暨南大学本科实验报告专用纸

课程名称 算法分析与设计实验 成绩评定

实验项目名称 最小权顶点覆盖问题 指导教师 李展

实验项目编号 实验十二 实验项目类型 综合性 实验地点

学生姓名 张印祺 学号 2018051948

学院 信息科学技术 系 计算机科学 专业 网络工程

实验时间 2020 年 6 月 2 日

一、问题描述

给定一个赋权无向图 G=（V，W），每个顶点v ∈ V都有一个权值 w(v)。如果U ⊆ V，且对任意(u，v)∈ E有 u∈ U或 v∈ 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。

对于给定的无向图 G，计算 G 的最小权顶点覆盖。

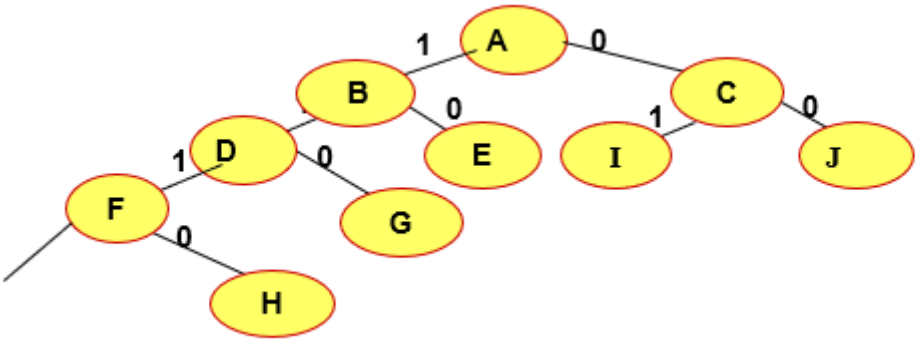
二、算法思路

0-1 背包问题的思想，先将所有的顶点进行排序，排序优先级为（顶点边数/顶点权重）,这样能优先选出边数大而权值小的顶点。

将第一个顶点入队，如果队列不为空，执行以下操作：

取出队列的第一个元素，判断其子状态是否能满足限制，如果不满足，则丢弃此结点，如果满足，将 1-0 两种情况进队。

剪枝条件：剩余的所有结点的边数之和小于需要覆盖的边数。例如，某个状态结点中，已经覆盖了两条边，还需要再覆盖 3 条边，但是某个状态的子状态结点能覆盖的边数总和为 2，那么这种状态不给予展开。



三、算法流程

由于本题过于繁琐，我使用语言描述我的算法（分支界限法）。

1、将原图数据构造成一个解空间树的节点，利用定界策略判断是否有解，如果无解直接退出，如果有可能有解则插入到优先队列中；

2、若优先队列不为空，那么便从优先队列中取出第一个可行的节点，进入步骤 3，如果优先队列为空则退出；

3、判断当前节点是否满足解的条件，如果满足便输出解退出，若不满足便进入 4；

4、检查当前节点是否可以扩展，不能扩展的话便进入 2 继续循环，如果能扩展的话则扩展，然后验证扩展到左右节点是否有解，将有解的扩展节点插入到优先队列中，然后进入 2 继续循环。

贪心法：

将可行的结点都是按照还需要覆盖的剩余边数的降序排列，即，每次选择的节点都是可行节点中还需要覆盖的边数最小的那个节点，因为它最接近结果了。

四、测试结果

```
EdgeNum = 7
VertexNum = 7
Weight = (1, 100, 1, 1, 1, 100, 10)
Edge:
    1 6, 2 4, 2 5, 3 6, 4 5, 4 6, 6 7
Result:
14
1 0 1 1 1 0 1
```

```
EdgeNum = 7
VertexNum = 7
Weight = (1, 2, 1, 1, 1, 1, 1)
Edge:
    1 2, 3 2, 4 2, 5 2, 6 2, 7 2
Result:
2
0 1 0 0 0 0 0
```

五、实验总结

我对这道题的理解还不够深入，主要原因是对限界分支的方法不够熟悉。

我的算法中，每个结点都记录了一个状态，每次判断需要进行 O(n)次运算，最坏的情况下有2^n个结点，因此算法的时间复杂度 T(N) = O(n2^n)。

其次，对于这道题的优化方法是这样的：

2	3
3	3
1	2
5	2
6	2
4	1
7	1

1、界的选择。在一个确定的无向图 G 中，每个顶点的边即确定了，那么对于该无向图中 k 个顶点能够覆盖的最多的边数 e 也就可以确定了！只要对顶点按照边的数目降序排列，然后选择前 k 个顶点，将它们的边数相加即能得到一个边数上界！因为这 k 个顶点相互之间可能有边存在也可能没有，所以这是个上界，而且有可能达到。以图 G 为例，各个顶点的边数统计，并采用降序排列的结果如左图所示。

2、假设取 k=3 个点，那么有 Up(e)=(3+3+2)=8＞7 条边（7 为图 G 的总边数），也就是说，如果从图 G 中取 3 个点，要覆盖 8 条边是有可能的。但是，如果取 k=2 个点，那么有 Up(e)=(3+3)=6＜7 条边，说明从图 G 中取 2 个点，是不可能覆盖 G 中的全部 7 条边的！基于这个上界，可以在分支树中扩展出来的节点进行验证，已知它还可以选择的顶点数目以及还需要覆盖的边的条数，加上顶点的状态（下面会分析说明）即可判断当前节点是否存在解！如果不存在即可进行剪枝了。

3、顶点的状态。该策略中顶点有三种状态，分别为已经选择了的状态 S1，不选择的状态 S2，可以选择的状态 S3。其中，不选择的状态 S2 对应解空间树中的右节点，不选择该节点，然后设置该节点为不选择状态 S2。这点很重要，因为有了这个状态，可以使得上界的判断更为精确，因为只能从剩余顶点集中选择那些状态 S3 的顶点，状态 S1 和 S2 都不行，那么上界便会更小，也就更加精确，从而利于剪枝。

综上所述，如果对于优先队列使用最小堆实现(O(nlgn))，解空间树的深度最多为顶点数目 n，每层都要进行分支定界，所以每层的时间复杂度为 O(nlgn)，所以算法总的时间复杂度为O(n^{2}lgn)。

六、源代码

```
package Experiment;
import java.util.*;
public class vertexCovery {
    Vertex[] nodes;
    int[][] edge;{
        edge = new int[][]{
            {1, 6}, {4, 2}, {2, 5}, {3, 6},
            {4, 5}, {4, 6}, {6, 7}
        };
    }
    int[] weight;{
        weight = new int[] {
            1, 100, 1, 1, 1, 100, 10
        };
    }
    boolean[] visited;{
        visited = new boolean[edge.length];
        Arrays.fill(visited, true);
    }
    vertexCovery(){
        int verNum = weight.length;

        nodes = new Vertex[verNum];
        for(int i = 0; i < verNum; i ++)
            nodes[i] = new Vertex(i, weight[i]);

        for(int i = 0; i < edge.length; i ++) {
            int idx1 = edge[i][0];
            int idx2 = edge[i][1];
            nodes[idx1 - 1].addEdge();
            nodes[idx2 - 1].addEdge();
        }

        Arrays.sort(nodes);
    }
```

```
    Queue<State> queue;
    public void solution() {
        queue = new LinkedList<>();
        queue.add(new State(visited, edge.length));
        queue.add(null);
        for(int i = 1, totE = edge.length * 2; i <
weight.length;) {
            State state = queue.remove();
            if(state == null) {
                i ++; totE -= nodes[i].edgeNum;
                continue;
            }
            if(state.leftEdge == 0){
                for(Integer idx : state.record)
                    System.out.print(idx);
                return;
            }
            if(state.leftEdge < totE)
                queue.add(new State(state, 0, 0));
            checkState(state, nodes[i], i);
            queue.add(new State(state, 1, nodes[i].weight));
        }
    }

    public void checkState(State state, Vertex ver, int i) {
        if(state.record.size() == 0) return;
        int idx = nodes[i].idx;
        for(int k = 0; k < edge.length; k ++) {
            if(edge[k][0] == idx + 1 || edge[k][1] == idx + 1)
                if(state.visR[idx]) {
                    state.visR[idx] = false;
                    state.leftEdge --;
                }
        }

    }

    public static void main(String[] args) {
        vertexCovery vc = new vertexCovery();

        for(Vertex v : vc.nodes)
            v.printNode();
```

```
        vc.solution();
    }
}

class Vertex implements Comparable<Vertex>{
    int idx, weight, edgeNum = 0;
    Vertex(int i, int w){
        idx = i;
        weight = w;
    }
    public void addEdge() {
        edgeNum++;
    }
    public void printNode() {
        System.out.println("Vertex" + idx + ", "
                    + weight + " " + edgeNum);
    }
    @Override
    public int compareTo(Vertex v) {
        double w = (double)(this.edgeNum) / this.weight;
        double nW = (double)(v.edgeNum) / v.weight;
        if(w > nW) return -1;
        else return 1;
    }
}

class State{
    int leftEdge, totWei;
    boolean[] visR;
    List<Integer> record;
    State(State s, int choice, int w){
        visR = Arrays.copyOf(s.visR, s.visR.length);
        record = new ArrayList<>(s.record);
        if(choice != -1) record.add(choice);
        else if(choice == 1) {
            totWei = s.totWei + w;
        }
        else {
            totWei = s.totWei;
        }
        leftEdge = s.leftEdge;

    }
    State(boolean[] visited, int le){
```

```
            visR = Arrays.copyOf(visited, visited.length);
            record = new ArrayList<>();
            leftEdge = le;
            totWei = 0;
        }
    }
```