作业5

202208010512 计科2205 刘志垚

- 算法实现题6-2 最小权顶点覆盖问题
  - 1.题目描述
  - 2.思路
  - 3.代码
  - 4.结果
  - 5.分析
- 算法实现题6-6 n后问题(分支限界法)
  - 1.题目描述
  - 2.思路
  - 3.代码
  - 4.结果
  - 5.分析

算法实现题6-2 最小权顶点覆盖问题

1.题目描述

问题描述： 给定一个赋权无向图 $G=(V, E)$，每个顶点 $v\in V$ 都有权值 $w(v)$。如果 $U\subseteq V$，且对任意$(u, v)\in E$ 有 $u\in U$ 或 $v\in U$，就称 U 为图 G 的一个顶点覆盖。G 的最小权顶点覆盖是指 G 中所含顶点权之和最小的顶点覆盖。

算法设计： 对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最小权顶点覆盖。

数据输入： 由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 1, 2, …, n。第 2 行有 n 个正整数表示 n 个顶点的权。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示图 G 的一条边(u, v)。

结果输出： 将计算的最小权顶点覆盖的顶点权之和以及最优解输出到文件 output.txt。文件的第 1 行是最小权顶点覆盖顶点权之和；第 2 行是最优解 x_i（$1\leqslant i\leqslant n$），$x_i=0$ 表示顶点 i

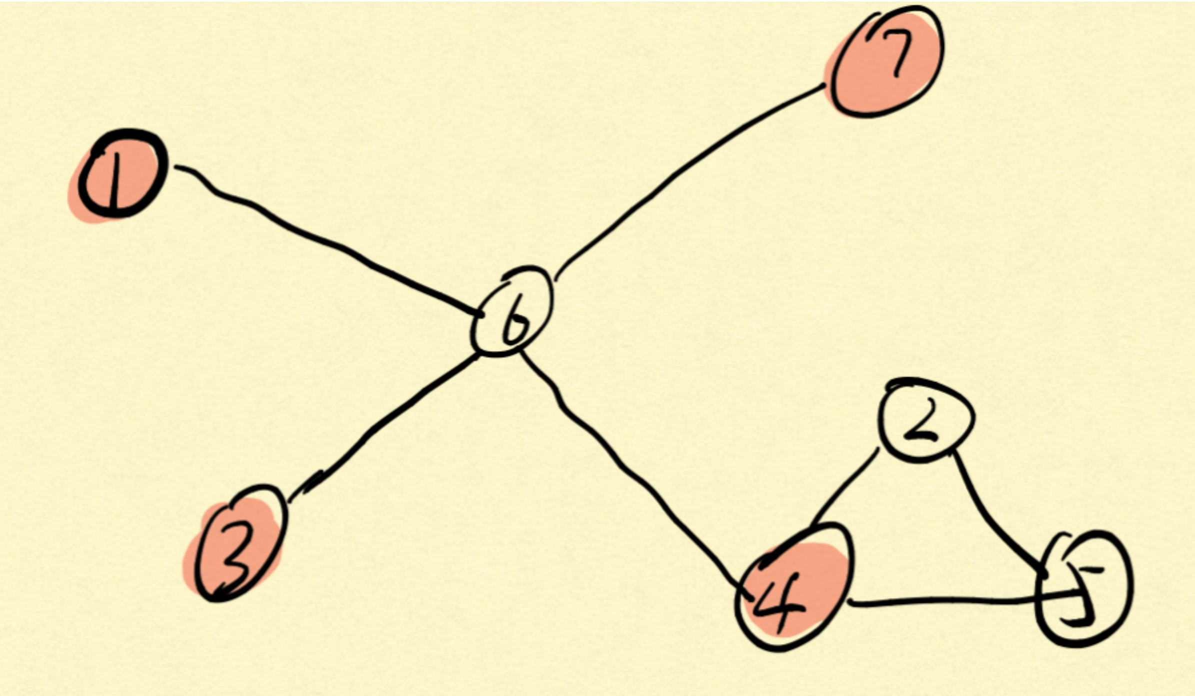
198

不在最小权顶点覆盖中，$x_i=1$ 表示顶点 i 在最小权顶点覆盖中。

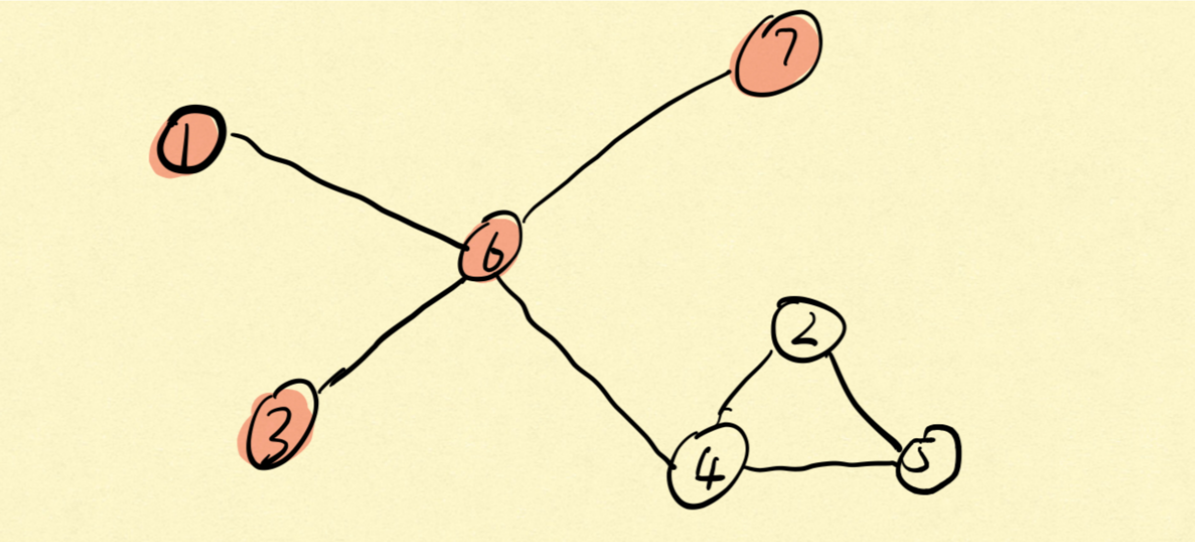
输入文件示例	输出文件示例
input.txt	output.txt
7 7	13
1 100 1 1 1 100 10	1 0 1 1 0 0 1
1 6	
2 4	
2 5	
3 6	
4 5	
4 6	
6 7	

2.思路

顶点覆盖呢：V 集合中的每个点，至少与一个 U 集合中的点直接相连。如图所示（红色点表示 U 集合中的点）：



可以看到 V 集合中的顶点2、5、6，都与至少一个 U 集合中的顶点直接相连。反而如果按照下图分配则不满足条件：



图中 V 集合的顶点2、5并没有 U 集合中的点与其直接相连，所以不是一种顶点覆盖。

那我们应该如何判断图是否被覆盖了呢？可以开辟一个数组 c，如果 c[j]==0，则表示 U 集合中没有任何一个顶点与其直接相连。

优先级的确定

在处理带权顶点的覆盖问题时，首先需要明确优先级的定义。因为我们目标是找到最小权覆盖，所以可以直接将权重作为优先级来建立最小堆（Min-Heap），形成一个优先队列。这样一来，可以确保我们每次取出的都是当前权重最小的顶点，从而有助于构建最小覆盖。

界限函数的应用

在我们的算法中，界限函数并不能有效地约束右孩子的路径。这是因为在某些情况下，即使当前节点未加入到集合U中，选择右孩子路径的点权有可能更小，但并不保证这条路径能够覆盖所有必要的边。因此，必须把右孩子也加入到优先队列中，即使是经过这种不确定的选择。这种处理方式确保了我们考虑所有可能的覆盖方案。

将活节点加入队列

当将“活节点”加入到优先队列时，需要对几个关键要素进行更新：

1. **优先级更新**：更新该节点在队列中的优先级，依据当前的权重和覆盖状态。
2. **结果向量更新**：维护一个结果向量（Result Vector），它记录了当前已经选取的节点，以便后续的覆盖判断。
3. **Cover数组更新**：更新cover数组，以反映该节点是否已经覆盖了某些边。这一数组将有助于判断当前状态下哪些边仍需被覆盖。

需要确保在每次加入新节点后，优先队列能够正确地调整结构，以保证下次抽取的仍是具有最小权重的节点。

3.代码

```
#include <iostream>
#include <queue>
using namespace std;
class HeapNode
{
    friend class VC;//求解最小权覆盖问题的类,融合了所有函数和所需的参数
    public:
        operator ()(int x,int y) const{return x < y;}//定义优先级
    private:
        int i,cn,*x,*c;//i表示结点序号,cn表示当前权重,x表示结果数组,c数组表示此时是否有一点
i属于U,且i与j相连,如果有,则c[j]!=0
};
//解最小权顶点覆盖大类
class VC
{
    friend MinCover(int **,int [],int);
    private:
        void BBVC();
        bool cover(HeapNode E);//判断图是否已经被全部覆盖了
        void AddLiveNode(priority_queue<HeapNode> &H,HeapNode E,int cn,int i,bool
ch);
        int **a,n,*w,*bestx,bestn;//邻接矩阵,节点数目,每个点的权重,结果向量,最优解
};
void VC::BBVC()
{
    priority_queue<HeapNode> H(100000);
    HeapNode E//扩展结点
    E.x = new int [n+1];//开辟结果向量
    E.c = new int [n+1];//开辟一数组,用于判断图是否被完全覆盖
    for(int j = 1;j <= n;j++)
    {
        E.x[j] = E.c[j] = 0;
    }
    int i = 1,cn = 0;//初始化当前点权总和为0
    while(true)
```

```
    {
        if(i > n)
        {
            if(cover(E))
            {
                for(int j = 1;j <= n;j++)
                    bestx[j]=E.x[j];
                bestn = cn;
                break;
            }
        }
        else
        {
            if(!cover(E))//如果当前没有完全覆盖,就将这个点加入到U集合中
                AddLiveNode(H,E,cn,i,1);
            AddLiveNode(H,E,cv,i,0);
        }
        if(H.size()==0)
            break;
        E = H.top();
        H.pop();
        cn = E.cn;
        i = E.i + 1;
    }
}
//判断图是否完全覆盖
bool VC::cover(HeapNode E)
{
    for(int j = 1;j <= n;j++)
    {
        if(E.x[j]==0 && E.c[j]==0)//如果此时j结点既不是U中的点,而且也没有U中的点与其相连,
则至少这个点未被覆盖
        {
            return false;
        }
    }
    return true;
}
void VC::AddLiveNode(priority_queue<HeapNode> &H,HeapNode E,int cn,int i,bool ch)
{
    HeapNode N;//创建一个新的堆结点
    N.x = new int [n+1];
    N.c = new int [n+1];
    for(int j = 1;j <= n;j++)
    {
        N.x[j] = E.x[j];
        N.c[j] = E.c[j];
    }
    N.x[i] = ch;
    if(ch)
    {
        N.cn = cn + w[i];//此时i要加入集合U,所以其权重应该加上cn
        for(int j = 1;j <= n;j++)
        {
            if(a[i][j])
                N.c[j]++;//表明此时对于结点j来说,有一节点i属于U与其连接,表明这个点被覆盖了
```

```
        }
    }
    else
        N.cn = cn;
    N.i = i;
    H.push(N);
}
//MinCover完成最小覆盖计算
int MinCover(int **a,int v[],int n)//v表示的是结点权重数组
{
    VC Y;
    Y.w = new int [n+1];
    for(int j = 1;j <= n;j++)
        Y.w[j] = v[j];
    Y.a = a;
    Y.n = n;
    Y.bestx = v;
    Y.BBVC();
    return Y.bestn;
}
//主函数
int main()
{
    int n,e,u,v;//结点数,边数,u,v为结点编号
    cin>>n>>e;
    int a[n+1][n+1];
    for(int i = 0;i <= n;i++)
    {
        for(int j = 0;j <= n;j++)
        {
            a[i][j] = 0;//初始化为0
        }
    }
    p = new int [n+1];//定义结果向量
    for(int i = 1;i <= e;i++)
    {
        cin>>u>>v;
        a[u][v] = 1;
        a[v][u] = 1;
    }
    cout<<MinCover(a,p,n)<<endl;
    for(int i = 1;i<=n;i++)
        cout<<p[i]<<" ";
    cout<<endl;
    return 0;
}
```

4.结果

```
7 7
1 100 1 1 1 100 10
1 6
2 4
2 5
3 6
4 5
4 6
6 7
13
1 0 1 0 1 0 1
```

5.分析

在分析BBVC函数的时间复杂度时，判断是否覆盖和活节点入队各需 $O(n)$ 时间，因此BBVC的整体时间复杂度为 $O(n^3)$。具体而言，while循环的时间复杂度为 $O(n)$，判断覆盖和入队操作也分别为 $O(n)$，所以整体为$O(n^3)$。在主函数中，MinCover的时间复杂度与BBVC相同，同时考虑输入邻接矩阵的边，最终时间复杂度为：

$$O(n^3 + e)$$

算法实现题6-6 n后问题(分支限界法)

1.题目描述

设计一个解n后问题的队列式分支限界法，计算在 $n \times n$ 个方格上放置彼此不受攻击的 n 个皇后的一个放置方案。

```
input
5
output
1 3 5 2 4
```

2.思路

1. 定义一个结构体node，表示棋盘上的每一个可能的位置，以及记录了当前状态的一些信息，如列、左右对角线等的占用情况。
2. 使用优先队列priority_queue来存储搜索过程中的状态，按照结构体中的x值进行排序。这里的x表示当前放置的皇后所在的行数。
3. 在主循环中，初始化棋盘的初始状态，将第一行的每一个位置作为起点，生成相应的初始状态，并加入优先队列中。
4. 进入主循环，每次从优先队列中取出一个状态，判断是否达到了目标状态（即放置了所有皇后），如果是则输出解，并结束程序（因为只需要找到一个可行解即可）。
5. 如果当前状态不是目标状态，继续在下一行尝试放置皇后。遍历每一列，对于每一个可行的位置，生成新的状态并加入优先队列中。
6. 在生成新状态时，进行剪枝操作，检查当前位置是否与之前的皇后冲突，如果冲突则跳过该位置。
7. 重复以上步骤，直到找到一个解或者队列为空。由于采用优先队列，搜索时会先尝试最有希望的位置，加速找到解的过程。

3.代码

```
#include <bits/stdc++.h>
using namespace std;
#define N 100
int n;
struct node
{
    int vis[N] = {0}, col[N] = {0}, lr[N] = {0}, rl[N] = {0};
    int x, y;
    node(int a, int b) : x(a), y(b) {}
    bool operator<(const node &a) const
    {
        return x < a.x;
    }
};
priority_queue<node> q;
int main()
{
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        node temp = node(0, i);
        temp.vis[0] = i + 1;
        temp.col[i] = 1;
        temp.rl[temp.x + temp.y] = 1;
        temp.lr[50 + temp.x - temp.y] = 1;
        q.push(temp);
    }
    while (!q.empty())
    {
        node temp = q.top();
        q.pop();
        if (temp.x == n - 1)
        {
            for (int i = 0; i < n; i++)
            {
                cout << temp.vis[i] << " ";
            }
            cout << endl;
            break; // 只需要给出一个答案即可
        }
        if (temp.x < n - 1)
        {
            for (int i = 0; i < n; i++)
            {
                node next = node(temp.x + 1, i);
                if (temp.col[next.y] || temp.lr[50 + next.x - next.y] ||
temp.rl[next.x + next.y])
                { // 剪枝
                    continue;
                }
                for (int i = 0; i < N; i++)
```

```
                {
                    next.lr[i] = temp.lr[i];
                    next.rl[i] = temp.rl[i];
                    next.col[i] = temp.col[i];
                }
                next.col[next.y] = 1;
                next.lr[50 + next.x - next.y] = 1;
                next.rl[next.x + next.y] = 1;
                for (int i = 0; i < next.x; i++)
                {
                    next.vis[i] = temp.vis[i];
                }
                next.vis[next.x] = i + 1;
                q.push(next);
            }
        }
    }
    return 0;
}
```

4.结果

n=10

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

n=15

```
15
1 3 5 2 10 12 14 4 13 9 6 15 7 11 8
```

5.分析

N-Queens 问题的时间复杂度在最坏情况下为 $O(N^N)$，但考虑剪枝后，实际复杂度通常为 $O(N!)$，因为剪枝策略显著减少了无效状态的搜索空间。同时，空间复杂度也大约为 $O(N!)$，主要由存储状态和优先队列的大小所决定。整体上，该算法通过剪枝和优先队列提高了效率，特别是在实际应用中，能够有效找到解。