶点构成三角形
void testCase5() {
    cout << "测试用例 5：V={1,2,3}, E={(1,2),(2,3),(1,3)}" <<
endl;
    Graph g(3);
    g.addEdge(1, 2);
    g.addEdge(2, 3);
    g.addEdge(1, 3);
    auto tc = g.undirectedTC();
    Graph::printTC(tc);
}

int main() {
    testCase1();
    testCase2();
    testCase3();
    testCase4();
    testCase5();
    return 0;
}
```

2.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw10\hw10.2> ./main.exe
测试用例1: 空图 V={1,2,3}, E=?
i=1: (0,0,0)
i=2: (0,0,0)
i=3: (0,0,0)
-------------------------
测试用例2: V={1,2}, E={(1,2)}
i=1: (0,0)
i=2: (0,0)
-------------------------
测试用例3: V={1,2,3}, E={(1,2),(2,3)}
i=1: (0,1,1)
i=2: (1,0,1)
i=3: (1,1,0)
-------------------------
测试用例4: V={1,2,3,4}, E={(1,2),(1,3),(1,4)}
i=1: (0,1,1,1)
i=2: (1,0,1,1)
i=3: (1,1,0,1)
i=4: (1,1,1,0)
-------------------------
测试用例5: V={1,2,3}, E={(1,2),(2,3),(1,3)}
i=1: (0,1,1)
i=2: (1,0,1)
i=3: (1,1,0)
-------------------------
PS C:\Users\Macro\Desktop\hw10\hw10.2>
```