数据结构实验报告四

题目： 图及其应用

班级：计科一班　　姓名：范文　　学号： PB18111679　　完成日期：2019.12.5

一．实验介绍

实验一： DFS 的应用

1. **实验要求：**
   基于邻接矩阵的存储结构，使用非递归的深度优先搜索算法，求无向连通图中的全部关节点，并按照顶点编号升序输出。
2. **设计思路：**
   设计 low 且 low[v] = min { visited[v] , low[w] , visited[k] }。其中 w 是以 v 为根的 dfs 树中与 v 相关联的顶点，k 是与以 v 为根的子树相连的祖先结点。当 low[w] >= visited[v] 且 v 是 w 的父亲结点时，说明 v 的子树中的所有顶都不与 v 的祖先相关联，砍去 v 之后该图必定不连通，则 v 就是关节点。
3. **核心代码：**
   首先，建立初始化的函数 find_cutvex,它初始化了 visited，low，min，previous 数组。它去第一个顶为初始顶，对其进行非递归 dfs 找结点。

```
/* The non-recursive function to figure out the low array.
 low[v] = min { visited[v],low[w],visited[k] };
 w is v's son in a dfs tree and v and w is connected.
 k is the ancestor that is connected(in dotted line) with v's subtree.
 This function also initializes the visited[],low[],min[] and count.
 Suppose that the input graph is connected. */
 void find_cutVex(const AdjMatrixGraph<char> G)
{
    stack<Couple> S;
    vector<char> vexInfo = G.get_vexs();
    int vexNum = G.get_vexNum(), count = 1,cutVexNum = 0;
    // visited shows the visiting order of vertex.
    // visited[v] = 0 means v has not been visited.
    int visited[vexNum];
    int *visitedPtr = visited;
    for (int i = 0; i < vexNum;++i)
```

```
        visited[i] = 0;
    //low[v] means the earliest ancestor that the subtree of v can connect
to.
    //the initialization of low doesn't matter.
    //- because we can use DFS to figure out low in a post order way.
    int low[vexNum];
    int *lowPtr = low;
    for (int i = 0; i < vexNum;++i)
        low[i] = 0;
    //since it is a non- recursive function, we need to use an array
    //-to store the min of each vertex.
    //In the process, mins[v] is originally set as visited[v];
    int mins[vexNum];
    int *minsPtr = mins;
    for (int i = 0; i < vexNum;++i)
        mins[i] = 0;
    bool isCutVex[vexNum];
    for(int i = 0;i < vexNum;++i)
        isCutVex[i] = false;
    bool* isCutVexPtr = isCutVex;
    int previous[vexNum];
    for (int i = 0; i < vexNum;++i)
        previous[i] = -1;
    int *previousPtr = previous;
    //set the vertex 0 as the root of the generating tree.
    visited[0] = 1;
    //first consider the first adjacent vertex of vertex 0.
    int vex_v;
    vex_v = get_next_adjVex(G, 0,-1);
    dfs_non_recur(G, vex_v, S,visitedPtr, lowPtr, minsPtr
                  ,previousPtr, isCutVexPtr,count, cutVexNum);
```

```
    // vertex 0 has more than 1 subtree, firstly it is a cut vertex.
    // then keep on searching its remaining subtrees.
    if(count < vexNum)
    {
        cout << "one cut vertex has been found: " << vexInfo[0] << endl;
        isCutVexPtr[0] = true;
        ++cutVexNum;
        for (vex_v = get_next_adjVex(G, 0, vex_v); vex_v != -1;
            vex_v = get_next_adjVex(G, 0, vex_v) )
        {
            dfs_non_recur(G, vex_v, S,visitedPtr, lowPtr, minsPtr
                        ,previousPtr,isCutVexPtr, count, cutVexNum);
        }
    }
     if( cutVexNum == 0)
         cout << "there is no cut vertex in this graph.\n";
    else
        cout << "there are " << cutVexNum << " cut vertexes in this graph.\
n";
}
```

然后，建立一个 get_next_adjvex 函数，因为在我的 AdjMatrixGraph 的 ADT 中没有设计取下一顶点的函数。

```
//Getting baseVex 's next adjacent vertex after currvex
// if currVex  = -1, return the first adjacent vertex of baseVex.
// there is no adjacent vertex after the currVex , return -1;
int get_next_adjVex(const AdjMatrixGraph<char> G,int baseVex, int currVex)
{
    const double *arcsPtr = G.get_arcs();
    int vexNum = G.get_vexNum();
    if(currVex == vexNum -1)
```

```
        return -1;
    for (int vex = currVex + 1; vex < vexNum;++vex)
    {
        if( *(arcsPtr + MAX_VERTEX_NUM * baseVex + vex) == 1 )
            return vex;
    }
    return -1;
}
```

最后，建立非递归 dfs 函数来求关节点。利用一个栈，把一组顶点（v，w）压栈，弹出，经过一系列迭代，最后可以判断出一个顶是否是关节点。

```
/* After ascertaining a root, use this to build its subtree,
-figure out visited,low and min in a non recursive way.
 low[v] = min { visited[v],low[w],visited[k] };
 w is v's son in a dfs tree and v and w is connected.
 k is the ancestor that is connected(in dotted line) with v's subtree. */
void dfs_non_recur(const AdjMatrixGraph<char> G, const int & vex_v, stack<C
ouple> & S,
                int * visited,int * low ,int * mins,int *previous,
                bool *isCutVexPtr, int & count, int & cutVexNum)
{
    mins[vex_v] = ++count;
    visited[vex_v] = mins[vex_v];
    Couple cp;
    cp.vex_1 = vex_v;
    cp.vex_2 = -1;
    S.push(cp);
    while( !S.empty())
    {
        cp = S.top();
        int v = cp.vex_1, w = cp.vex_2;
        //case : w exists.
```

```
        // try to use (v,w) to change mins[v] and get low[v]
        if (w != -1)
        {
            // v is the father of w
            if (visited[v] < visited[w] && previous[w] == v)
            {
                if (low[w] < mins[v])
                    mins[v] = low[w];
                // isCutvex can avoid repeating reporting the same vertex a
s cut vertex.
                if (low[w] >= visited[v] && isCutVexPtr[v] == false)
                {
                    cout << "one cut vertex has been found: " << (G.get_vex
s())[v] << endl;
                    isCutVexPtr[v] = true;
                    ++cutVexNum;
                }
            }
            //v is the son of w
            else
            {
                if (visited[w] < mins[v])
                    mins[v] = visited[w];
            }
            } //if( c2.vex_2 != -1 )
        S.pop();
        w = get_next_adjVex(G, v, w);
        if(w != -1)
        {
            cp.vex_1 = v;
            cp.vex_2 = w;
```

```
            S.push(cp);
            if( visited[w] == 0 )
            {
                mins[w] = ++count;
                visited[w] = mins[w];
                previous[w] = v;
                cp.vex_1 = w;
                cp.vex_2 = -1;
                S.push(cp);
            }
        }
        // w == -1 that is w doesn't exist.
        //we can now get the low[v]
        else
        {
            low[v] = mins[v];
        }
    }//while( !S.empty())
}
```

实验二：bfs 的应用

1. **实验要求：**
   基于邻接表的存储结构，依次输出从顶点 0 到顶点 1、2、……、n-1 的最短路径和各路径中的顶点信息。

2. **设计思路：**
   利用队列，将图中的顶加入队列并弹出。假设 u 被弹出后，其子节点 u 和 w 被加入队列，则 orig[v] = orig[w] = u。当 v 被弹出的时候，说明 v 被找到，通过 orig 向前找直至找到初始结点，因此通往 v 的最短路径被找到。

3. **核心代码**:
   Bfs 的核心代码如下，visited 记录了结点被访问的顺序，shtpath 记录了初始结点到达每个顶点的最短路径，orig 记录了在广度优先搜索中一个顶点的前继结点。

```cpp
//bfs function for traversing a graph
void bfs_traverse_adjList(const AdjListGraph<char> &G)
{
    vector<VexNode<char>> G_adjList = G.get_adjList();
    int vexNum = G.get_vexNum();
    //orig[w] = v means that v is w's father.
    int orig[vexNum];
    for(int i = 0; i < vexNum;++i)
        orig[i] = -1;
    bool visited[vexNum];
    for(int i = 0;i < vexNum;++i)
        visited[i] = false;
    // record the shortest path of every vex.
    vector< stack<char> > shtPath;
    stack<char> onePath;
    for(int i = 0;i < vexNum;++i)
        shtPath.push_back(onePath);
    queue<int> vexQueue;
    //be careful of the different connected components.
    for (int i = 0; i < vexNum;++i)
    {
        if(visited[i] == false)
        {
            vexQueue.push(i);
            visited[i] = true;
            while( ! vexQueue.empty() )
            {
                int tempVex = vexQueue.front();
                vexQueue.pop();
                // when poping a vex, it means that I have found it.
                // So record the shortest path of that tempVex.
```

```
                for( int temp = tempVex;orig[temp] != -
1;temp = orig[temp] )
                {
                    shtPath[tempVex].push( G_adjList[temp].info );
                }
                //pushing the tempVex's sons into the stack.
                for (ArcNode *arc = G_adjList[tempVex].firstArc; arc != nul
lptr;
                       arc = arc->next)
                {
                    if (visited[arc->adjvex] == false)
                    {
                        vexQueue.push(arc -> adjvex);
                        visited[arc -> adjvex] = true;
                        //
                        orig[arc->adjvex] = tempVex;
                    }
                }//for (ArcNode *arc = G_adjList[tempVex].firstArc;
            }//while
        }//if
    }//for (int i = 0; i < vexNum;++i)
    //output the shortest path of every vex.
    for(int i = 1; i < vexNum;++i)
    {
        int tempLen = shtPath[i].size();
         cout << G_adjList[i].info << " : " << G_adjList[0].info;
         while( ! shtPath[i].empty() )
         {
             cout << " -> " << shtPath[i].top();
             shtPath[i].pop();
         }
```

```
        cout << "  length: " << tempLen << endl;
    }
}
```

选做实验：无向图中圈的查找

1. **实验要求：**
   以邻接矩阵的形式给定带权的无向图，判断图中是否存在圈，若存在则输出图中的最长圈和圈的长度，圈的长度定义为圈上的边权值之和，而不是边的数
2. **设计思路：**
   对每一个顶点 u，以其为初始顶点，进行 dfs 搜索回路。当搜索到回路之后，判断其是否与之前已搜索到的回路重复，若不重复就记录下来。在这些所有回路中找权值之和最大的回路输出。
3. **核心代码**:
   首先，建立初始化函数 search_all_circle,这个函数初始化了 visited，allpath 变量。其中 visited 记录了结点的访问次序，allPath 记录了所有不重复的已经搜到的圈。然后对每个顶进行 dfs。

```
//OVERALL FUNCTION FOR DFS
//initialization of visited,allPaths
void search_all_circle(const AdjListGraph<char> & G)
{
    vector<VexNode<char>> G_adjList = G.get_adjList();
    int vexNum = G.get_vexNum();
    bool visited[vexNum];
    bool *visitedPtr = visited;
    vector<int> path;
    vector<vector<int>> allPaths;
    for(int i = 0;i < vexNum;++i)
        visited[i] = false;
    //be careful of the different connected part.
    for (int i = 0; i< vexNum;++i)
    {
        dfs_search(G,i,i,visitedPtr,path,allPaths);
    }
```

```
    //output the result.
    int pathNum = allPaths.size();
    cout << "there are " << pathNum << " paths in total.\n";
    for (int i = 0; i < pathNum;++i)
    {
        int pathLen = allPaths[i].size();
        for (int j = 0; j < pathLen;++j)
        {
            cout << G_adjList[(allPaths[i])[j]].info << " ";
        }
        cout << endl;
    }
    // find the circle with the biggest weight.
    if( pathNum > 0)
    {
        int maxCircleWeight = 0;
        int maxCircleIndex = get_longest_circle(G, allPaths, maxCircleWeigh
t);
        cout << "the circle with the greatest weight is \n";
        int maxPathLen = allPaths[maxCircleIndex].size();
        for (int j = 0; j < maxPathLen;++j)
        {
            cout << G_adjList[(allPaths[maxCircleIndex])[j]].info << " ";
        }
        cout << "\nits weight is " << maxCircleWeight << endl;
    }
}
```

其次，写一个 dfs 搜索圈的函数，当搜索到了一个圈之后，对这个圈进行判重和添加。

```
// dfs function for traversing a graph
//RECURSIVE FUNCTION
// almost the same as that of the directed graph.
void dfs_search(const AdjListGraph<char> &G,const int & currVex,
            const int& endVex,bool * visitedPtr,vector<int>& path,
            vector<vector<int>> &allPaths)
{
    vector<VexNode<char>> G_adjList = G.get_adjList();
    visitedPtr[currVex] = true;
    path.push_back(currVex);

    if (currVex == endVex && path.size() != 1 )
    {
        //this is the "fake ring"! we need to backtrack.
        if(path.size() == 3 )
            path.pop_back();
        //it is a circle, so we then need to check repetition and append.
        else
            non_repe_append(path, allPaths);
        return;
    }
    for (ArcNode *arc = G_adjList[currVex].firstArc; arc != nullptr;
         arc = arc->next)
    {
        if (visitedPtr[arc->adjvex] == false || arc->adjvex == endVex)
        {
            dfs_search(G,arc->adjvex,endVex,visitedPtr,path,allPaths);
        }
    }
```

```
    //backtrack in dfs
    visitedPtr[currVex] = false;
    path.pop_back();
}
```

那么判重和添加圈的函数就是这样：如果两个圈的顶点数相同且顶点次序是相同的或者相反的，那么这两个圈就是重复的。

```
/*adding the newly found circle to all the paths.
make sure that this circle is unique.
compared with that append function in the directed graph version,
-the way to judge repetition is different:
1. if the size of path is 3 (such as a b a), then it is illegal.
2. if two sequence of vex in paths is the same or the opposite,
-then the second one is of repetition. */
void non_repe_append(vector<int>& path,vector<vector<int>> &allPaths)
{
    if( path.size() == 3 )
        return;
    path.pop_back();
    int currPathLen = path.size();
    int pathNum = allPaths.size();
    bool repeFlag = false;
    bool equal = true;
    // check whether the new path is of repetition with all the
    // -other paths that has been found
    // by checking whether the sequence is the same or the opposite.
    for (int i = 0; i < 2*pathNum;++i)
    {
        equal = true;
        vector<int> tempPath = allPaths[i/2];
```

```
        //for the second time, check whether the newPath's vex sequence
        // -is opposite to that of the path found.
        if(i % 2 == 1)
            reverse(tempPath.begin(), tempPath.end());
        int tempPathLen = tempPath.size();
        //we don't need to compare one by one.
        if(tempPathLen != currPathLen)
        {
            equal = false;
        }
        else
        {
            int start = -1;
            //find the starting index in the temppath.
            // if have not found, it means not equal.
            for (int j = 0; j < tempPathLen;++j)
            {
                if(tempPath[j] == path[0])
                {
                    start = j;
                    break;
                }
            }
            //have not found the start.
            if(start == -1)
            {
                equal = false;
            }
            //have found the start.
            else
            {
```

```
                //check whether the vex is of repetition one by one.
                for (int j = 0; j < tempPathLen;++j)
                {
                    if( tempPath[ (start + j)%tempPathLen ] != path[j] )
                    {
                        equal = false;
                        break;
                    }
                }
            }//else(start != -1 )
        }//else (tempPathLen == currPathLen)

        if(equal == true)
        {
            repeFlag = true;
            break;
        }
    }//for (int i = 0; i < pathNum;++i)
    if(repeFlag == false)
    {
        allPaths.push_back(path);
    }
}
```

当所有无重复的圈都被找到之后，可以再写一个函数简单地找到最大权值和所对应的圈。

```
/* knowing all of the circles, get the circle with the biggest overall weig
ht.
return the index of that circle in the allPaths vector.
the max Weight is changed as a reference parameter */
```

```
int get_longest_circle(const AdjListGraph<char> & G,vector<vector<int>> &al
lPaths,
                    int & maxCircleWeight)
{
    int maxCircleIndex = 0;
    vector<VexNode<char>> G_adjList = G.get_adjList();
    int pathNum = allPaths.size();
    ArcNode *arc;
    for (int i = 0; i < pathNum;++i)
    {
        int currWeight = 0;
        int pathLen = allPaths[i].size();
        for (int j = 0; j < pathLen;++j)
        {
            // a stupid way to find the weight of each arc
            // given its two vexes.
            arc = G_adjList[ (allPaths[i])[j] ].firstArc;
            while( arc->adjvex != (allPaths[i]) [ (j+1) % pathLen ]  )
                arc = arc->next;
            currWeight += arc->weight;
        }
        if(currWeight > maxCircleWeight)
        {
            maxCircleWeight = currWeight;
            maxCircleIndex = i;
        }
    }
    return maxCircleIndex;
}
```

二．实验测试

对于前两个实验，用的测试数据如下：

```
8 10  // 8 个顶点，10 条边

0 1 2 3 4 5 6 7  //8 个顶点的信息

0 1

0 2

1 2

1 3

1 6

2 5

3 4

3 6

5 6

5 7   //10 条无向边
```

然后得到的结果如下：

```
now getting the cut vertex of the graph.
one cut vertex has been found: 3
one cut vertex has been found: 5
there are 2 cut vertexes in this graph.
```

```
now start to search the shortest path from v0 to every other vertex.
1 : 0 -> 1  length: 1
2 : 0 -> 2  length: 1
3 : 0 -> 1 -> 3  length: 2
4 : 0 -> 1 -> 3 -> 4  length: 3
5 : 0 -> 2 -> 5  length: 2
6 : 0 -> 1 -> 6  length: 2
7 : 0 -> 2 -> 5 -> 7  length: 3
PS C:\Users\Eddie\Desktop\sophomore\data_structure\graph\graph_application>
```

结果正确。

三．实验总结

本次实验通过自己写图的 ADT，图的 dfs，bfs 算法，我更加了解了图的遍历的高妙之处。其中寻找关节点的非递归 dfs 算法是我通过结合书上寻找关节点的代码和吴锋老师讲的非递归深度优先遍历代码写出来的，不禁赞叹把点对压栈的神操作。另外，关于选做题，我虽然可以跑出来，但是感觉方法比较暴力，算法还可以继续优化。比如先找到图的关节点，从关节点开始遍历，当遍历完了之后，就可以把关节点删掉，这样可以省下不少时间。总而言之，能把图论的理论知识和数据结构的实验方法相结合，还是一种很棒的体验。

四．实验源代码

adjListGraph.h　存放邻接表的 ADT
adjMatrixGraph.h 存放邻接矩阵的 ADT
graph_cutVex_nonRecur.cpp 存放实验一的 cpp 文件
graph_shortest_path.cpp 存放实验二的 cpp 文件
graph_longest_circle.cpp 存放实验三的 cpp 文件
lab4_test.cpp 存放测试调用函数和测试数据的 cpp 文件