实验报告 #6

开发环境

- 编译器：gcc v6.3.0
- 编辑器：VS Code
- 平台：X86-64
- OS：win10 home

程序使用结果截图

实验样例**1**

```
enter data
Number of point: 4
Number of path: 3
Two point in one line to show a path(begin with 1)
1 2
1 3
2 4
[M]atrix graph
[L]ist graph
[B]FS
[D]FS
[S]hortest route
[E]xit
M
Initializing a Matrix Graph...
B
BFS:
1 2 3 4
D
BFS:
1 2 4 3
S
Shortest route from 1 to other:
0 1 1 2
L
Initializing a List Graph...
B
BFS:
1 2 3 4
D
BFS:
1 2 4 3
S
Shortest route from 1 to other:
0 1 1 2
```

实验样例2

```
enter data
Number of point: 6
Number of path: 9
Two point in one line to show a path(begin with 1)
1 2
1 3
2 3
2 4
3 5
4 3
4 5
4 6
5 6
[M]atrix graph
[L]ist graph
[B]FS
[D]FS
[S]hortest route
[E]xit
M
Initializing a Matrix Graph...
B
BFS:
1 2 3 4 5 6
D
DFS:
1 2 3 4 5 6
S
Shortest route from 1 to other:
0 1 1 2 2 3
L
Initializing a List Graph...
B
BFS:
1 2 3 4 5 6
D
DFS:
1 2 3 4 5 6
S
Shortest route from 1 to other:
0 1 1 2 2 3
```

程序说明

程序逻辑

1. 进入程序，输入节点的数量和边的数量。

   > 此程序的节点从1开始，最大值为1000。边为无向边。

2. 以每行两个数字，以空格分隔的形式，输入边。
3. 进入功能选单，输入 `M/L` 以选择邻接矩阵或邻接表版本的图。

> 如果不选择版本，则默认为邻接矩阵。

4. 输入 `B/D/S` 来选择执行广度优先搜索、深度优先搜索以及最短路径算法。

> 此时亦可输入 `M/L` 切换图的版本（虽然说输出不会有什么不同就是了）。

> 最短路径算法计算的是从 1 节点到其他节点的最短路径。

数据结构

```
class Mgraph    //邻接数组
{
  public:
    int _point; //记录节点的数量，下同
    int _path;  //记录边的数量，下同
    bool path[1001][1001];  //记录各个边
    bool visited[1001];     //记录是否访问过某节点，下同
};

class Lgraph    //邻接表
{
  public:
    int _point;
    int _path;
    list<int> neighbors[1001];  //记录各个边
    bool visited[1001];
};
```

算法设计

> 以邻接数组为例。以便阅读，跳过了部分与算法无关的代码。

```
Mgraph()    //图的构建
{
        _point = stoi(str_point);   //使用stoi()函数确保输入数据合法性。
        _path = stoi(str_path);
        int a, b;
        for (int i = 0; i < _path; ++i) //循环读入无向边
        {
            cin >> a >> b;
            if (a <= 0 || b <= 0 || a > _point || b > _point)   //判断边的合法性
                path[a][b] = path[b][a] = true; //构建无向边
        }
}
```

```
void bfs()  //广度优先搜索
{
        init(visited);  //初始化【表示已走过的节点的】 visited 数组，下同
        q.push(1);      //将头节点 1 放入队中
        while (!q.empty())
        {
            int temp = q.front();
            visit(temp);            //访问当前节点，下同
            for (int i = 1; i <= _point; ++i)   //对于每个节点
                if (path[temp][i] == true && visited[i] != true)    //如果有【存在路径】且【未访问
的】下一节点
                {
                    visited[i] = true;  //将全部下一层节点标记为已访问
                    q.push(i);      //将当前节点的全部下一层节点放入队中
                }
            q.pop();                //弹出当前节点，进入下一个等候访问的节点
        }
}
```

```
void dfs()  //深度优先搜索
{
        init(visited);
        dfs_do(1);  //调用辅助函数，传入头节点 1
}
void dfs_do(int temp)   //深度优先搜索的辅助函数
{
        visit(temp);
        for (int i = 1; i <= _point; ++i)
            if (path[temp][i] == true && visited[i] != true)
            {
                visited[i] = true;
                dfs_do(i);          //递归调用辅助函数，搜索/访问下一层节点
            }
}
```

```
 void sr()      //最短路径算法
{
        init(dis,65535);  //初始化记录路径长度的 dis 数组为一个极大的数值

        for (int i = 2; i <= _point; ++i)   //将与节点 1 存在直接路径的全部节点路径长度标1
            if (path[1][i] == true)
                dis[i] = 1;
        //松弛
        for (int i = 1; i <= _point; ++i)   //遍历全部可能存在的路径
            for (int j = 1; j <= _point; ++j)
                if (path[i][j] == true)     //对于某条路径，如果存在一条替代路径
                    if (dis[i] > dis[j] + path[i][j])   //并且，如果借由这条替代路径后，新的路径长度小于原路径
                        dis[i] = dis[j] + path[i][j];   //改用借由替代路径

        print(dis); //输出结果
}
```

info

16340247 席睿 软件工程教务三班