分支限界法的核心思想

分支限界法在回溯的基础上引入了上下界的计算，能通过更加精细的剪枝来提高搜索效率，适用于优化问题。

1. 问题划分 (branch)
2. 界限计算 (bound)
3. 剪枝 (prune)
4. 选择和更新 (select and update)

问题 6-2

6-2 最小权顶点覆盖问题。

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。

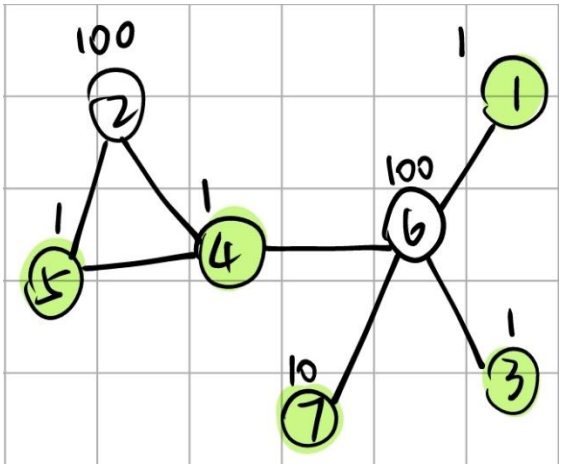
算法设计：对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最小权顶点覆盖。

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 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	

（输出不正确debug好久，后来发现示例输出有误）



优先队列式分支限界法

思路：

1. 状态空间树：子集树每个结点对应一个顶点覆盖状态
2. 分支：BranchBound对每个顶点选择加入或不加入
    - 加入则该顶点覆盖了与之相连的所有边
    - 不加入则需要保证所有与之相连的边都被其他顶点覆盖
3. 优先队列：队列中的结点根据其下界值排序，最小的下界优先扩展
    - push和pop操作保持最小堆性质
4. 下界：calc_bound计算，考虑已覆盖和未覆盖的边。未覆盖的边选择权值较小的顶点
5. 剪枝：如果当前结点的下界已经大于已知最优解bestw，则不再扩展该节点
6. 检查：check检查可行解，即所有边是否被覆盖

伪代码：

核心模块

Function calc_bound(node)

bound ← node.weight // 初始化 bound 为当前的权值

covered ← array of size (n+1) initialized to 0 // 创建标记数组，用于标记已经覆盖的顶点

// 标记已覆盖的顶点

for i from 1 to node.level do

```
        if node.x[i] = 1 then
            for j from 1 to n do
                if G[i][j] = 1 then
                    covered[j] ← 1   // 标记顶点 j 已被覆盖
            end for
            covered[i] ← 1   // 标记顶点 i 自身被覆盖
        end if
    end for
    // 计算下界：遍历尚未覆盖的边，选择权值较小的顶点
    for i from 1 to node.level do
        for j from 1 to n do
            if G[i][j] = 1 and covered[i] = 0 and covered[j] = 0 then
                bound ← bound + min(w[i], w[j])   // 选择权值较小的顶点
            end if
        end for
    end for
    free(covered)
    return bound
End Function

Function BranchBound()
root ← new Node   // 创建根节点
    root.x ← array of size (n+1) initialized to 0   // 解向量初始化为 0
    root.level ← 0   // 层次为 0
```

```
root.weight ← 0   // 权值为 0
root.bound ← calc_bound(root)   // 计算根节点的下界
push(root)   // 将根节点推入优先队列
while pq_size > 0 do
    cur ← pop()   // 从优先队列中弹出一个节点
    // 剪枝：如果当前节点的下界大于等于最优解，剪枝
    if cur.bound >= bestw then
        free(cur)
        continue
    end if
    // 叶节点检查：当层次达到 n 时，检查当前解是否是最优解
    if cur.level = n then
        if check(cur) and cur.weight < bestw then
            bestw ← cur.weight   // 更新最优解的权值
            bestx ← cur.x   // 更新最优解的解向量
        end if
        free(cur)
        continue
    end if
    // 不选当前顶点：生成左子节点
    left ← copy(cur)   // 复制当前节点
    left.level ← cur.level + 1   // 层次加 1
    left.bound ← calc_bound(left)   // 计算左子节点的下界
```

```
        if left.bound < bestw then
            push(left)   // 如果左子节点的下界小于最优解，推入优先队列
        else
            free(left)
        end if
        // 选择当前顶点：生成右子节点
        right ← copy(cur)   // 复制当前节点
        right.level ← cur.level + 1   // 层次加 1
        right.x[right.level] ← 1   // 选择当前顶点
        right.weight ← cur.weight + w[right.level]   // 更新权值
        right.bound ← calc_bound(right)   // 计算右子节点的下界
        if right.bound < bestw and right.weight < bestw then
            push(right)   // 如果右子节点的下界和权值都小于最优解，推入优先队列
        else
            free(right)
        end if
        free(cur)
    end while
```

End Function

时间复杂度：$O(2^n \cdot n^2)$ （最坏）

节点扩展数：最坏$O(2^n)$；

优先队列：每次push或pop一个结点到优先队列的时间复杂度是 $O(\log k)$，其中 k是优先队列中的结点数。最坏情况下，优先队列中可能会有 $O(2^n)$ 个结点

每个结点处理时间：计算下界最坏$O(n^2)$，检查合法性$O(n^2)$;

空间复杂度：$O(2^n \cdot n + n^2)$ （多数情况下$2^n >> n^2$，主要由$2^n \cdot n$决定）

存储矩阵：存储G需要$O(n^2)$

优先队列：$O(2^n \cdot n)$;

每个节点的存储：O(n);

辅助数组：O(n);

问题 6-7

6-7 布线问题。

问题描述：假设要将一组元件安装在一块线路板上，为此需要设计一个线路板布线方案。各元件的连线数由连线矩阵 conn 给出。元件 i 和元件 j 之间的连线数为 conn(i, j)。如果将元件 i 安装在线路板上位置 r 处，而将元件 j 安装在线路板上位置 s 处，则元件 i 和元件 j 之间的距离为 dist(r, s)。确定了所给的 n 个元件的安装位置，就确定了一个布线方案。与此布线方案相应的布线成本为 $\text{dist}(r,s) \times \sum_{1 \leqslant i \leqslant j \leqslant n} \text{conn}(i, j)$。试设计一个优先队列式分支限界法，找出所给 n 个元件的布线成本最小的布线方案。

算法设计：对于给定的 n 个元件，设计一个优先队列式分支限界法，计算最佳布线方案，使布线费用达到最小。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$1 \leqslant n \leqslant 20$）。接下来的 $n-1$ 行，每行 $n-i$ 个数，表示元件 i 和元件 j 之间连线数（$1 \leqslant i < j \leqslant 20$）。

结果输出：将计算的最小布线费用以及相应的最佳布线方案输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3	10
2 3	1 3 2
3	

优先队列式分支限界法

思路：

1. 状态空间树：排列树每个结点表示一种可能的布线方案
2. 分支：BranchBound将一个元件放置在某个未使用的位置上

- 每个结点从其父结点继承了所有已布置的元件，并尝试将未布置的元件逐一安装到一个位置上
- 每个位置只能被一个元件使用，所以在分支时需要考虑哪些位置尚未被占用

3. 优先队列：队列中的节点根据其下界值排序，最小的下界优先扩展
    - push和pop操作保持最小堆性质
4. 下界：calc_bound通过计算当前节点的部分布线方案及其剩余未布置元件的最优布置来获得当前结点可能的最小成本
5. 剪枝：如果当前结点的下界已经大于已知最优解bestc，则不再扩展该节点

伪代码：

核心模块

```
Function calc_bound(node)

    bound = node.cost

    used_pos = array of size n, initialized to 0

    // 标记已使用的元件位置

    for i from 0 to n-1 do

        used_pos[i] = node.used[i]

    end for

    // 对未安排的元件计算最小可能成本

    for i from node.level to n-1 do

        min_cost = infinity   // 当前元件的最小成本初始化为无穷大

        for j from 0 to n-1 do

            if used_pos[j] == 0 then   // 如果位置j未被使用

                cost = 0

                // 计算与已安排元件的成本
```

```
                for k from 0 to node.level-1 do
                    cost = cost + conn[k][i] * dist[node.x[k]][j]
                end for
                // 更新
                if cost < min_cost then
                    min_cost = cost
                end if
            end if
        end for
        bound = bound + min_cost    // 增加最小成本
    end for
    return bound
End Function
Function BranchBound()
root = new Node
    for i from 0 to n-1 do
        root.x[i] = -1
        root.used[i] = 0
    end for
    root.level = 0
    root.cost = 0
    root.bound = calc_bound(root)
    push(root)
```

```
while pq_size > 0 do
    cur = pop()
    // 剪枝：如果当前节点的下界大于等于最佳解，则跳过
    if cur.bound >= bestc then
        free(cur)
        continue
    end if
    // 找到一个完整解
    if cur.level == n then
        if cur.cost < bestc then
            bestc = cur.cost    // 更新最优解
            for i from 0 to n-1 do
                bestx[i] = cur.x[i]    // 更新最优布线方案
            end for
        end if
        free(cur)
        continue
    end if
    // 扩展子节点
    for i from 0 to n-1 do
        if cur.used[i] == 0 then    // 如果位置i未被使用
            next = copy(cur)    // 复制当前节点到下一个节点
```

```
                // 安排当前元件到位置i
                next.x[next.level] = i
                next.used[i] = 1
                next.level = next.level + 1
                // 计算新增成本
                new_cost = 0
                for j from 0 to next.level-2 do
                    new_cost = new_cost + conn[j][next.level-1] * dist[next.x[j]][i]
                end for
                next.cost = next.cost + new_cost



                next.bound = calc_bound(next)
                // 如果下界小于最佳解，则加入优先队列
                if next.bound < bestc then
                    push(next)
                else
                    free(next)
                end if
            end if
        end for
        free(cur)
    end while
```

End Function

时间复杂度：$O(n! \cdot n^2 \cdot \log n!)$ （最坏）

状态空间：树深度为n，每层n个可能位置，状态空间的规模 n!（剪枝会显著减少）

结点：calc_bound遍历所有已布置的元件并计算剩余布置的最小成本，最坏$O(n^2)$；计算当前节点的布线成本$O(n^2)$

优先队列：每次push或pop一个结点到优先队列的时间复杂度是 $O(\log k)$，其中 k 是优先队列中的结点数。最坏情况下，优先队列中可能会有 $O(n!)$ 个结点

空间复杂度：$O(n \cdot n! + n^2)$ （多数情况下$n! >> n^2$，主要由$n \cdot n!$决定）

存储矩阵：存储conn和dist需要$O(n^2)$；

优先队列：$O(n \cdot n!)$；

每个节点的存储：O(n);