Experiment—BFS

PB18010496 杨乐园

Introduction

针对无向图，通过邻接多重表方式进行存储。以节点A为起始节点，输出图的广度优先遍历的过程。

Purpose

实验目的：学习邻接多重表的数据结构以及存储方式，掌握广度优先遍历算法。

Data structure

顶点的数据结构为：

$$
\begin{aligned}
EVerNode = \{ \\
&data : char; \\
&number : int; \\
&vi : int; \\
&firstEdge : edgeNode*; \\
\}
\end{aligned}
$$

其中$data$记录节点名称，$number$记录节点编号，vi记录是否被访问。

边的数据结构为：

$$
\begin{aligned}
edgeNode = \{ \\
&ivex : int; \\
&jvex : int; \\
&ilink : ENode*; \\
&jlink : ENode*; \\
\} * linkNode
\end{aligned}
$$

其中，$ivex$和$jvex$表示该边的两个顶点在顶点数组中的位置；$ilink$和$jlink$分别表示指向依附于顶点$ivex$和$jvex$下一条边的指针。

Idea

广度优先遍历BFS算法是较为经典的一个算法，通过队列的“先进先出”原则，将起始节点入队，后，对其访问后将其出兑，并将其邻接节点依次入队；新一轮在出队一个节点，访问，将其不在队列里的邻接节点入队，依次步骤下去，直到这一连通分量结束，即队列为空就，完成遍历。若图为不连通的，只需选取新的起始节点开始遍历即可。

Algorithm

首先我们先利用每次插入一条边的方法，构造好相应的图：

```
#define vexnumber  8
char symbol[vexnumber] = { 'A','B','C','D','E','F','G','H' };
int first[] = { 0,0,2,2,3,5 };
int second[] = { 1,2,3,4,4,6 };
```

```
//逐边插入；
void insertEdgeAction(AMLGraph& G, int index1, int index2) {
    edgeNode* p, * q;
    edgeNode* edge = new edgeNode[1];

    edge->iVex = index1;
    edge->jVex = index2;

    p = G.adjmultiList[index1].firstEdge;//相当于链表的插入
    if (!p)
    {
        G.adjmultiList[index1].firstEdge = edge;
        edge->iLink = NULL;
    }
    else
    {
        G.adjmultiList[index1].firstEdge = edge;
        edge->iLink = p;
    }

    q = G.adjmultiList[index2].firstEdge;//相当于链表的插入
    if (!q)
    {
        G.adjmultiList[index2].firstEdge = edge;
        edge->jLink = NULL;
    }
    else
    {
        G.adjmultiList[index2].firstEdge = edge;
        edge->jLink = q;
    }
}

//建图；
void createAMLGraph(AMLGraph& G, int vexNum) {
    G.vexNum = vexNum;
    G.edgeNum = 0;
    for (int i = 0; i < G.vexNum; i++) {
        G.adjmultiList[i].number = i;
        G.adjmultiList[i].data = symbol[i];
        G.adjmultiList[i].vi = 0;
        G.adjmultiList[i].firstEdge = NULL;
    }
    for (int j = 0; j < size(first); j++)
    {
        if (first[j] < 0 || second[j] < 0)
            exit(-1);
        insertEdgeAction(G, first[j], second[j]);
    }
}
```

与此同时，为了输出图，我们给出邻接多重表的结构输出：

```
//邻接多重表输出；
void printAMLGraph(AMLGraph& G) {
    for (int i = 0; i < G.vexNum; i++) {
```

```
        cout << i << " " << G.adjmultiList[i].data;
        edgeNode* edge = G.adjmultiList[i].firstEdge;

        while (edge) {
            cout << "-->|" << edge->iVex << "|" << edge->jVex << "|";
            if (edge->iVex == i) {
                edge = edge->iLink;
            }
            else {
                edge = edge->jLink;
            }
        }
        cout << "-->NULL" << endl;
    }
}
```

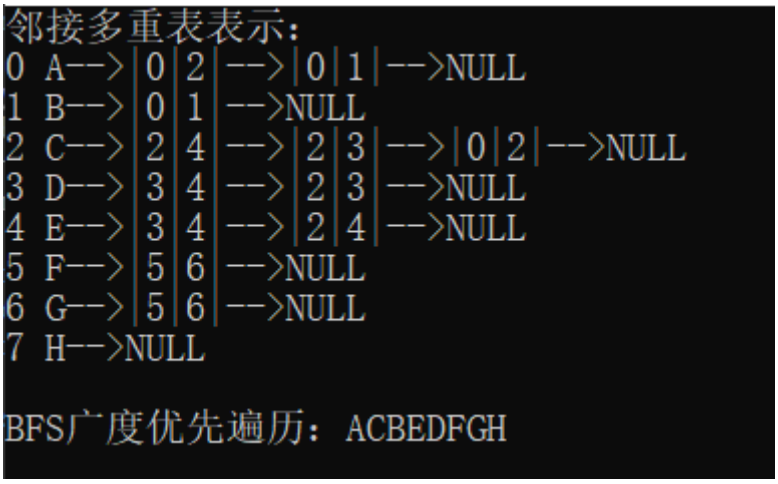
最后我们根据上述想法给出BFS遍历的算法：

```
//BFS;
void BFS(AMLGraph& G, int start)
{
    queue<vertexNode> q;
    q.push(G.adjmultiList[start]);
    while (q.size() > 0)
    {
        vertexNode u = q.front();
        q.pop();
        //访问节点;
        cout << u.data;
        G.adjmultiList[u.number].vi = 1;

        edgeNode* p = u.firstEdge;
        while (p != NULL)
            if (u.data == G.adjmultiList[p->iVex].data && G.adjmultiList[p-
>jVex].vi == 0)
            {
                q.push(G.adjmultiList[p->jVex]);
                p = p->iLink;
            }
            else
                p = NULL;
    }
}
```

Results

通过运行程序与测试数据，我们有如下输出结果：



我们可以看到，输出结果正确。

Code

具体完整代码，参看附件文件$BFSsearch$。