HW4实验报告

1. 涉及数据结构和相关背景

- 图的概念
  - 图的存储
    - 邻接矩阵
    - 邻接表
    - 逆邻接表
    - 链式前向星
- 图的遍历
  - `bfs`
    - 深度限制
  - `dfs`
    - 深度限制
- 最小生成树
  - `Kruskal`
    - 稀疏图
  - `Prim`
    - 稠密图
- 拓扑排序
  - `AOV`
  - `AOE`
- 最短路
  - `Dijkstra`
    - 堆优化`Dijkstra`
  - `Floyed`

2. 实验内容

2.1 图的遍历

2.1.1 问题描述

- 本题给定一个无向图，用邻接表作存储结构，用`dfs`和`bfs`找出图的所有连通子集。
  所有顶点用0到n-1表示，搜索时总是从编号最小的顶点出发。使用邻接矩阵存储，或者邻接表（使用邻接表时需要使用尾插法）

2.1.2 基本要求

- 输入
  - 第1行输入2个整数n m，分别表示顶点数和边数，空格分割后面m行，每行输入边的两个顶点编号，空格分割
- 输出
  - 第1行输出`dfs`的结果
  - 第2行输出`bfs`的结果
    连通子集输出格式为`{v11 v12 ...}{v21 v22 ..}...`连通子集内元素之间用空格分割，子集之间无空格,'{'和子集内第一个数字之间、'}'和子集内最后一个元素之间、子集之间均无空格
    对于20%的数据，有`0<n<=15`；对于40%的数据，有`0<n<=100`；对于100%的数据，有0<n<=10000；对于所有数据，`0.5n<=m<=1.5n`，保证输入数据无错

2.1.3 数据结构设计

- 利用邻接表进行图的存储，此处用线性表模拟链表进行存储

```
vector<int> linkt[10010];
```

- 访问数组记录该点是否被访问过

```
int vis[10010];
```

- 利用队列实现`bfs`

```
queue<int> q;
```

- 利用深度和广度求连通分量

2.1.4 功能说明

- 分别通过深度优先进行遍历和通过广度优先进行遍历

- 深度优先遍历

```
void dfs(int seq)
{
    /*
        递归实现dfs遍历图，利用访问数组记录是否被访问过
        seq : 编号
    */
```

```
    if (vis[seq]) return;

    vis[seq] = 1;

    temp.push_back(seq);

    for (int i = 0; i < linkt[seq].size(); i++)
    {
        dfs(linkt[seq][i]);
    }
}
```

- 广度优先遍历

```
void bfs(int n)
{
    /*
        广度优先遍历用队列实现遍历图
        n : 编号
    */
    q.push(n);
    temp.push_back(n);
    while (!q.empty())
    {
        int temp1 = q.front();
        vis[temp1] = 1;
        q.pop();
        for (int i = 0; i < linkt[temp1].size(); i++)
        {
            if (!vis[linkt[temp1][i]])
            {
                temp.push_back(linkt[temp1][i]);
                vis[linkt[temp1][i]] = 1;
                q.push(linkt[temp1][i]);
            }
        }
    }
}
```

- 主函数调用

```
int main()
{
    cin >> n >> m;

    int minl = 0;
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        linkt[x].push_back(y);
```

```
        linkt[y].push_back(x);


    }

/*输出部分*/
    for (int i = 0; i < n; i++)
    {
        /*从下一个没有访问的节点开始，即访问下一个连通分量*/
        if (!vis[i])
        {
            dfs(i);
            cout << "{";
            for (int j = 0; j < temp.size() - 1; j++)
            {
                cout << temp[j] << " ";
            }
            cout << temp[temp.size() - 1];
            cout << "}";
            temp.clear();
        }
    }
    cout << endl;
/*两次遍历之间需将访问数组清零*/
    memset(vis, 0, sizeof(vis));

    /*输出部分*/
    for (int i = 0; i < n; i++)
    {
        if (!vis[i])
        {
            bfs(i);
            cout << "{";
            for (int j = 0; j < temp.size() - 1; j++)
            {
                cout << temp[j] << " ";
            }
            cout << temp[temp.size() - 1];
            cout << "}";
            temp.clear();
        }
    }

    return 0;
}
```

2.1.5 调试分析

- 图的两种遍历方式是图数据结构的基础
- 利用两种遍历方式可以求出图中的各个联通分量，判断图的连通性

2.1.6 总结和体会

- 两种遍历方式是两种思想，`dfs`注重求出一个可行解，而`bfs`注重于求出所有解中具有特殊性质的解，具体解决问题时可以根据需要进行选择
- 两种遍历方式均可以进行遍历
- 从树结构进行过渡到图中，两种方式均适用（树是一种特殊的图结构）

2.2 小世界现象

2.2.1 问题描述

- 六度空间理论又称小世界理论。理论通俗地解释为：“你和世界上任何一个陌生人之间所间隔的人不会超过6个人，也就是说，最多通过五个人你就能够认识任何一个陌生人。
- 假如给你一个社交网络图，请你对每个节点计算符合“六度空间”理论的结点占结点总数的百分比。说明：由于浮点数精度不同导致结果有误差，请按float计算。

2.2.2 基本要求

- 输入：
  - 第1行给出两个正整数，分别表示社交网络图的结点数N（1<N≤2000 ，表示人数）、边数M（≤33×N，表示社交关系数）
  - 随后的M行对应M条边，每行给出一对正整数，分别是该条边直接连通的两个结点的编号（节点从1到N编号）
- 输出
  - 对每个结点输出与该结点距离不超过6的结点数占结点总数的百分比，精确到小数点后2位。每个结节点输出一行，格式为“结点编号:（空格）百分比%”

2.2.3 数据结构设计

- 用邻接表进行图的存储

```
vector<int> linkt[10010];
```

- 队列实现深度限制的广度优先遍历

```
 queue<int> q;
```

- 每一个节点所在层数形成的顺序队列

```
queue<int> tag;
```

- 思路：
  - 限制从"原点"开始，拓扑距离为6以内的所有点均进行该原点的计数
  - 涉及到拓扑距离限制，可以利用深度优先搜素或者广度优先搜索实现
  - 深度优先限制函数递归次数不超过6
  - 广度优先遍历限制层数不大于6即可

- 遍历过程中注意节点是否访问过的问题，防止重复计数

2.2.4 功能说明

- 具有深度限制的`bfs`

```
int bfs(int n)
{
    queue<int> q;

    queue<int> tag;
    int vis[2050] = { 0 };
    q.push(n);

    int dph = 0;
    int cnt = 0;
    tag.push(1);         //   记录当前节点是相对于原点节点是第几层

    while (!q.empty())
    {
        int temp1 = q.front();
        q.pop();
        vis[temp1] = 1;

        dph = tag.front();
        tag.pop();

        if (dph > 6) break; //达到限制深度就退出


        for (int i = 0; i < linkt[temp1].size(); i++)
        {
            if (!vis[linkt[temp1][i]])
            {
                cnt++;
                q.push(linkt[temp1][i]);
                vis[linkt[temp1][i]] = 1;
                tag.push(dph + 1);
            }
        }
    }
    return cnt + 1;      //最后需要加上
}
```

2.2.5 调试分析

- 对于广度优先搜索来说，最需要注意的问题是层数的检查，不是遍历到一个节点就算加一层，因因为广度优先会将每一层遍历完，每层都是一个编号，所以下方的写法是错误的
- 这种想法其实是`dfs`的方式，当`dfs`进行遍历的时候，进行递归确实是一个节点就是一层，二者不要弄混

```
int bfs(int n)
{
    queue<int> q;

    queue<int> tag;
    int vis[2050] = { 0 };
    q.push(n);

    int dph = 0;
    int cnt = 0;

    while (!q.empty())
    {
        int temp1 = q.front();
        q.pop();
        vis[temp1] = 1;

        dph = tag.front();

        if (dph > 6) break; //达到限制深度就退出

        for (int i = 0; i < linkt[temp1].size(); i++)
        {
            if (!vis[linkt[temp1][i]])
            {
                cnt++;
                q.push(linkt[temp1][i]);
                vis[linkt[temp1][i]] = 1;
                tag.push(dph + 1);
            }
        }
        dph++;      //错误写法，并不是遍历到一个就加一层
    }
    return cnt + 1;
}
```

2.2.6 总结和体会

- 进行有深度限制的遍历时一定要注意这个深度的测量方式，尤其对于广度优先遍历来说，深度测量更加困难一些，需要与遍历一同进行，
- 进行有关深度限制的遍历问题时可以优先选择`dfs`

2.3 村村通

2.3.1 问题描述

- N个村庄，从1到N编号，现在请你修建一些路使得任何两个村庄都彼此连通。我们称两个村庄A和B是连通的，当且仅当在A和B之间存在一条路，或者存在一个村庄C，使得A和C之间有一条路，并且C和B是连通的。
- 已知在一些村庄之间已经有了一些路，您的工作是再兴建一些路，使得所有的村庄都是连通的，并且新建的路的长度是最小的。

2.3.2 基本要求

- 输入：
  - 第一行包含一个整数n（3<=n<=100），表示村庄数目。
    接下来n行,每行n个非负整数，表示村庄i和村庄j之间的距离。距离值在[1,1000]之间。
  - 接着是一个整数m，后面给出m行，每行包含两个整数a,b,(1<=a<b),表示在村庄a和b之间已经修建了路。
- 输出：
  - 输出一行，仅有一个整数，表示为使所有的村庄连通，要新建公路的长度的最小值。

2.3.3 数据结构设计

- 邻接矩阵存储

```
int mat[105][105];
```

- 并查集判断`Kruskal`是否成环

```
int father[105];
```

- 边权排序记录

```
vector<Edge> edge;
```

- 边集

```
class Edge
{
public:
    Edge(int n1, int n2, int w)
    {
        this->node1 = n1;
        this->node2 = n2;
        this->weight = w;
        this->vis = 0;
    }

    int node1;                  //边的端点1
    int node2;                  //边的端点2
    int weight;                 //边的权重
    int vis;                    //访问标记
};
```

- 思路：
  - 已经修好的道路相当于在`Kruskal`中已经放置完的边（其实也相当于这些边的边权最小，已经加入图中且没有成环）
  - 排列其他剩下的边，将其按照边权由小到大进行排序，依次尝试加入图中，判断图中是否成环
  - 最后形成的树就是最小生成树

2.3.4 功能说明

- 初始化邻接矩阵、边排序

```
for (int i = 0; i < 105; i++)
{
    father[i] = i;                              //并查集初始化
}

cin >> n;
/*初始化邻接矩阵*/
for (int i = 1; i <= n; i++)
{
    for (int j = 1; j <= n; j++)
    {
        cin >> mat[i][j];
    }
}
/*初始化边集*/
for (int i = 1; i <= n; i++)
{
    for (int j = 1; j < i; j++)
    {
        Edge e = Edge(i, j, mat[i][j]);
        //e.vis = 1;
        edge.push_back(e);
    }
}

cin >> m;

sort(edge.begin(), edge.end(), mC());           //按照边权进行排序
```

- 初始化已经建好的边

```
for (int i = 1; i <= m; i++)
{
    int x, y;
    cin >> x >> y;
    for (int j = 0; j < edge.size(); j++)
    {
        if (edge[j].node1 == x && edge[j].node2 == y || edge[j].node1 == y
&& edge[j].node2 == x)
```

```
        {
            edge[j].vis = 1;            //访问过边标记
            break;
        }
    }
    unite(x, y);
}
int weight = 0;
int cnt = 0;
```

- `kruskal`将边依次放入图中并检验是否成环

```
for (int i = 0; i < edge.size() && cnt != n - 1; i++)
{
    if (!edge[i].vis)                                    //访问最开始没有建立道路的边
    {
        if (find(edge[i].node1) != find(edge[i].node2)) //并查集探测是否成环
        {
            unite(edge[i].node1, edge[i].node2);         //并查集合并
            weight += edge[i].weight;
            cnt++;                                       //记录遍历节点个数
        }
    }
}
```

- 并查集寻找根节点以及集合合并

```
int find(int n)
{
    /*递归找到根节点*/
    if (father[n] != n) return father[n] = find(father[n]);
    else return father[n];
}


void unite (int n1, int n2)
{
    int pRoot = find(n1);
    int qRoot = find(n2);

    if (pRoot == qRoot)
    {
        return;
    }
    father[pRoot] = qRoot;
}
```

2.3.5 调试分析

- 本体就是`Kruskal`的最小生成树建立方式，相当于有些边已经建立好，不用再进行排序
- 所以要注意在真正进行排序的时候需要去堆那些没有建好的边进行访问，此处需要开一个访问数组来进行记录

2.3.6 总结和体会

- `Kruskal`最小生成树的排序方式比较麻烦的一点是需要进行判环的操作，此处利用并查集进行根节点的查找。
- 需要注意并查集的寻找根节点的递归方式
- 并查集初始化需要进行自己指向自己

2.4 给定条件下构造矩阵

2.4.1 问题描述

- 给你一个 正 整数 k ，同时给你：
  - 一个大小为 n 的二维整数数组`rowConditions` ，其中`rowConditions[i] = [abovei, belowi]` 和
  - 一个大小为 m 的二维整数数组 `colConditions` ，其中 `colConditions[i] = [lefti, righti]`。

  两个数组里的整数都是 1 到 k 之间的数字。
- 你需要构造一个 k x k 的矩阵，1 到 k 每个数字需要 恰好出现一次 。剩余的数字都是 0 。矩阵还需要满足以下条件：
  - 对于所有 0 到 n - 1 之间的下标 i ，数字 `abovei` 所在的 行 必须在数字 `belowi` 所在行的上面。
  - 对于所有 0 到 m - 1 之间的下标 i ，数字`lefti` 所在的 列 必须在数字 `righti` 所在列的左边。
- 返回满足上述要求的矩阵，题目保证若矩阵存在则一定唯一；如果不存在答案，返回一个空的矩阵。

2.4.2 基本要求

- 输入：
  - 第一行包含3个整数k、n和m，接下来n行，每行两个整数`abovei`、`belowi`，描述`owConditions`数组，接下来m行，每行两个整数`lefti`、`righti`，描述`colConditions`数组
- 输出：
  - 如果可以构造矩阵，打印矩阵；否则输出-1，矩阵中每行元素使用空格分隔

2.4.3 数据结构设计

- 邻接表进行图的存储，每一个节点对应连接相邻顶点的链表、

```
vector<vector<int>> row;          //此处二维数组模拟邻接表存储，不是邻接矩阵
```

- 记录入度的数组

```
int inDegreeR[1050] = {0};
```

- 访问数组

```
int visR[1050] = { 0 };
```

- 位置二维数组

```
int pos[1050][2] = {0};     //[0]: 横坐标 [1]: 纵坐标
```

- 输出矩阵

```
int out[1050][1050] = {0};
```

- 思路：
  - 分别对行标和列标进行处理，分别建立有向图
  - 该有向图的拓扑排序顺序确定了相应数字的横（纵）坐标顺序
  - 该题目保证了排序顺序唯一，其实是确定了此拓扑图可以化为一个链表
  - 将横纵坐标分别进行拓扑排序之后，其在数组中对应的序号就是横纵坐标，将输出数组相应位置填入相应数字即可

2.4.4 功能说明

- 仅对横坐标进行说明，纵坐标处理方式完全相同
- 建立图

```
for (int i = 0; i < n; i++)
{
    int x, y;
    cin >> x >> y;
    row[x].push_back(y);
    inDegreeR[y]++;
}
```

- 进行拓扑排序

```
stack<int> RO;
vector<int> res;

/*入度为0点入栈*/
for (int i = 1; i <= k; i++)
{
    if (inDegreeR[i] == 0)
    {
        RO.push(i);
    }
}

/*进行拓扑排序*/
while (!RO.empty())
```

```
{
    int temp = R0.top();
    R0.pop();
    res.push_back(temp);                    //拓扑排序序列

    visR[temp] = 1;

    for (int i = 0; i < row[temp].size(); i++)
    {
        inDegreeR[row[temp][i]]--;
    }

    for (int i = 1; i <= k; i++)
    {
        if (inDegreeR[i] == 0 && !visR[i])
        {
            R0.push(i);
        }
    }
}
```

- 无法构建判断--拓扑排序成环

```
 if (res.size() != k)
 {
     cout << -1 << endl;
     return 0;
 }
```

- 纵坐标进行相同操作后进行输出数组填充

```
for (int i = 0; i < res.size(); i++)
{
    pos[res[i]][0] = i;
}
```

2.4.5 调试分析

- 主要是要考虑到坐标与拓扑排序之间的关系，进行拓扑排序，确定最终坐标

2.4.6 总结和体会

- 问题中涉及到先后顺序排列的问题，仅仅给出某两个元素之间的相对关系，可以利用有向图拓扑排序进行解决

2.5 必修课*

2.5.1 问题描述

- 某校的计算机系有 n 门必修课程。学生需要修完所有必修课程才能毕业。
- 每门课程都需要一定的学时去完成。有些课程有前置课程，需要先修完它们才能修这些课程；而其他课程没有。 不同于大多数学校，学生可以在任何时候进行选课，且同时选课的数量没有限制。
- 现在校方想要知道：
  - 从入学开始，每门课程最早可能完成的时间（单位：学时）；
  - 对每一门课程，若将该课程的学时增加1，是否会延长入学到毕业的最短时间。

2.5.2 基本要求

- 输入：
  - 第一行，一个正整数 n，代表课程的数量。接下来 n 行，每行若干个整数：
    - 第一个整数为 ti ，表示修完该课程所需的学时。
    - 第二个整数为 ci，表示该课程的前置课程数量。
    - 接下来 ci 个互不相同的整数，表示该课程的前置课程的编号。

    该校保证，每名入学的学生，一定能够在有限的时间内毕业。
- 输出：
  - 输出共 n 行，第 i 行包含两个整数：
    - 第一个整数表示编号为 i 的课程最早可能完成的时间。
    - 第二个整数表示，如果将该课程的学时增加1，入学到毕业的最短时间是否会增加。如果会增加则输出 11，否则输出 00。

    每行的两个整数以一个空格隔开。

2.5.3 数据结构设计

- 二维数组模拟邻接表

```
vector<int> mat[105];       //正向求ve邻接表
vector<int> mate[105];      //反向求vl邻接表（出度）
```

- 记录入度与出度

```
vector<int> Pind;
vector<int> Nind;
```

- 记录是否访问过

```
int visp[105];
int visn[105];
```

- `ve` `vl` `ae` `al`

```
vector<int> ve;
vector<int> vl;

vector<int> ae;
vector<int> al;
```

- 记录从上一状态到特定状态的时间

```
int timer[105];
```

2.5.4 功能说明

- 初始化

```
for (int i = 0; i < 200; i++)
{
    ve[i] = vl[i] = ae[i] = al[i] = Pind[i] = Nind[i] = 0;
}
```

- 初始化图，包括入度出度等

```
for (int i = 1; i <= n; i++)
{
    int t, pre, num;
    cin >> t >> num;
    timer[i] = t;
    chong.clear();
    for (int j = 1 ; j <= num; j++)
    {
        int found = 0;
        cin >> pre;
        for (int k = 0; k < chong.size(); k++)
        {
            if (chong[k] == pre) found = 1;
        }
        if (found) continue;
        chong.push_back(pre);
        mat[pre].push_back(i);
        Pind[i]++;
        mate[i].push_back(pre);
        Nind[pre]++;
    }

    if (num == 0)
    {
        mat[0].push_back(i);
        Pind[i]++;
        mate[i].push_back(0);
        Nind[0]++;
    }
}
```

- 源点与汇点的最早于最晚发生时间确定，前者均为0，后者依据计算出的最早发生时间确定
- 正向拓扑排序记录当前节点的所有上游节点的最早发生时间加上到当前节点状态时间的最大值，ve_{in}表示所有指向当前状态i的上一个状态的最早发生时间，$w_{in}[k]$表示某条入边的权值
- $$ve[i] = max\{ve_{in}[k] + w_{in}[k]\}$$

```
/*最早发生时间*/
ve[0] = 0;
stack<int> st;
st.push(0);

stack<int> out;

while (!st.empty())
{
    int vet = st.top();
    st.pop();

    if (!visp[vet])
    {
        visp[vet] = 1;
        int maxer = 0;

        for (int i = 0; i < mat[vet].size(); i++)
        {
            Pind[mat[vet][i]]--;
        }
        for (int i = 0; i < mate[vet].size(); i++)
        {
            maxer = max(maxer, timer[vet] + ve[mate[vet][i]]);
        }
        ve[vet] = maxer;

    }

    for (int i = 0; i <= n; i++)
    {
        if (Pind[i] == 0 && !visp[i]) st.push(i);
    }
}
```

- 最晚发生时间逆序拓扑排序，求下游`vl`与相应边权差的最小值
- $$vl[i] = min\{vl_{out}[k] - w_{out}[k]\}$$
- ```
  int start = 0;
  /*将最晚的最早开始时间状态入栈*/
  for (int i = 0; i <= n; i++)
  {
      if (Nind[i] == 0)
      {
          start = (ve[start] < ve[i]) ? i : start;
  ```
  ```

```
    }
}

st.push(start);

while (!st.empty())
{
    int vet = st.top();
    st.pop();

    if (!visn[vet])
    {
        visn[vet] = 1;
        int miner =0x3f3f3f3f;

        for (int i = 0; i < mate[vet].size(); i++)
        {
            Nind[mate[vet][i]]--;
        }
        for (int i = 0; i < mat[vet].size(); i++)
        {
            miner = min(miner, vl[mat[vet][i]] - timer[mat[vet][i]]);
        }

        vl[vet] = (miner == 0x3f3f3f3f)? ve[vet] : miner;

    }

    for (int i = 0; i <= n; i++)
    {
        if (Nind[i] == 0 && !visn[i] && check(i, Nindcopy)) st.push(i);
    }
}
```

- 输出判断是否为关键路径上的点，如果是则其对最终毕业时间有影响

```
for (int i = 1; i <= n; i++)
{
    cout << ve[i] << " " << ((ve[i] == vl[i]) ? 1 : 0) << endl;
}
```

2.5.5 调试分析

- 正常进行拓扑排序可以实现，上述方法无法适用于多终点的拓扑排序，其原因在于在求解逆向求解 `vl` 时将所有的汇点的 `vl` 与 `ve` 赋初值相等，导致在反向遍历时将较小的最早开始时间对应汇点也进行反向排序，出现错误

2.5.6 总结和体会

- 主要考察`AOE`的求解，正向求解事件最早开始时间，反向求解事件最晚发生事件，二者相等说明事件没有变化余地，则说明此路径对应为关键路径。

2.6 小马吃草

2.6.1 问题描述

- 假设无向图G上有N个点和M条边，点编号为1到N，第i条边长度为w_i，其中H个点上有可以食用的牧草。
- 另外有R匹小马，第j匹小马位于点start_j，需要先前往任意一个有牧草的点进食牧草，然后前往点end_j，
- 请你计算每一匹小马需要走过的最短距离。

2.6.2 基本要求

- 输入：
  - 第一行两个整数N、M，分别表示点和边的数量
  - 接下来M行，第i行包含三个整数`x_i`,`y_i`,`w_i`，表示从点`x_i`到点`y_i`有一条长度为`w_i`的边，保证`x_i≠y_i`
  - 接下来一行有两个整数H和R，分别表示有牧草的点数量和小马的数量
  - 接下来一行包含H个整数，为H个有牧草的点编号
  - 接下来R行，第j行包含两个整数start_j和end_j，表示第j匹小马起始位置和终点位置
  - 题目保证两个点之间一定是连通的，并且至少有一个点上有牧草
    - 对于20%的数据，`1<=N`、`R<=10`，`1 <=w_i<=10`；
    - 对于40%的数据，`1<=N`、`R<=100`，`1<=w_i<=100`；
    - 对于100%的数据，`1<=N`、`R<=1000`，`1<=w_i<=100000`；
    - 对于所有数据，`N-1<=M<=2N`，`1<=H<=N`。
- 输出：
  - 输出共R行，表示R匹小马需要走过的最短距离

2.6.3 数据结构设计

- 邻接表存储所有的边

```
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
```

- 每个点的最短距离是否确定

```
bool st[N];     // 存储每个点的最短距离是否已确定
```

- 记录每一处草地到图中其他点的最短路径

```
int way[1005][1005];
```

- 思路：
  - 从草地出发的单源最短路
  - `Dijkstra`最短路分别求草地到出发点和终点的最短距离，相加即为最短路径长度
  - 由于`Dijkstra`复杂度在$O(n^2)$，其中每次遍历更新到每一个点的最小值时可以用堆进行存储，将时间复杂度降低到$O(nlog(n))$
  - 遍历草地位置，算出草地到每一个点的最小值，最后进行查表相加即可

2.6.4 功能说明

- `Dijkstra`

```
void dijkstra(int sta)
{
    memset(dist, 0x3f, sizeof (dist));
    //memset(dist, 0, sizeof(st));
    dist[sta] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({ 0, sta });      // first存储距离，second存储节点编号

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second, distance = t.first;

        if (st[ver]) continue;

        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > distance + w[i])
            {
                dist[j] = distance + w[i];
                heap.push({ dist[j], j });
            }
        }
    }
}
```

- 添加图中边

```
void add(int a, int b, int weight)
{
    e[idx] = b;
    ne[idx] = h[a];
    w[idx] = weight;
    h[a] = idx++;

    e[idx] = a;
    ne[idx] = h[b];
    w[idx] = weight;
    h[b] = idx++;
}
```

- 遍历草地得到最短路径

```
int h, r;
cin >> h >> r;

vector<int> gr;

for (int i = 1; i <= h; i++)
{
    int grass;
    cin >> grass;
    gr.push_back(grass);
    dijkstra(grass);
    for (int i = 1; i <= n; i++)
    {
        way[grass][i] = dist[i];
    }
}
```

- 对于每一对起点与终点，遍历草地位置，选择距离和最小的草地位置，对应的距离和为最短路径长度

```
 for (int i = 1; i <= r; i++)
 {
     int start_j, end_j;
     cin >> start_j >> end_j;

     int temp = 0x3f3f3f3f;
     for (int j = 0; j < gr.size(); j++)
     {

         temp = (temp > way[gr[j]][start_j] + way[gr[j]][end_j]) ?
way[gr[j]][start_j] + way[gr[j]][end_j] : temp;

     }
```

```
    cout << temp << endl;
}
```

2.6.5 调试分析

- 因可能草地数量与图中节点数量均较大，所以$O(n^2)$的原版`Dijkstra`时间复杂度应该是会超过限制时间
- 用堆优化进行图遍历，节约时间到$O(nlogn)$

2.6.6 总结和体会

- 多次进行`Dijkstra`要注意其平方复杂度带来的时间消耗，其过程可以用堆的结构进行改善，将查找的时间复杂度由$O(n)$降低到$O(logn)$, 可以得到一定改善

3. 实验总结

- 图的遍历
  - `bfs`
    - 深度的计量-队列
  - `dfs`
    - 深度的计量-递归深度
- 最小生成树
  - `Kruskal`
    - 边权排序加入图中
    - 判断是否成环
  - `Prim`
    - 记录已经遍历过的区域节点以及该区域与外部连接的所有边
    - 选权值最小的边进行连接，更新遍历过的区域
- 最短路径
  - `Dijkstra`
    - 每一个节点更新到下一节点的长度
  - `Floyed`
    - 添加新节点观察是否新节点添加后对应的路径和更小
    - 若是则更新最短路
- 拓扑排序
  - `AOV`
    - 每次将入度为1的点入栈，更新下游节点入度
  - `AOE`
    - 求状态最早发生时间与最晚发生时间，相等的状态路径为关键路径