实验四 图

崔士强 PB22151743

1 问题描述

1. 输入一个无向图，输出图的深度优先搜索遍历顺序与广度优先搜索遍历顺序。
2. 输入一个无向铁通讯网图，用 Prim 和 Kruskal 算法计算最小生成树并输出。
3. 输入一个无向铁路交通图、始发站和终点站，用 Dijkstra 算法计算从始发站到终点站的最短路径。

2 算法描述

在三个程序中均使用一个类`Graph`进行操作，其定义分别如下：

2.0.1 图的遍历

```
class Graph
{
    public:
        int VertexNum;
        int EdgeNum;
        int **AdjMatrix; // adjacency matrix
        int *visited; // used for mark visited vertices
        Status InitVisitedArray();
        Status InitAdjMatrix();
        Status InitGraph();
        Status DFS(int start, Stack &S); // recursively travers the graph using a stack
        void DFSCheck(int obj, Stack &S);
        void BFSCheck(int obj, Queue &Q);
        Status BFS(int start, Queue &Q); // recursively travers the graph using a queue
        bool NoAdjVex(int vertex);
        int Search(int vertex);
};
```

2.0.2 最小代价生成树

```
class Graph
{
```

```
    public:
        int VertexNum;
        int EdgeNum;
        int **AdjMatrix;
        int *visited;
        int **ConnecMatrix;
        Status InitVisitedArray();
        Status InitAdjMatrix();
        Status InitGraph();
        int Prim();
        int FindMinimum();
        int Kruskal();
        void FindLowestCost(int &vertex1, int &vertex2);
        void Connect(int vertex1, int vertex2);
        bool AllConnected();
        bool AllVisited();
        int Search(int vertex);
};
```

2.0.3 最短路径

```
class Graph
{
    public:
        int VertexNum;
        int EdgeNum;
        int **AdjMatrix;
        int *visited;
        int *distance;
        Status InitVisitedArray();
        Status InitAdjMatrix();
        Status InitGraph();
        int ShortestPath(int start, int end);
        Status InitDistance(int start);
        int NearestVertex();
        Status Relax(int vertex);
        bool AllVisited();
};
```

3 调试分析

3.1 测试数据

3.1.1 图的遍历

第一行是两个数 n，$m(1 < n < 30\ 1 < m < 300)$，分别表示顶点数量和边的数量接下来的 m 行每行输入两个数 a、b；表示顶点 a 与顶点 b 之间有边相连，顶点编号从 1 到 n 最后输入一个数 s 表示遍历的起始顶点编号

3.1.2 最小代价生成树

第一行是两个数 n，$m(1 < n < 10000\ 1 < m < 100000)$，分别表示顶点数量和边的数量。接下来的 m 行每行输入三个数 a、b、w；表示顶点 a 与顶点 b 之间有代价为 w 的边相连，顶点编号从 1 到 n。

3.1.3 最短路径

第一行是两个数 n，$m(1 < n < 100000\ 1 < m < 1000000)$，分别表示顶点数量和边的数量接下来的 m 行每行输入三个数 a、b、w；表示顶点 a 与顶点 b 之间有长度为 w 的边相连，顶点编号从 1 到 n 最后输入两个数 s，t 表示遍历的起始顶点编号和终点编号

3.1.4 问题及解决方法

1. 在实验过程中，Kruskal 算法无法给出正确答案，经 debug 发现，原因在于判断两节点是否处于同一连通分量的算法出现错误。程序中利用一个二维数组进行标记，每次连接一条边后应当更新矩阵。不仅需要更新被连接的两个点，还需要更新这两个点所在连通分量上所有点.

2. `DFS`和`BFS`在本程序的实现中具有较高的时间复杂度，主要由于搜索邻接顶点的方式导致每次递归或循环调用都有较高的时间成本。优化这一点可以通过使用邻接表代替邻接矩阵来实现，这样可以将搜索邻接顶点的操作从 $O(N_{vertex})$ 降低到 $O(N_{edge})$，从而使`DFS`和`BFS`的时间复杂度接近于其理论最低复杂度 $O(N_{vertex} + N_{edge})$。

4 算法的时空分析

4.1 图的遍历

1. 初始化邻接矩阵（`InitAdjMatrix`）：这个函数创建一个二维数组来存储图的邻接矩阵。由于它包含两层循环，每个循环遍历 N_{vertex} 次，因此其时间复杂度为 $O(N_{vertex}^2)$。

2. 初始化访问数组（`InitVisitedArray`）：这个函数初始化一个标记顶点是否被访问过的数组。这是一个一层循环，时间复杂度为 $O(N_{vertex})$。

3. 初始化图（`InitGraph`）：这个函数读取顶点和边的数量，初始化邻接矩阵和访问数组。它的时间复杂度由初始化邻接矩阵决定，为 $O(N_{vertex}^2)$。

4. 深度优先搜索（`DFS` 和 `DFSCheck`）：`DFS` 的时间复杂度通常为 $O(N_{vertex} + N_{edge})$，因为每个顶点和边都会被访问一次。但由于本程序的实现在搜索邻接顶点时对所有顶点进行了循环，这导致每次递归调用都有 $O(N_{vertex})$ 的时间复杂度，因此总的时间复杂度可能接近于 $O(N_{vertex}^2)$。

5. 广度优先搜索（`BFS` 和 `BFSCheck`）：`BFS` 的时间复杂度与 `DFS` 相似，通常为 $O(N_{vertex}+N_{edge})$。然而，同样由于邻接顶点搜索的实现方式，总的时间复杂度可能也接近于 $O(N_{vertex}^2)$。

6. 搜索未访问的邻接顶点（`Search`）：这个函数对一个顶点的所有邻接顶点进行搜索，时间复杂度为 $O(N_{vertex})$。

4.2 最小代价生成树

1. 初始化邻接矩阵（`InitAdjMatrix`）：这个函数创建并初始化一个二维数组作为邻接矩阵，其时间复杂度为 $O(N_{vertex}^2)$。

2. 初始化图（`InitGraph`）：这个函数读取顶点和边的数据，初始化邻接矩阵和访问数组。整体时间复杂度由邻接矩阵的初始化决定，为 $O(N_{vertex}^2)$。

3. Prim 算法：Prim 算法的时间复杂度取决于如何找到每个顶点的最小权重边。在此实现中，对于每个未访问的顶点，都遍历所有边来找到最小权重边，因此时间复杂度为 $O(N_{vertex}^2)$。

4. Kruskal 算法：Kruskal 算法的时间复杂度取决于边的排序和连接组件的检查。在此实现中，每次都从未连接的边中找到最小权重边，这需要遍历所有边。在最坏情况下，这会导致 $O(N_{vertex}^2)$ 的时间复杂度。

5. `AllConnected` 和 `AllVisited` 方法：这两个方法都有 $O(N_{vertex})$ 的时间复杂度。

6. 连接组件（`Connect`）：在 Kruskal 算法中，每次添加边时都需要更新连接组件。在最坏情况下，这可能需要 $O(N_{vertex}^2)$ 的时间。

Prim 和 Kruskal 算法的时间复杂度在这个实现中都是 $O(N_{vertex}^2)$。这对于较小的图是可行的，但对于大型图可能不够高效。通常，Prim 算法可以通过使用优先队列（如最小堆）来降低时间复杂度到 $O((N_{vertex} + N_{edge}) \log N_{vertex})$，而 Kruskal 算法可以通过对边进行排序（如快速排序或归并排序）以及使用并查集来降低时间复杂度至 $O(N_{edge} \log N_{edge})$

4.3 最短路径

1. 初始化邻接矩阵（`InitAdjMatrix`）：这个函数创建并初始化一个二维数组作为邻接矩阵，其时间复杂度为 $O(N_{vertex}^2)$，其中 N_{vertex} 是顶点的数量。

2. 初始化图（`InitGraph`）：这个函数读取顶点和边的数据，初始化邻接矩阵和访问数组。整体时间复杂度由邻接矩阵的初始化决定，为 $O(N_{vertex}^2)$。

3. 计算最短路径（`hortestPath`）：这个函数使用迪杰斯特拉算法计算从起始点到终点的最短路径。它在每次迭代中找到未访问顶点中距离最短的顶点，然后更新所有邻接顶点的距离。算法的总体时间复杂度为 $O(N_{vertex}^2)$，因为它涉及到对每个顶点的搜索和松弛（Relax）操作。

4. 初始化距离数组（`InitDistance`）：这个函数初始化距离数组，时间复杂度为 $O(N_{vertex})$。
5. 寻找最近顶点（`NearestVertex`）：这个函数遍历所有顶点以找到未访问的距离最短的顶点，时间复杂度为 $O(N_{vertex})$。
6. 更新操作（`Relax`）：这个函数更新与给定顶点相邻的所有未访问顶点的距离，时间复杂度为 $O(N_{vertex})$。
7. 检查所有顶点是否访问过（`AllVisited`）：这个函数检查是否所有顶点都已访问，时间复杂度为 $O(N_{vertex})$。

5 测试结果及分析

5.1 图的遍历

输入：

```
5 6
1 5
1 4
3 1
2 1
2 5
5 3
1
```

输出：

```
1 2 5 3 4
1 2 3 4 5
```

5.2 最小代价生成树

输入：

```
4 5
1 2 2
1 3 2
1 4 3
2 3 4
3 4 3
```

输出：

```
7
```

5.3 最短路径

输入:

```
4 5
1 2 10
1 3 20
2 3 15
2 4 30
3 4 20
1 4
```

输出:

```
40
```

6 实验体会和收获

通过本次实验，掌握了图的存储结构，熟悉了遍历、求最小代价生成树、求最短路径的算法