实验八：图搜索 BFS 算法及存储优化

学号：PB21000224　　姓名：陈鸿绪　　日期：12.7.2023

实验内容：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。 数据集 1：使用 data.txt 中的数据，看做无向图，选择合适的方式进行存储（提示：其特征为 节点数较少而边比较密集），并以 A 为起始顶点输出遍历过程。数据集 2、3 为 twitter 真实数据集，数据集规模如下： Nodes 81306, Edges 1768149, 有向图； twitter_large: 数据 3 规模如下： Nodes 11316811, Edges 85331846, 有向图。对 twitter_small, 选择一种合适的存储方式存储数据，并输出 BFS 的遍历时间。对于 twitter_large，实验中并不做要求。

实验目的：掌握针对不同的图的特点选择适合的存储结构, 同时学会图广度搜索。

算法设计思路：

1. 考虑 twitter_small 的数据点较多边相对较少，所以对于 twitter_small 设计 read_data 函数，读取文件中的边信息，并以有向图的形式邻接表存储，由于 twitter_small 中图的结点索引并不是连续的，所以需要构造一个 map，让不连续的结点索引映射到连续的索引上，此时就可用一个连续的数组形式存储结点信息。
2. 构造一个队列，从第一个点开始进行广度搜索，直到队列为空，标记已访问结点，再进行第二个点的广度搜素，标记已访问结点，以此类推直至所有点

都已访问。

3. 计算该部分所需时间，同时输出访问结点数目，对比题中所给数目检查是否正确。
4. 考虑 data 的数据点较少边相对较多，所以采取无向图的邻接矩阵存储结构，设计 read_data_1 函数读取邻接矩阵。和 twitter_small 一致也需要进行映射到连续索引的操作。
5. 与 twitter_small 一致的利用队列的广度搜素操作，直至所有点均已访问。
6. 计算该部分所需时间，同时输出访问结点数目，对比题中所给数目检查是否正确。

主要代码以及注释：

```
struct arc{
	struct arc* next;
	int next_node;
};
struct node{
	struct arc *first_arc;
};
typedef struct arc arc;
typedef struct node node;
map<int,int> dict;
node nodes[N];
int flag[N]={0};
//以上是建立邻接表的基本工作
void read_data(){
	FILE *fp=fopen("twitter_small.txt","r+"); //读取文件
	int len=0;
	while(!feof(fp)){
		int s,d;
		fscanf(fp,"%d %d\n",&s,&d);
		arc *temp_arc=(arc *)malloc(sizeof(arc));
		// 读取边信息
		auto it_1=dict.find(s);
```

```
            if(it_1==dict.end()) dict[s]=len++; //看 s 是否再字典中，如果不在则加入
            auto it_2=dict.find(d);
            if(it_2==dict.end()) dict[d]=len++; //看 d 是否再字典中，如果不在则加入
            temp_arc->next_node=dict[d];
            temp_arc->next=NULL;
            if(nodes[dict[s]].first_arc==NULL){
                nodes[dict[s]].first_arc=temp_arc;
            }
            else{
                temp_arc->next=nodes[dict[s]].first_arc;
                nodes[dict[s]].first_arc=temp_arc;
            }
        }//头插法插入新边
        cout << len <<endl;
        fclose(fp);
}//twitter_small 的邻接表建立

int matrix[N1][N1]={0};
map<char,int> dict_1;
int flag_1[N1]={0};
void read_data_1(){
        FILE *fp=fopen("data.txt","r+");
        char s[30];
        fscanf(fp,"%s\n",s);
        // cout << s;
        int len=0;
        while(!feof(fp)){
            char s,d;
            fscanf(fp,"%c-%c\n",&s,&d);
            auto it_1=dict_1.find(s);
            if(it_1==dict_1.end()) dict_1[s]=len++;
            auto it_2=dict_1.find(d);
            if(it_2==dict_1.end()) dict_1[d]=len++;
            matrix[dict_1[s]][dict_1[d]]=1;
            matrix[dict_1[d]][dict_1[s]]=1;
        }//同 read_data 建立字典
        for(int i=0;i<N1;i++){
            for(int j=0;j<N1;j++){
                printf("%d ",matrix[i][j]);
            }
            printf("\n");
        }//读入邻接矩阵
}//data 的邻接矩阵建立
```

```
void BFS_matrix(int start,int &time){
//从 start 处开始，time 是记录现已经访问的结点数
    queue<int> q;
    q.push(start);
    flag_1[start]=1;
    while(!q.empty()){
        int temp=q.front();
        time++;
        // cout << time<< ' '<<temp<< endl;
        q.pop();
        int p=0;
        while(p<N1){
            if(matrix[temp][p]==1){
                if(flag_1[p]!=1){
                    flag_1[p]=1;
                    q.push(p);
                }
            }
            p++;
        }
    }//广度搜索的队列操作
}//邻接矩阵对单一结点的 BFS

void BFS_matrix_all(int &time){
    for(int i=0;i<N1;i++){
        if(flag_1[i]==1) continue;
        else BFS_matrix(i,time);
    }
}//邻接矩阵对所有点进行 BFS 访问。

void BFS(int start,int &time){
    queue<node> q;
    q.push(nodes[start]);
    flag[start]=1;
    while(!q.empty()){
        node temp=q.front();
        q.pop();
        arc *p=temp.first_arc;
        time++;
        while(p){
            if(flag[p->next_node]==1) p=p->next;
            else{
                flag[p->next_node]=1;
                q.push(nodes[p->next_node]);
```

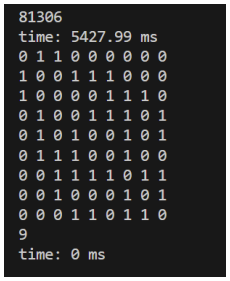
```
                }
            }
        }//广度搜索的队列操作
}//对单一结点的 BFS，用于邻接表。

void BFS_all(int &time){
        for(int i=0;i<N;i++){
            if(flag[i]==1) continue;
            else BFS(i,time);
        }
}//邻接表对所有的点进行 BFS 访问。
```

算法测试结果：

以下为测试结果：

.



可以对照输出访问结点数是否与题中所给是否相同，可以发现 81306 和 9 均是正确答案，且时长分别为 5427.99、0ms（ms 过于小，所以打印出来近似成为了 0）

实验中的困难与收获：

由于想到两个 BFS 极为相近，所以我几乎没有怎么改动邻接表的 BFS 具体算法来套用，所以在写代码的时候发现邻接矩阵的算法跑出来是一个错误的访问数，最后在 debug 的过程中发现了在邻接表的 BFS 算法代码中，不小心没有改一些变量名，导致邻接矩阵 BFS 的有些参数用的是邻接表程序的参数，最终正确改正。收获：学会了 BFS 算法，同时也掌握了如何选择适当的存储结构。