数据结构课程设计

北京郵電大學

姓　　名 tobeApe6eHok

学　　院 人工智能学院

专　　业 智能科学与技术

班　　级 1111111111

学　　号 1111111111

班内序号 1

指导教师 周延泉

2022 年　5 月

目录

数据结构课程设计实验五

一、实验题目

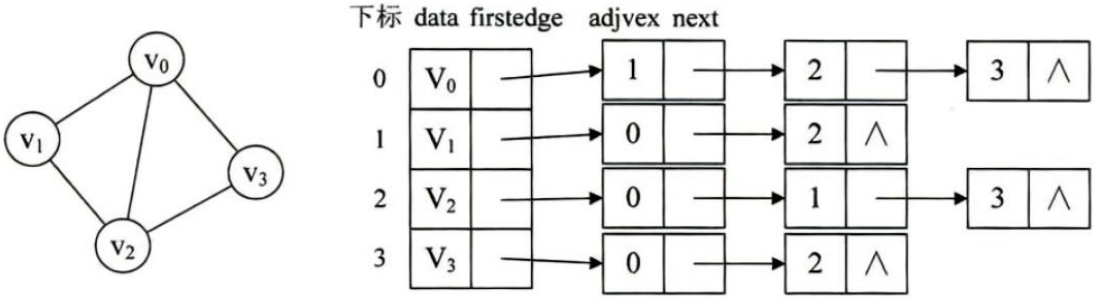
以邻接表为存储结构，设计深度优先遍历的非递归算法。

二、题目分析与算法设计

1、设计思路：

首先我将本次实验中要进行深度优先遍历的图规定为有向图。

其次设计邻接表的结构。邻接表是将数组和链表结合起来表示图的存储结构，邻接表中数组的元素使图节点，在代码中我将其结构定义为 GridNode 模板类，每个节点中包含节点中存储的 Data 字段和相邻图节点的下标链表节点，我将链表节点定义为了 ListNode 模板类，其中包含一个存储节点下标的 int 字段和下一个链表节点的地址。邻接表的结构如下图所示：



为了使得图操作更加方便，我定义了一个图的模板类，使用类中的成员函数实现非递归的深度优先遍历算法。

2、算法描述：

深度优先遍历的主要思想是首先以一个为访问过的节点作为起始节点，沿着当前节点的边走到为访问过的节点，如果当前节点的邻居中没有未访问过的节点，则回到上一个节点，继续试探别的邻居，直到起始节点所属的强连通分量中的节点都被访问。

使用递归的方式很容易实现深度优先遍历，只需要将当前访问到的节点的所有邻居节点依次进行递归访问即可，直到递归结束，就可以保证强连通分量中的所有节点都被访问过。

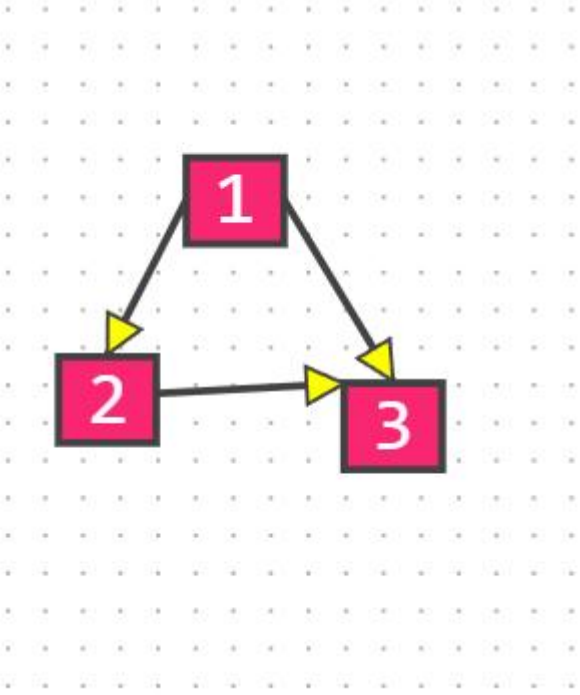
由于递归函数的本质是在内存中不断地产生新的栈帧，不断地压栈和弹出，所以我们可以使用栈数据结构将此递归过程迭代地模拟出

来，具体地伪代码如下：

```
使用 JudgePassed[]来记录节点是否访问过,处置设置为 False
初始化栈 Remain
while Remain 栈不为空:
    获得栈顶元素 Node=Remain.top()
    如果本节点未访问过,则访问内部数据,并将其在 JudgePassed 中设置为 True

    for 每个与 Node 节点相邻的节点
        if 和 Node 相邻 && 未访问过:
            入栈 Remain.push()
```

此处要注意将 JudgePassed 中对应的下标设置为 True 的操作，这个操作要在刚刚访问完该节点时执行，不可以在将本节点入栈时执行，因为如果在入栈时执行，则只能将该节点入栈一次，意味着不可以先于本次入栈访问该节点，但是按照深度优先遍历的思想这是可以的，如下图所示，如果从 1 开始深度优先遍历，访问顺序应该是 1、2、3，而不是 1、3、2：



三、数据结构描述

1、数据结构的选择：

在非递归深度优先遍历算法的实现中我使用了栈数据结构。由于深度优先遍历有着递归函数的形式，而递归函数的本质就是不断地产生新的栈帧序列，所以显然可以使用栈数据结构来模拟递归的过程来实现迭代方式的深度优先遍历。

其次在邻接表的结构中我定义了两个模板类：ListNode 和 GridNode 类，其中 ListNode 类定义了邻接链表中的链表节点的结构，GridNode 类定义了图节点的内部结构。然后我将邻接表和深度优先遍历函数抽象成了一个 Grid 类，内含图邻接表创建、深度优先遍历、邻接表的打印等操作。

2、数据结构的定义：

```
//邻接链表中的边界点
struct ListNode
{
    int place;          //存储该相邻图节点在数组中的下标
    ListNode* next;     //存储下一个相邻节点的指针
    ListNode(int p)     //初始化函数
    {
        place = p;
        next = NULL;
    }
};

//图节点
template<class T>
struct GridNode
{
    T Data;                             //存储图节点内部的信息
    ListNode* NeiborHead;               //存储相邻节点链表的头节点
    GridNode() { NeiborHead = NULL; };  //初始化函数
};
//图模板类
template<class T>
class Grid
{
private:
    GridNode<T>* Graph;
    int GridSize;
    int VertexNum;
public:
    Grid(int GridNodeCnt, T GridNodeInfo[], int VertexCnt, int
GridVertex[]);
    void ShowAdjacentList();
    ~Grid();
    void DeepFirstSearch(int Start=0);
};
```

四、程序清单

以下是我调试好的代码，使用 C++20：

```
#include<iostream>
using namespace std;

//邻接链表中的边界点
struct ListNode
{
    int place;          //存储该相邻图节点在数组中的下标
    ListNode* next;     //存储下一个相邻节点的指针
    ListNode(int p)     //初始化函数
    {
        place = p;
        next = NULL;
    }
};
//图节点
template<class T>
struct GridNode
{
    T Data;                             //存储图节点内部的信息
    ListNode* NeiborHead;               //存储相邻节点链表的头节点
    GridNode() { NeiborHead = NULL; };  //初始化函数
};
//图模板类
template<class T>
class Grid
{
private:
    GridNode<T>* Graph;
    int GridSize;
    int VertexNum;
public:
    Grid(int GridNodeCnt, T GridNodeInfo[], int VertexCnt, int
GridVertex[]);
    void ShowAdjacentList();
    ~Grid();
    void DeepFirstSearch(int Start=0);
};
//构造函数
//GridNodeCnt 表示图节点的数量,GridNodeInfo 依次存储图节点中的信息
//VertextCnt 表示图中有向边的数量,GridVertex 中使用(2*i)和(2*i+1)分别表示边
的始末节点下标
template<class T>
Grid<T>::Grid(int GridNodeCnt, T GridNodeInfo[], int VertexCnt, int
GridVertex[])
{
    this->GridSize = GridNodeCnt;                                   //初始
化图节点的数量
    this->VertexNum=VertexCnt;                                      //初始
化图的边数
```

```
    this->Graph = new GridNode<T>[GridNodeCnt];                          //分配
存储图节点的数组内存
    for (int i = 0; i < GridNodeCnt; i++)
    {
        this->Graph[i].Data = GridNodeInfo[i];                           //依次
初始化图节点的内部信息
        this->Graph[i].NeiborHead = NULL;                                //将邻
居链表的头节点设置为 NULL
    }
    for (int cnt = 0, VP = 0; cnt < VertexCnt; cnt++, VP += 2)       //偶
数下标表示边的起始节点,奇数表示终止节点下标
    {
        //头插链表节点
        ListNode* temp = new ListNode(GridVertex[VP + 1]);
        temp->next = this->Graph[GridVertex[VP]].NeiborHead;
        this->Graph[GridVertex[VP]].NeiborHead = temp;
    }
}
//将创建好的邻接表打印出来
template<class T>
void Grid<T>::ShowAdjacentList()
{
    //遍历每个图节点
    for (int i = 0; i < this->GridSize; i++)
    {
        cout << this->Graph[i].Data <<"("<<i << ") : ";
        ListNode* t = this->Graph[i].NeiborHead;
        //遍历每个图节点的邻居节点下标
        while (t != NULL)
        {
            cout << t->place;
            t = t->next;
            if (t != NULL)
            {
                cout << " -> ";
            }
        }
        cout << endl;
    }
}
//析构函数
template<class T>
Grid<T>::~Grid()
{
    //遍历每个图节点
    for (int i = 0; i < this->GridSize; i++)
    {
        ListNode* t = this->Graph[i].NeiborHead;
        //释放每个链表节点的内存
        while (t != NULL)
        {
            ListNode* tt = t;
            t = t->next;
            delete tt;
        }
```

```
        cout << endl;
    }
    //释放存储图节点的数组内存
    delete[] this->Graph;
}
//图的非递归深度优先遍历函数
template<class T>
void Grid<T>::DeepFirstSearch(int Start)
{
    //初始化记录每个节点有无被访问过的数组
    bool* JudgePassed = new bool[this->GridSize];
    for (int i = 0; i < this->GridSize; i++)
    {
        JudgePassed[i] = 0;//将每个标记设置为 未访问
    }
    //使用数组模拟 栈结构
    //Remain[0]表示栈的大小
    //Remain[Remain[0]]表示栈顶元素
    //+1 是因为 0 下标元素用来表示栈目前占用大小

    int* Remain = new int[this->VertexNum+ 1];
    Remain[0] = 1, Remain[1] = Start;
    //只要栈中还有元素,就表示还有可能有未访问的节点
    while (Remain[0])
    {
        //取出栈顶元素
        int First = Remain[Remain[0]--];

        //栈顶元素是上次访问过的节点的第一个未被访问过的邻居节点
        //与递归的方式对称
        if (!JudgePassed[First])
        {
            JudgePassed[First] = 1;
            cout << First << "  ";//访问
        }
        //获得每个邻居节点
        ListNode* t = this->Graph[First].NeiborHead;
        while (t != NULL)
        {
            //模拟递归方式中的顺序遍历邻居节点
            //迭代方式中只需要将每个未访问的邻居节点都放入栈中即可
            if (!JudgePassed[t->place])
            {
                Remain[++Remain[0]] = t->place;
            }
            t = t->next;
        }
    }
    cout << endl;
    //释放内存
    delete[] JudgePassed;
    delete[] Remain;
}
//测试样例 1
```

```
void test1()
{
    int GridNodeCnt = 8;//表示有 8 个节点
    char GridNodeInfo[] = { 'a','b','c','d','e','f','g','h' };//节点中的
内部信息
    int VertexCnt = 16;//表示有 16 条边
    //(2*i)->(2*i+1)有一条边
    //0->1,1->0,1->2,1->3...
    int GridVertex[] =
{ 0,1,1,0,1,2,1,3,2,4,2,6,3,1,3,5,4,2,4,6,4,7,5,6,6,2,6,4,7,5,7,6 };
    Grid<char> a(GridNodeCnt, GridNodeInfo, VertexCnt, GridVertex);
    cout<<"第一个测试样例结果:\n"<<endl;
    cout << "邻接表结构如下 :" << endl;
    a.ShowAdjacentList();
    cout << "深度优先遍历的节点顺序如下 :" << endl;
    a.DeepFirstSearch();
}

//测试样例 2
void test2()
{
    int GridNodeCnt = 8;//表示有 8 个节点
    int GridNodeInfo[] = { 101,102,103,104,105,106,107,108};//节点中的内部
信息
    int VertexCnt = 14;//表示有 14 条边
    //(2*i)->(2*i+1)有一条边
    //1->5,2->0,2->1,2->5...
    int GridVertex[] =
{ 1,5,2,0,2,1,2,5,3,1,3,5,3,7,4,2,5,1,5,3,5,6,6,2,6,5,7,5 };
    Grid<int> a(GridNodeCnt, GridNodeInfo, VertexCnt, GridVertex);
    cout<<"第二个测试样例结果:\n"<<endl;
    cout << "邻接表结构如下 :" << endl;
    a.ShowAdjacentList();
    cout << "深度优先遍历的节点顺序如下 :" << endl;
    a.DeepFirstSearch(4);
}
int main()
{
    test1();
    test2();
}
```

五、程序复杂度分析

1、时间复杂度：

在深度优先遍历的过程中，每个节点的入栈和出栈次数决定着程序的时间复杂度。如果一个图节点入栈，则表示有一条边指向该节点，同时也表示该条边的起始节点出栈，一旦一个图节点出栈，则代表这个图节点将永远也不会入栈，所以由上面分析可得，一个元素入栈的

次数上限是该节点的入度，所以所有节点入栈的次数最大值是边数。

综上，假设边的个数是 K，则时间复杂度是 O（K）。

2、空间复杂度：

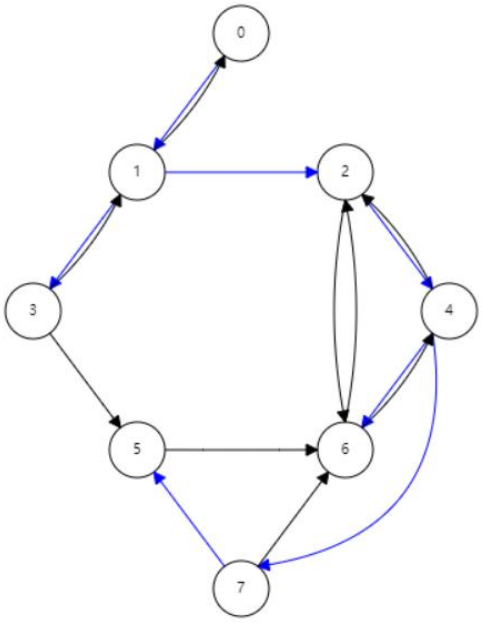
由于每个节点入栈的次数上限是节点入度，所以栈的大小最大也是 K，所以空间复杂度也是 O（K）。

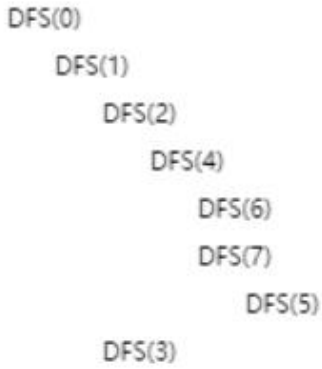
六、程序运行结果

1、实验环境： WIN10 + VScode + C++20.

2、测试用例：

（1）第一个测试样例中的有向图、从 0 节点开始深度优先遍历搜索顺序如下：



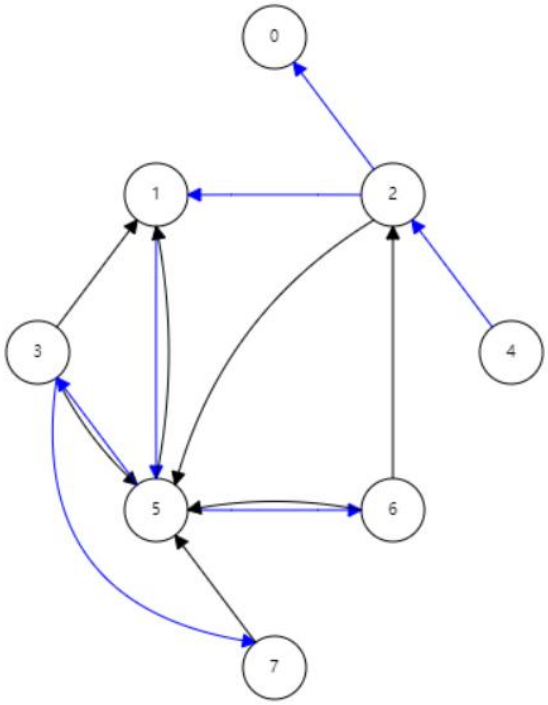


程序中我将 0-7 图节点中内部信息设置成了 a-h.

如果使用数组偶数下标 2*i 表示第 i+1 条边的起始节点，2*i+1 下标表示终止节点，则可以表示成如下：

```
{0,1,1,0,1,2,1,3,2,4,2,6,3,1,3,5,4,2,4,6,4,7,5,6,6,2,6,4,7,5,7,6};
```

（2）第一个测试样例中的有向图、从 0 节点开始深度优先遍历搜索顺序如下：



DFS(4)
DFS(2)
DFS(0)
DFS(1)
DFS(5)
DFS(3)
DFS(7)
DFS(6)

程序中我将 0-7 图节点的内部信息设置成了 101-108.

如果使用数组偶数下标 2*i 表示第 i+1 条边的起始节点，2*i+1 下标表示终止节点，则可以表示成如下：

```
{ 1,5,2,0,2,1,2,5,3,1,3,5,3,7,4,2,5,1,5,3,5,6,6,2,6,5,7,5 };
```

3、测试结果：

```
第一个测试样例结果:

邻接表结构如下 :
a(0) : 1
b(1) : 3 -> 2 -> 0
c(2) : 6 -> 4
d(3) : 5 -> 1
e(4) : 7 -> 6 -> 2
f(5) : 6
g(6) : 4 -> 2
h(7) : 6 -> 5
深度优先遍历的节点顺序如下 :
0  1  2  4  6  7  5  3



第二个测试样例结果:

邻接表结构如下 :
101(0) :
102(1) : 5
103(2) : 5 -> 1 -> 0
104(3) : 7 -> 5 -> 1
105(4) : 2
106(5) : 6 -> 3 -> 1
107(6) : 5 -> 2
108(7) : 5
深度优先遍历的节点顺序如下 :
4  2  0  1  5  3  7  6
```