# 《数据结构与算法》作业三

姓名：余弦 学号：2021112893 班级：2103401

## 1. 图的存储

**领接矩阵存储图**

```
int mtx[N][N];
DATA data[N];//用于储存节点的数据
int dim;
```

其中，mtx用于储存节点之间的连接情况，data数组用于储存每个节点储存的数据，dim用于储存节点的个数。

显然，对于一个有n个节点的图，其空间占用情况为$O(n^3)$

**领接表储存图**

```
//节点的定义
typedef struct node{
    DATA elm;
    int next[N];
    int next_num = 0;
}NODE;
NODE* head[N];
int dim;
```

其中，head数组用于存储每个节点的地址，dim用于储存节点的个数。next用于存储指向下一个节点的索引。对于一个有n个节点和m条边的图，需要n个节点空间存储节点地址，m个int空间用来储存节点的指向关系。所以，其空间占用情况为$O(nm)$

## 2. 图存储类型的转换

**领接矩阵转邻接表**

```
graph_link_table* matrix_to_link(graph_matrix* matrix){
    graph_link_table* link = new graph_link_table(matrix -> dim);
    for(int i = 1; i <= matrix -> dim; i++){
        link -> add_node(i, matrix -> data[i]);
    }
    for(int i = 1; i <= matrix -> dim; i++){
        for(int j = 1; j <= matrix -> dim; j++){
            if(matrix -> mtx[i][j] != 0){
                link -> add_edge(i, j);
            }
        }
    }
    return link;
}
```

领接表转领接矩阵

```
graph_matrix* link_to_matrix(graph_link_table* link){
    graph_matrix* matrix = new graph_matrix(link -> dim);
    for(int i = 1; i <= (link -> dim); i++){
        matrix -> add_node(i, link -> head[i] -> elm);
        for(int j = 1; j <= (link -> dim); j++){
            if(link -> head[i] -> next[j] != 0){
                matrix -> add_edge(i, j);
            }
        }
    }
    return matrix;
}
```

## 3.深度优先遍历与广度优先遍历

**代码**

```
void mtx_dfs(graph_matrix &g, int x, int num){
    st[x] = true;
    printf_s("V%d:%d ->", x, g.get_elm(x));
    if(num == g.dim){
        printf_s(".\n");
        return;
    }
    for(int i = 1; i <= g.dim; i++){
        if(g.mtx[x][i] > 0 && !st[i]){
            mtx_dfs(g, i, num + 1);
            break;
        }
    }
    if(num < g.dim){
        for(int i = 1; i <= g.dim; i++){
            if(!st[i]){
                printf_s(".\n");
                mtx_dfs(g, i, num + 1);
            }
        }
    }
}
void mtx_bfs(graph_matrix &g, int x, int num){
    vector<int> q;
    q.push_back(x);
    st[x] = true;
    while(!q.empty()){
        int t = q[0];
        q.erase(q.begin());
        printf_s("V%d:%d ->", t, g.get_elm(t));
        for(int i = 1; i <= g.dim; i++){
            if(g.mtx[t][i] > 0 && !st[i]){
                q.push_back(i);
                num++;
                st[i] = true;
            }
        }
    }
```

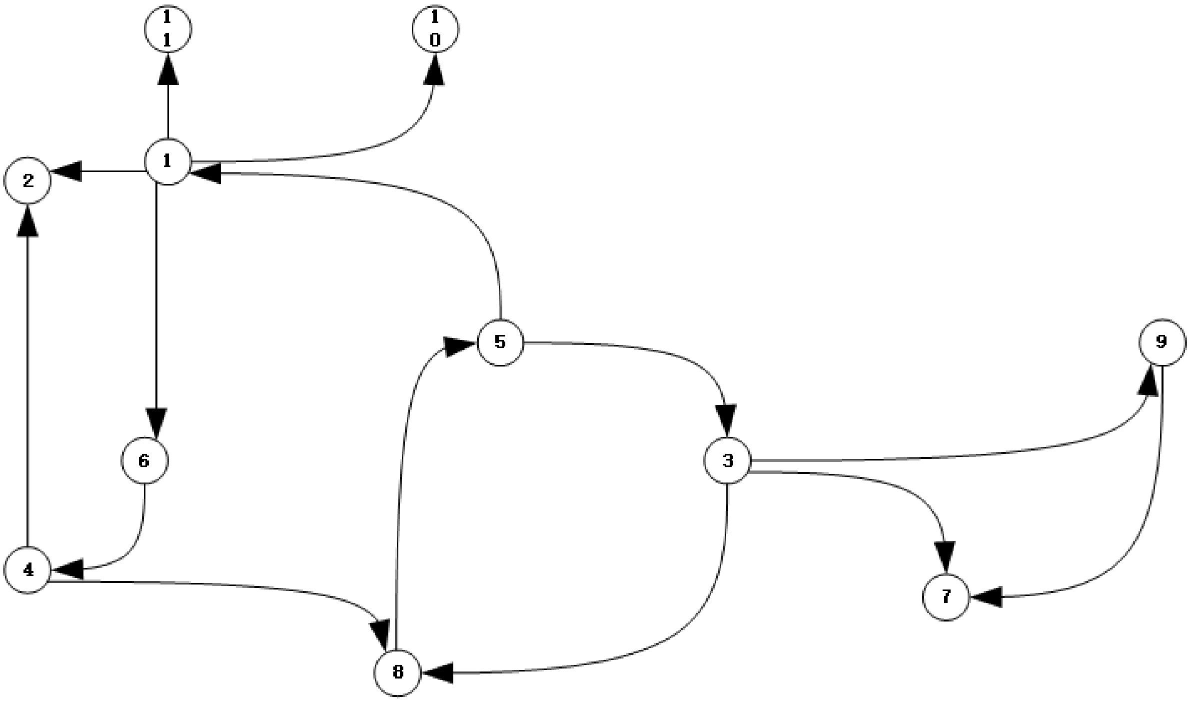
```
    if(num == g.dim){
        printf_s(".\n");
        return;
    }
    for(int i = 1; i <= g.dim; i++){
        if(!st[i]){
            printf_s(".\n");
            mtx_bfs(g, i, num + 1);
        }
    }
}

void link_dfs(graph_link_table &g, int x, int num){
    st[x] = true;
    printf_s("V%d:%d ->", x, g.head[x] -> elm);
    if(num == g.dim){
        printf_s(".\n");
        return;
    }
    for(int i = 0; i < g.head[x] -> next_num; i++){
        if(!st[g.head[x] -> next[i]]){
            link_dfs(g, g.head[x] -> next[i], num + 1);
            break;
        }
    }
    if(num < g.dim){
        for(int i = 1; i <= g.dim; i++){
            if(!st[i]){
                printf_s(".\n");
                link_dfs(g, i, num + 1);
            }
        }
    }
}
void link_bfs(graph_link_table &g, int x, int num){
    vector<int> q;
    q.push_back(x);
    st[x] = true;
    while(!q.empty()){
        int t = q[0];
        q.erase(q.begin());
        printf_s("V%d:%d ->", t, g.head[t] -> elm);
        for(int i = 0; i < g.head[t] -> next_num; i++){
            if(!st[g.head[t] -> next[i]]){
                q.push_back(g.head[t] -> next[i]);
                num++;
                st[g.head[t] -> next[i]] = true;
            }
        }
    }
    if(num == g.dim){
        printf_s(".\n");
        return;
    }
    for(int i = 1; i <= g.dim; i++){
```

```
        if(!st[i]){
            printf_s(".\n");
            link_bfs(g, i, num + 1);
        }
    }
}
```

**运行结果**

对于图




遍历结果为

```
DFS:
V1:1 ->V2:2 ->.
V3:3 ->V7:7 ->.
V4:4 ->V8:8 ->V5:5 ->.
V6:6 ->.
V9:9 ->.
V10:10 ->.
V11:11 ->.

BFS:
V1:1 ->V2:2 ->V6:6 ->V10:10 ->V11:11 ->V4:4 ->V8:8 ->V5:5 ->V3:3 ->V7:7 ->V9:9 ->.

DFS:
V1:1 ->V2:2 ->.
V3:3 ->V7:7 ->.
V4:4 ->V8:8 ->V5:5 ->.
V6:6 ->.
V9:9 ->.
V10:10 ->.
V11:11 ->.

BFS:
V1:1 ->V2:2 ->V6:6 ->V10:10 ->V11:11 ->V4:4 ->V8:8 ->V5:5 ->V3:3 ->V7:7 ->V9:9 ->.
```

**复杂度分析**

若采用邻接矩阵存储图：

对于深度优先遍历，对于每个节点，需要遍历每一个节点来判断是都邻接，故其时间复杂度为$O(n^2)$

而对于广度优先遍历，对于每个节点都需要遍历所有节点，将相连的顶点加入到队列当中，其时间复杂度为$O(n^2)$，m为图边的条数。

若采用领接表存储图：

对于深度优先遍历，顶点直接的连接关系已有，故无需遍历，时间复杂度为$O(n)$

对于广度优先遍历，需要将某个顶点所有的相邻接的顶点加入到队列当中，故其时间复杂度为$O(m+n)$，其中m为边的条数

## 4.出度入度的计算

**代码**

```
//邻接表
int in_degree(int num){
        int cnt = 0;
        for(int i = 1; i <= this -> dim; i++){
            for(int j = 0; j < head[i] -> next_num; j++){
                if(head[i] -> next[j] == num) cnt ++;
            }
        }
        return cnt;
    }
int out_degree(int num){
    return head[num] -> next_num;
}

//邻接矩阵
int in_degree(int num){
    int cnt = 0;
    for(int i = 1; i <= this->dim; i++){
        cnt += mtx[i][num];
    }
    return cnt;
}

int out_degree(int num){
    int cnt = 0;
    for(int i = 1; i <= this -> dim; i++){
        cnt += mtx[num][i];
    }
    return cnt;
}
```

**测试结果**

```
V1:1 -> in_degree:1, out_degree:4
V2:2 -> in_degree:2, out_degree:0
V3:3 -> in_degree:1, out_degree:3
V4:4 -> in_degree:1, out_degree:2
V5:5 -> in_degree:1, out_degree:2
V6:6 -> in_degree:1, out_degree:1
V7:7 -> in_degree:2, out_degree:0
V8:8 -> in_degree:2, out_degree:1
V9:9 -> in_degree:1, out_degree:1
V10:10 -> in_degree:1, out_degree:0
V11:11 -> in_degree:1, out_degree:0

V1:1 -> in_degree:1, out_degree:4
V2:2 -> in_degree:2, out_degree:0
V3:3 -> in_degree:1, out_degree:3
V4:4 -> in_degree:1, out_degree:2
V5:5 -> in_degree:1, out_degree:2
V6:6 -> in_degree:1, out_degree:1
V7:7 -> in_degree:2, out_degree:0
V8:8 -> in_degree:2, out_degree:1
V9:9 -> in_degree:1, out_degree:1
V10:10 -> in_degree:1, out_degree:0
V11:11 -> in_degree:1, out_degree:0
```

**复杂度分析**

邻接矩阵

不管是计算入度还是出度，只需要把每一行或者每一列的值相加即可，故时间复杂度为$O(n)$

对于邻接表

计算入度的时候，需要扫描其他点的领接矩阵，看是否有边进入，故时间复杂度为$O(n+m)$

计算出度是，只需要直接读取next数组中元素的个数即可，故时间复杂度为$O(1)$

### 5.点和边的输入方式

```
input the number of nodes:11
input the number of edges:
14
1 2
1 11
1 10
1 6
3 8
3 7
3 9
4 2
4 8
5 1
5 3
6 4
8 5
9 7
```

附录:

输入

```
11
14
1 2
1 11
1 10
1 6
3 8
3 7
3 9
4 2
4 8
5 1
5 3
6 4
8 5
9 7
```

完整源代码

```cpp
#include <iostream>
#include "vector"
#include <cstring>
#define DATA int
const int N = 100;
using namespace std;
typedef struct node{
    DATA elm;
    int next[N];
    int next_num = 0;
}NODE;
bool st[N];
class graph_matrix{
public:
    int mtx[N][N];
    DATA data[N];//用于储存节点的数据
    int dim;

    graph_matrix(int n){
        memset(this -> mtx, -1, sizeof(this ->mtx));
        memset(this -> data, 0, sizeof(this -> data));
        this -> dim = n;
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                this -> mtx[i][j] = 0;
            }
        }
    }

    void add_node(int num, DATA elm){
        if(!is_legal(num)){
            printf_s("ERROR!\n");
            return;
        }
        this -> data[num] = elm;
    }


    bool is_legal(int num){
        return num > 0 && num <= this -> dim;
    }

    void change_node_val(int num, DATA elm){
        if(!is_legal(num)){
            printf_s("ERROR!\n");
            return;
        }
        data[num] = elm;
    }

    void add_edge(int x, int y){
        if(!(is_legal(x) && is_legal(y))){
            printf_s("ERROR!\n");
        }
        this -> mtx[x][y] ++;
```

```
    }

    DATA get_elm(int num){
        if(!is_legal(num)){
            printf_s("ERROR!\n");
            return -1;
        }
        return this -> data[num];
    }

    int in_degree(int num){
        int cnt = 0;
        for(int i = 1; i <= this->dim; i++){
            cnt += mtx[i][num];
        }
        return cnt;
    }

    int out_degree(int num){
        int cnt = 0;
        for(int i = 1; i <= this -> dim; i++){
            cnt += mtx[num][i];
        }
        return cnt;
    }
};
void mtx_dfs(graph_matrix &g, int x, int num){
    st[x] = true;
    printf_s("V%d:%d ->", x, g.get_elm(x));
    if(num == g.dim){
        printf_s(".\n");
        return;
    }
    for(int i = 1; i <= g.dim; i++){
        if(g.mtx[x][i] > 0 && !st[i]){
            mtx_dfs(g, i, num + 1);
            break;
        }
    }
    if(num < g.dim){
        for(int i = 1; i <= g.dim; i++){
            if(!st[i]){
                printf_s(".\n");
                mtx_dfs(g, i, num + 1);
            }
        }
    }
}
void mtx_bfs(graph_matrix &g, int x, int num){
    vector<int> q;
    q.push_back(x);
    st[x] = true;
    while(!q.empty()){
        int t = q[0];
        q.erase(q.begin());
```

```
        printf_s("V%d:%d ->", t, g.get_elm(t));
        for(int i = 1; i <= g.dim; i++){
            if(g.mtx[t][i] > 0 && !st[i]){
                q.push_back(i);
                num++;
                st[i] = true;
            }
        }
    }
    if(num == g.dim){
        printf_s(".\n");
        return;
    }
    for(int i = 1; i <= g.dim; i++){
        if(!st[i]){
            printf_s(".\n");
            mtx_bfs(g, i, num + 1);
        }
    }
}

class graph_link_table{
public:
    //利用领接表实现图
    NODE* head[N];
    int dim;
    graph_link_table(int n){
        this -> dim = n;
        for(int i = 1; i <= n; i++){
            this -> head[i] = nullptr;
        }
    }
    bool if_legal(int num){
        return (num > 0 && num <= this ->dim);
    }
    bool is_node(int num){
        if(!head[num]){
            printf_s("ERROR:The node has not been created!\n");
            return 0;
        }
        return 1;
    }
    void add_node(int num, DATA data){
        if(!if_legal(num)){
            printf_s("ERROR!\n");
        }
        if(!head[num]){
            head[num] = (NODE*) malloc(sizeof(NODE));
            head[num] -> elm = data;
            head[num] -> next_num = 0;
        }
        else{
            printf_s("ERROR:Node has existed!\n");
        }
    }
```

```
    void change_node_val(int num, DATA data){
        if(!if_legal(num)){
            printf_s("ERROR!\n");
            return;
        }
        if(!head[num]){
            printf_s("ERROR:The node has not been created!\n");
            return;
        }
        head[num] -> elm = data;
    }
    void add_edge(int x, int y){
        if(is_node(x) and is_node(y)){
            head[x] -> next[head[x] -> next_num++] = y;
        }
    }

    void delete_edge(int x, int y){
        if(is_node(x) and is_node(y)){
            if(head[x] -> next[y] > 0) head[x] -> next[y] --;
            else printf_s("ERROR:There is no edge between %d and %d!\n", x, y);
        }
    }

    int in_degree(int num){
        int cnt = 0;
        for(int i = 1; i <= this -> dim; i++){
            for(int j = 0; j < head[i] -> next_num; j++){
                if(head[i] -> next[j] == num) cnt ++;
            }
        }
        return cnt;
    }

    int out_degree(int num){
        return head[num] -> next_num;
    }

};
void link_dfs(graph_link_table &g, int x, int num){
    st[x] = true;
    printf_s("V%d:%d ->", x, g.head[x] -> elm);
    if(num == g.dim){
        printf_s(".\n");
        return;
    }
    for(int i = 0; i < g.head[x] -> next_num; i++){
        if(!st[g.head[x] -> next[i]]){
            link_dfs(g, g.head[x] -> next[i], num + 1);
            break;
        }
    }
    if(num < g.dim){
        for(int i = 1; i <= g.dim; i++){
```

```
            if(!st[i]){
                printf_s(".\n");
                link_dfs(g, i, num + 1);
            }
        }
    }
}
void link_bfs(graph_link_table &g, int x, int num){
    vector<int> q;
    q.push_back(x);
    st[x] = true;
    while(!q.empty()){
        int t = q[0];
        q.erase(q.begin());
        printf_s("V%d:%d ->", t, g.head[t] -> elm);
        for(int i = 0; i < g.head[t] -> next_num; i++){
            if(!st[g.head[t] -> next[i]]){
                q.push_back(g.head[t] -> next[i]);
                num++;
                st[g.head[t] -> next[i]] = true;
            }
        }
    }
    if(num == g.dim){
        printf_s(".\n");
        return;
    }
    for(int i = 1; i <= g.dim; i++){
        if(!st[i]){
            printf_s(".\n");
            link_bfs(g, i, num + 1);
        }
    }
}
//将图的邻接表转换为邻接矩阵
graph_matrix* link_to_matrix(graph_link_table* link){
    graph_matrix* matrix = new graph_matrix(link -> dim);
    for(int i = 1; i <= (link -> dim); i++){
        matrix -> add_node(i, link -> head[i] -> elm);
        for(int j = 1; j <= (link -> dim); j++){
            if(link -> head[i] -> next[j] != 0){
                matrix -> add_edge(i, j);
            }
        }
    }
    return matrix;
}
//将图的邻接矩阵转换为邻接表
graph_link_table* matrix_to_link(graph_matrix* matrix){
    graph_link_table* link = new graph_link_table(matrix -> dim);
    for(int i = 1; i <= matrix -> dim; i++){
        link -> add_node(i, matrix -> data[i]);
    }
    for(int i = 1; i <= matrix -> dim; i++){
        for(int j = 1; j <= matrix -> dim; j++){
```

```
                if(matrix -> mtx[i][j] != 0){
                    link -> add_edge(i, j);
                }
            }
        }
        return link;
}

void set_false(bool st[]){
    for(int i = 1; i <= N; i++){
        st[i] = false;
    }
}
//将边保存到vector中
int main() {
    cout<<"input the number of nodes:";
    int nn;
    cin>>nn;
    graph_matrix G(nn);
    //**添加点和边**//
    //添加点
    for(int i = 1; i <= nn; i++){
        G.add_node(i, i);
    }
    //添加边
    int tmp;
    cout<<"input the number of edges:"<<endl;
    cin>>tmp;
    for(int i = 0; i < tmp; i++){
        int x, y;
        cin>>x>>y;
        G.add_edge(x, y);
    }
    graph_link_table* g = matrix_to_link(&G);
    set_false(st);
    printf_s("DFS:\n");
    mtx_dfs(G,1, 1);
    printf_s("\n");
    set_false(st);
    printf_s("BFS:\n");
    mtx_bfs(G, 1, 1);
    printf_s("\n");
    set_false(st);
    printf_s("DFS:\n");
    link_dfs(*g, 1, 1);
    printf_s("\n");
    set_false(st);
    printf_s("BFS:\n");
    link_bfs(*g, 1, 1);

    //每个节点的入度和出度
    for(int i = 1; i <= G.dim; i++){
        printf_s("V%d:%d -> in_degree:%d, out_degree:%d\n", i, G.data[i],
G.in_degree(i), G.out_degree(i));
    }
```

```c
    puts("");
    for(int i = 1; i <= g -> dim; i++){
        printf_s("V%d:%d -> in_degree:%d, out_degree:%d\n", i, g -> head[i] -> elm, g -> in_degree(i), g -> out_degree(i));
    }
    return 0;
}

```