天津大学

程序设计综合实践课程报告

图论实验

学生姓名 陈昊昆

学院名称 智算学部

专　　业 软件工程

学　　号 3021001196

1. dfs

1.1 题目分析

需要建立邻接矩阵和访问标记

从节点 1 开始，搜索到其所有未被访问的相邻节点，再从这些相邻节点开始搜索它们未被访问的相邻节点，以此类推，直到所有节点都被访问过为止

在 DFS 中，需要不断更新访问标记，并将遍历过的节点序号输出

1.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int n, e;

// 深度优先遍历
void dfs(int v, bool** adj, bool* visited){
    if (visited[v-1] == 1) return;  // 若已经被访问，则不再访问
    visited[v-1] = 1;  // 标记被访问
    cout << v << " ";
    for(int i = 0; i < n; i++){
        if(adj[v-1][i] == 1){
            dfs(i+1, adj, visited);  // 访问下一个节点
        }
    }
}

int main(){
    cin >> n >> e;
    bool **adj = new bool* [n];  // 邻接矩阵
    bool *visited = new bool [n]; // 标记是否被访问
    for(int i = 0; i < n; i++) adj[i] = new bool [n];

    // 初始化
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            adj[i][j] = false;
```

```
        }
    }

    for(int i = 0; i < e; i++){
        visited[i] = false;
    }

    for(int j = 0; j < e; j++){
        int v1, v2;
        cin >> v1 >> v2;
        adj[v1-1][v2-1] = true;
        adj[v2-1][v1-1] = true;
    }

    // 从节点 1 开始访问
    dfs(1, adj, visited);

}
```

2. bfs

2.1 题目分析

利用队列来进行 BFS 搜索

先将节点 1 入队

每弹出一个节点，就将所有该节点邻接且未被访问的节点入队，并标记已经访问

直到队列为空

2.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int n, e;

// 广度优先遍历
void bfs(bool** adj, bool* visited){
    queue <int>q;
    q.push(0);     // 从节点 1 开始访问
    while(!q.empty()){
        int v;
        v = q.front();
        q.pop();
        visited[v] = 1;
        cout << v+1 << " ";

        // 将所有该节点连接且未被访问的节点入队
        for(int i = 0; i < n; i++){
            if(adj[v][i] == 1 && visited[i] == 0) {
                visited[i] = 1;
                q.push(i);
            }
        }
    }

}
```

```
int main(){
    cin >> n >> e;
    bool **adj = new bool* [n];  // 邻接矩阵
    bool *visited = new bool [n]; // 标记是否被访问
    for(int i = 0; i < n; i++) adj[i] = new bool [n];

    // 初始化
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            adj[i][j] = false;
        }
    }

    for(int i = 0; i < n; i++){
        visited[i] = false;
    }

    for(int j = 0; j < e; j++){
        int v1, v2;
        cin >> v1 >> v2;
        adj[v1-1][v2-1] = true;
        adj[v2-1][v1-1] = true;
    }


    bfs(adj, visited);

}
```

3. 蜜罐

3.1 题目分析

利用 Kruskal 算法，得到最小生成树

寻找 n-1 条边，按权值从小到大排列

若不构成环，则加入最小生成树

若遍历的所有边，还没有找齐 n-1 条边，则没有生成树

3.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int n, e, totalw;


void Kruskal (int** adj, bool* visited){
    int u = n - 1; // 最小生成树的边数
    int t = 0;  // 已经检查的边数
    while(u > 0){
        t++;
        int min = 10000;
        int mi, mj;
        // 寻找最小权值的边
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                if(adj[i][j] < min) {
                    min = adj[i][j];
                    mi = i;
                    mj = j;
                }
            }
        }

        // 若不能构成环，则加入最小生成树
        if(visited[mi] == 0 || visited[mj] == 0){
            totalw += min;
```

```
            visited[mi] = 1;
            visited[mj] = 1;
            u--;
        }

        // 不在检查这条边
        adj[mi][mj] = 10000;

        // 如果已经检查了所有边，还没有退出，则没有生成树
        if(t == e+1){
            totalw = 0;
            return;
        }
    }
}

int main(){
    int num;
    cin >> num;
    for(int p = 0; p < num; p++){
        cin >> n >> e;
        int **adj = new int* [n];  // 邻接矩阵
        bool *visited = new bool [n]; // 标记是否被访问
        for(int i = 0; i < n; i++) adj[i] = new int [n];

        // 初始化
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                adj[i][j] = 10000;
            }
        }

        for(int i = 0; i < n; i++){
            visited[i] = false;
        }

        // 记录权值
        for(int j = 0; j < e; j++){
            int v1, v2, w;
            cin >> v1 >> v2 >> w;
            adj[v1-1][v2-1] = w;
            adj[v2-1][v1-1] = w;
        }
```

```
        Kruskal (adj, visited);

        cout << totalw;
    }


}
```

4. 村村通

4.1 题目分析

创建一个数组，存放联通集的情况
对每一条边进行判断，创建新联通集或把节点加入某一联通集或合并联通集
最后需要添加的边数就为联通集数量减一

4.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n, e;
    while(cin >> n >> e){
        set<int> s;  // 创建一个集合，用于统计联通集个数
        int w = n;
        int x;

        int *arr = new int [n];
        for(int i = 0; i < n; i++) arr[i] = i;

        if(e == 0) cin >> x;

        // 判断是否在一个联通集中
        for(int i = 0; i < e; i++){
            int v1, v2;
            cin >> v1 >> v2;
            int vv1 = v1-1;
            int vv2 = v2-1;

            // 创建新联通集
            if(arr[vv1] == vv1 && arr[vv2] == vv2){
                arr[vv1] = w;
                arr[vv2] = w;
                w++;
            }
```

```
            // 把节点加入某一联通集
            else if(arr[vv1] != vv1 && arr[vv2] == vv2){
                arr[vv2] = arr[vv1];
            }

            else if(arr[vv1] == vv1 && arr[vv2] != vv2){
                arr[vv1] = arr[vv2];
            }

            // 合并联通集
            else{
                if(arr[vv1] < arr[vv2]) {
                    for(int u = 0; u < n; u++){
                        if(arr[u] == arr[2]) arr[u] = arr[1];
                    }
                }
                else {
                    for(int u = 0; u < n; u++){
                        if(arr[u] == arr[1]) arr[u] = arr[2];
                    }
                }

            }


        }

        // 统计联通集个数
        for(int i = 0; i < n; i++){
            s.insert(arr[i]);
        }

        cout << s.size() - 1 << endl;

    }
}
```

5. 一个人的旅行

5.1 题目分析

利用 dijkstra 算法 求单源最短路
然后计算所有起点的 d 数组 找出所有终点中 d 值最小的代价

5.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;
#define inf 1000
#define N 100

int adj[N][N];
int d[N];
bool visited[N];
int t,s,dd;

// dijkstra 算法 求单源最短路
void dijkstra(int v){
    for(int i = 0; i < N; i++) {
        d[i] = inf;
        visited[i] = 0;
        }
    d[v] = 0;
    // 寻找 d[i]最小的 i 出队
    for(int i = 0; i < N; i++){
        int md = inf;
        int mi;
        int j = 0;
        for(j = 0; j < N; j++){
            if (d[j] < md && visited[j] == 0){
                md = d[j];
                mi = j;
            }
        }
        visited[mi] = 1;
        // 重新计算其邻接节点的 d
```

```
        for(int u = 0; u < N; u++){
            if(adj[mi][u] > 0 && visited[u] == 0){
                d[u] = min(d[u], d[mi] + adj[mi][u]);
            }
        }
    }
}

int main(){
    cin >> t >> s >> dd;
    int *start = new int [s];
    int *des = new int [dd];
    for(int i = 0; i < t; i++){
        int v1, v2, w;
        cin >> v1 >> v2 >> w;
        int vv1 = v1 - 1;
        int vv2 = v2 - 1;
        adj[vv1][vv2] = w;
        adj[vv2][vv1] = w;
    }
    for(int i = 0; i < s; i++) {
        int v;
        cin >> v;
        int vv = v - 1;
        start[i] = vv;
    }
    for(int i = 0; i < dd; i++) {
        int v;
        cin >> v;
        int vv = v - 1;
        des[i] = vv;
    }

    int min = inf;

    // 计算所有起点的 d 数组 找出所有终点中 d 值最小的代价
    for(int i = 0; i < s; i++){
        dijkstra(start[i]);
        for(int j = 0; j < dd; j++){
            if(d[des[j]] < min) min = d[des[j]];
        }
    }

    cout << min;
```

```
}
```

6. 文化之旅

6.1 题目分析

利用 Dijkstra 算法，但是是有限制的搜索

不能访问已学习的文化国家和文化冲突的国家

将已经访问过的国家的文化计入一个集合

当访问新的国家时，遍历集合，看是否有冲突

6.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;
#define inf 10000 //定义一个无穷大的值



int n, k, m, s, t;
int we[105];  //we 数组表示每个国家的文化
int g[105][105];  //g 数组表示各个文化之间是否互相排斥
int e[105][105];  //e 数组 a 表示各个国家之间的距离
bool v[105];
int d[105];  //d 数组表示源点 s 到各个点的最短距离

void dijkstra()
{
    set <int> setset;
    int i, j;
    memset(v, 0, sizeof(v));  //v 数组表示某个点是否已被访问过，初始化为 0
    // v[s] = 1;  //源点 s 标记为已访问
    // setset.insert(s);

    for(i = 1; i <= n; i++) d[i] = inf;  //初始化 d 数组为源点 s 到各个点的
距离
    d[s]  =  0;
    while(1)
    {
        int k = -1;
        int mm = inf;
```

```
        setset.clear();
        for(i = 1; i <= n; i++)
        {
            if(!v[i]&&d[i]<mm) //如果 i 未被访问且 i 到源点 s 的距离比 mm 更小
            {
                k = i;  //记录距离 s 最近的未访问点
                mm = d[i];  //更新最小距离
            }
        }
        // cout << "DFDF";
        if(k == -1) break;  //如果所有点都已被访问，则结束
        v[k] = 1;  //标记 k 已被访问
        setset.insert(k);
        for(i = 1; i <= n; i++)
        {
            bool flag1  =  false;
            bool flag2  =  false;
            set<int>::iterator it;
            for(it = setset.begin();  it!= setset.end(); it++){
                flag1  =  flag1 || g[i][*it];
                //cout << flag << endl;
            }
            set<int>::iterator itt;
            for(itt = setset.begin();  itt!= setset.end(); itt++){
                if (we[i]  ==  we[*itt]) flag2  =  true;
            }

            if(!flag2&&!flag1&&d[i]>d[k]+e[k][i])
            //如果 k 和 i 的文化不同、k 和 i 的文化之间不互相排斥、从源点 s 到 i 的距离比从源点 s 到 k 再到 i 的距离更短
            {
                //cout << "DFDF";
                d[i] = d[k]+e[k][i];  //更新从源点 s 到 i 的最短距离
                // cout << i << " " << d[i] << endl;
            }




        }
    }
}

int main()
{
```

```
    cin>>n>>k>>m>>s>>t;
    int i, j;
    for(i = 1; i<= n; i++)
        for(j = 1; j<= n; j++)
        {
            if(i == j) e[i][j] = 0;  //如果 i 和 j 相等，表示这是同一个点，距离为 0
            else e[i][j] = inf;  //否则初始化为无穷大
        }
    for(i = 1; i<= n; i++) cin>>we[i];  //输入每个国家的文化
    for(i = 1; i<= k; i++)
    {
        for(j = 1; j<= k; j++)
        {
            cin>>g[i][j];  //输入文化之间的排斥情况
        }
    }
    for(i = 1; i<= k; i++)
    {
        for(j = 1; j<= k; j++)
        {
            if(i  ==  j) g[i][j]  =  1;
        }
    }
    for(i = 1; i<= m; i++)
    {
        int x, y, z;
        cin>>x>>y>>z;
        if(e[x][y]>z)
        {
            e[x][y] = z;
            e[y][x] = z;
        }
    }
    dijkstra();
    int ooo =  -1;
    if(d[t] == inf)
    cout<<ooo;
    else
    cout<<d[t];
    return 0;
}
```

7. 公交线路

7.1 题目分析

利用 Dijkstra 算法，求单源最短路问题

7.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;
#define inf 10000
#define N 1000

int adj[N][N];
int d[N];
bool visited[N];

// dijkstra 算法 求单源最短路
void dijkstra(int v){
    for(int i = 0; i < N; i++) {
        d[i] = inf;
        visited[i] = 0;
        }
    d[v] = 0;
    // 寻找 d[i]最小的 i 出队
    for(int i = 0; i < N; i++){
        int md = inf;
        int mi;
        int j = 0;
        for(j = 0; j < N; j++){
            if (d[j] < md && visited[j] == 0){
                md = d[j];
                mi = j;
            }
        }
        visited[mi] = 1;
        // 重新计算其邻接节点的 d
        for(int u = 0; u < N; u++){
            if(adj[mi][u] > 0 && visited[u] == 0){
```

```
                d[u] = min(d[u], d[mi] + adj[mi][u]);
            }
        }
    }
}

int main(){
    int n,e,s,t;
    cin >> n >> e >> s >> t;
    for(int i = 0; i < e; i++){
        int v1, v2, w;
        cin >> v1 >> v2 >> w;
        int vv1 = v1 - 1;
        int vv2 = v2 - 1;
        adj[vv1][vv2] = w;
        adj[vv2][vv1] = w;
    }


    dijkstra(s-1);

    if(d[t-1] == inf) cout << -1;
    else cout << d[t-1];




}
```

8. 弗洛伊德

8.1 题目分析

利用动态规划的思想，引入中间节点

对任意点对（i，j），考虑所有中间节点 k 的情况，判断路程是否缩短

8.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

const int MAXN = 100;
const int inf = 60000;
int d[MAXN][MAXN]; //表示任意两个点之间的距离

// 三重循环实现 Floyd 算法
void Floyd(int n)
{
    // 插入 k 节点
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (d[i][j] > d[i][k] + d[k][j]) {
                    d[i][j] = d[i][k] + d[k][j];
                }
                if(i == j) d[i][j] = 0;
            }
        }
    }
}

int main(){
    int n;
    cin >> n;
    for(int i = 1; i <= n; i++ ){
        for(int j = 1; j <= n; j++){
            int w;
            cin >> w;
```

```
            if(i == j) d[i][j] = 0;
            else if(w == 0) d[i][j] = inf;
            else d[i][j] = w;
        }
    }
    Floyd(n);
    for(int i = 1; i <= n; i++ ){
        for(int j = 1; j <= n; j++){
            if(d[i][j] == inf) cout << -1 << " ";
            else cout << d[i][j] << " ";
        }
        cout << endl;
    }
}
```

9. 奖学金(reward)

9.1 题目分析

利用拓扑排序，判断有向图是否存在环

如果存在环，输出 "impossible"，否则输出最长路径长度

通过计算每个顶点的入度，构建图的拓扑结构，并进行拓扑排序，更新每个顶点的最长路

9.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

const int maxn = 10000;
struct node{
    int num; // 入度
    int mon; // 最小时间
    node ():num(0),mon(100){};
}G[maxn];
struct Edge{
    int to,next;
}g[20000];
int head[maxn],tot=0;
void add(int u,int v){
    g[++tot].to = v;
    g[tot].next = head[u];
    head[u] = tot;
}
int cnt=0;
int q[maxn],l=0,r=0;
int main(){
    int n,m;
    cin >> n >> m;
    bool flag = true;int a,b;
    for (int i = 1; i <= m; i++){
        cin >> a >> b;
        add(b, a); // 建反向边
```

```
        G[a].num++; // 纪录每一个点的入度
    }
    int h = 1;
    // 找入度为 0 的点
    while (h <= n){
        if (G[h].num == 0){
            q[r] = h;
            ++r;
        }
        ++h;
    }
    int tmp;
    // 拓扑排序
    while (l<r){
        tmp = q[l];++l;
        for(int i = head[tmp]; i ; i = g[i].next){
            G[g[i].to].num--;
            if(G[g[i].to].mon <= G[tmp].mon)
               G[g[i].to].mon = G[tmp].mon + 1;
            if(G[g[i].to].num == 0){
                //入度为零则不可能再一次被更新了
                q[r] = g[i].to ; ++r;
            }
        }
    }
    int hh = 1;
    // 判断是否有环
    while (hh <= n){
        if (G[hh].num != 0){
            flag = false;
            break;
        }
        cnt += G[hh].mon;
        hh++;
    }
    if (!flag) cout << "impossible";
    else cout << cnt;
    return 0;
}
```