算法基础 实验8 图搜索BFS算法及存储优化

PB21081601 张芷苒

实验目标

针对图，根据给定的数据选择合适的存储方式（邻接矩阵和邻接表中的一种）进行存储（存储方式选择也是实验的检查内容之一），并进行图的广度优先遍历的过程。

算法设计思路

图存储方式

本次实验选择两种图存储方式：邻接矩阵和邻接表。对于图 $G(V,E)$ 来说，其邻接表由一个包含 $|V|$ 条链表的数组 Adj 所构成，每个节点有一条链表。对于每个节点 $u \in V$,邻接表 $Adj[u]$ 包含所有与结点u之间有边相连的结点 v 。对于邻接矩阵表示来说，通常将图 G 中的结点编为 $1,2,\cdots,|V|$ 。图 G 由一个 $|V| \times |V|$ 的矩阵 $A=(a_{ij})$ 予以表示，矩阵满足以下条件：

$$a_{ij}=\begin{cases} nbsp; & nbsp;1 & 若(i,j)\in E \\ nbsp; & nbsp;0 & 其他 \end{cases}$$

空间复杂度：

存储方式	空间复杂度
邻接表	$\Theta(V+E)$
邻接矩阵	$\Theta(V^2)$

由空间复杂度可见，如果结点数较少且边比较密集，那么使用邻接矩阵存储图优于邻接表；如果结点数较多且边比较稀疏，那么使用邻接表存储图优于邻接矩阵。

BFS广度优先遍历

广度优先搜索首先将所有结点赋为白色，然后维护一个队列，首先将源节点加入队列，然后每次出队一个节点，将节点赋为灰色，然后将所有与该节点相连的节点入队，之后将当前节点赋为黑色。重复上述步骤，直至队空，遍历完成。

实现过程

data

1. 从 'A' 开始，加入队列：[A]。
2. 访问 'A'，加入 'B' 和 'C'：[B, C]。
3. 访问 'B'，加入 'D', 'E', 'G'：[C, D, E, G]。

4. 访问 'C'，加入 'H', 'F'：[D, E, G, H, F]。
5. 访问 'D'：[E, G, H, F]。
6. 访问 'E'：[G, H, F]。
7. 访问 'G'：[H, F]。
8. 访问 'H'：[F]。
9. 最后访问 'F'：[]。
10. 遍历结束。

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

#define MaxVEX 100

enum Color { White, Black, Grey };

struct MGraph {
    Color vex[MaxVEX];
    int edge[MaxVEX][MaxVEX];
    int numVertexes, numEdges;
    map<char, int> vertexMap;  // 映射顶点字符到索引
} G;

void BFS(int v) {
    queue<int> q;
    q.push(v);
    G.vex[v] = Grey;
    while (!q.empty()) {
        int temp = q.front();
        q.pop();
        cout << (char)(temp + 'A') << " ";
        for (int i = 0; i < G.numVertexes; i++) {
            if ((G.edge[temp][i] == 1) && (G.vex[i] == White)) {
                q.push(i);
                G.vex[i] = Grey;
            }
        }
        G.vex[temp] = Black;
    }
    cout << endl;
}

int main() {
```

```
    ifstream infile("data.txt");
    string verticesLine, edgeLine;
    getline(infile, verticesLine);

    stringstream ss(verticesLine);
    string vertex;
    vector<char> vertexList;

    while (getline(ss, vertex, ',')) {
        vertexList.push_back(vertex[0]);
    }

    G.numVertexes = vertexList.size();
    for (char v : vertexList) {
        G.vertexMap[v] = v - 'A';
        G.vex[v - 'A'] = White;
        for (int j = 0; j < G.numVertexes; j++) {
            G.edge[v - 'A'][j] = 0;
        }
    }

    while (getline(infile, edgeLine)) {
        char a = edgeLine[0], b = edgeLine[2];
        int u = G.vertexMap[a], v = G.vertexMap[b];
        G.edge[u][v] = G.edge[v][u] = 1;
    }

    // 初始化所有顶点为White
    for (int i = 0; i < G.numVertexes; i++) {
        G.vex[i] = White;
    }

    BFS(G.vertexMap['A']);
    return 0;
}
```

twitter_small

主要思路是利用图的邻接表表示来实现广度优先搜索（BFS），并从特定顶点开始遍历图。程序主要分为几个部分：

1. **图的表示**：
   - 使用邻接表表示图。图中每个顶点存储在一个顶点数组中，每个顶点包含其数据（`data`）、指向其第一个邻接点的指针（`firstEdge`）以及用于 BFS 的颜色标记（`color`）。
   - 每个顶点的邻接点通过边表节点（`EdgeNode`）链接，这些节点形成一个链表。
2. **从文件读取图数据**：

- 读取顶点信息，并创建顶点名称到索引的映射（使用 `map<string, int>`）。
- 读取边信息，构建每个顶点的邻接链表。

3. **广度优先搜索（BFS）实现**：
   - 从指定顶点开始，使用队列实现 BFS。
   - 遍历过程中，将当前顶点的邻接顶点（尚未访问的）加入队列，并更新顶点的颜色状态（`White` 未访问，`Grey` 正在访问，`Black` 已访问）。
   - 在 BFS 过程中，统计访问过的顶点数量，并将访问顺序输出到文件。
4. **主函数**：
   - 打开并读取图数据文件（`twitter_small.txt`），构建图。
   - 输入要开始 BFS 的顶点名称，并调用 BFS 函数。
   - 计时 BFS 的执行时间，并输出统计结果。

该程序的核心是用 BFS 遍历图，并对遍历过程进行计时和统计。邻接表表示法使得程序可以高效地处理稀疏图。

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

// 颜色枚举，用于 BFS 遍历中标记顶点状态
enum Color {
    White,  // 未访问
    Black,  // 已访问
    Grey    // 正在访问
};

typedef int EdgeType;  // 边的权重类型（此处未使用）
typedef string VertexType;  // 顶点类型

#define MaxVEX 1000000  // 定义最大顶点数

// 边表节点
typedef struct EdgeNode {
    int adjVex;  // 邻接点索引
    EdgeNode *next;  // 指向下一个邻接点的指针
} EdgeNode;

// 顶点表节点
typedef struct VertexNode {
    VertexType data;  // 顶点信息
    EdgeNode *firstEdge;  // 指向第一个邻接点的指针
    Color color;  // BFS中顶点的颜色状态
} VertexNode, AdjList[MaxVEX];

// 邻接表图
typedef struct GraphAdjList {
    AdjList adjList;  // 顶点数组
```

```
    int numVertexes, numEdges;  // 顶点数和边数
} GraphAdjList;

GraphAdjList G;  // 创建一个图的实例

// BFS函数，从顶点v开始进行广度优先搜索
void BFS(int v, ofstream &outfile) {
    int count = 0;  // 遍历到的顶点数
    if (G.adjList[v].color != White) {
        return;  // 如果顶点已经被访问过，则直接返回
    }
    queue<int> q;
    q.push(v);
    G.adjList[v].color = Grey;  // 标记为正在访问

    while (!q.empty()) {
        int temp = q.front();
        q.pop();
        outfile << G.adjList[temp].data << endl;  // 输出顶点信息到文件
        count++;  // 计数增加
        EdgeNode *temp1 = G.adjList[temp].firstEdge;
        while (temp1) {
            if (G.adjList[temp1->adjVex].color == White) {
                q.push(temp1->adjVex);
                G.adjList[temp1->adjVex].color = Grey;
            }
            temp1 = temp1->next;
        }
        G.adjList[temp].color = Black;  // 标记为已访问
    }
    cout << "BFS遍历到的顶点数为：" << count << endl;
}

// 程序主入口
int main() {
    int numEdges = 0, numVertexes = 0;
    string p;
    set<string> s;
    map<string, int> m;  // 用于存储顶点名称到索引的映射
    ifstream
infile("D:\\code\\cc\\single\\23algorithms\\lab8\\v1\\twitter_small.txt",
ios_base::in);

    // 从文件中读取顶点数据
    while (infile >> p) {
        s.insert(p);  // 插入到集合中，自动去重
        numEdges++;
    }
    infile.close();
```

```
    infile.clear(ios::goodbit); // 恢复流状态
    numEdges /= 2;
    numVertexes = s.size();  // 计算顶点数
    G.numEdges = numEdges;
    G.numVertexes = numVertexes;

    // 构建顶点表
    int i = 0;
    for (auto it = s.begin(); it != s.end(); it++, i++) {
        G.adjList[i].data = *it;
        m.insert(pair<string, int>(*it, i));  // 建立映射
        G.adjList[i].firstEdge = NULL;
        G.adjList[i].
```

实验结果

对于数据集data.txt:

从A开始的BFS遍历:

```
PS D:\code\cc\single\23algorithms>  &
debugAdapters\bin\WindowsDebugLauncher
MIEngine-Out-avyp1qho.1ed' '--stderr=M
mfmis.f0f' '--dbgExe=D:\settings\mingw
A B C D E G F H I
```

对于数据集twitter_small.txt:

```
请输入编号:
17116707
BFS遍历到的顶点数为: 81304
耗时 1.796s
PS D:\code\cc\single\23algorithms>
```

遍历的过程会输出到small.txt：