习题7

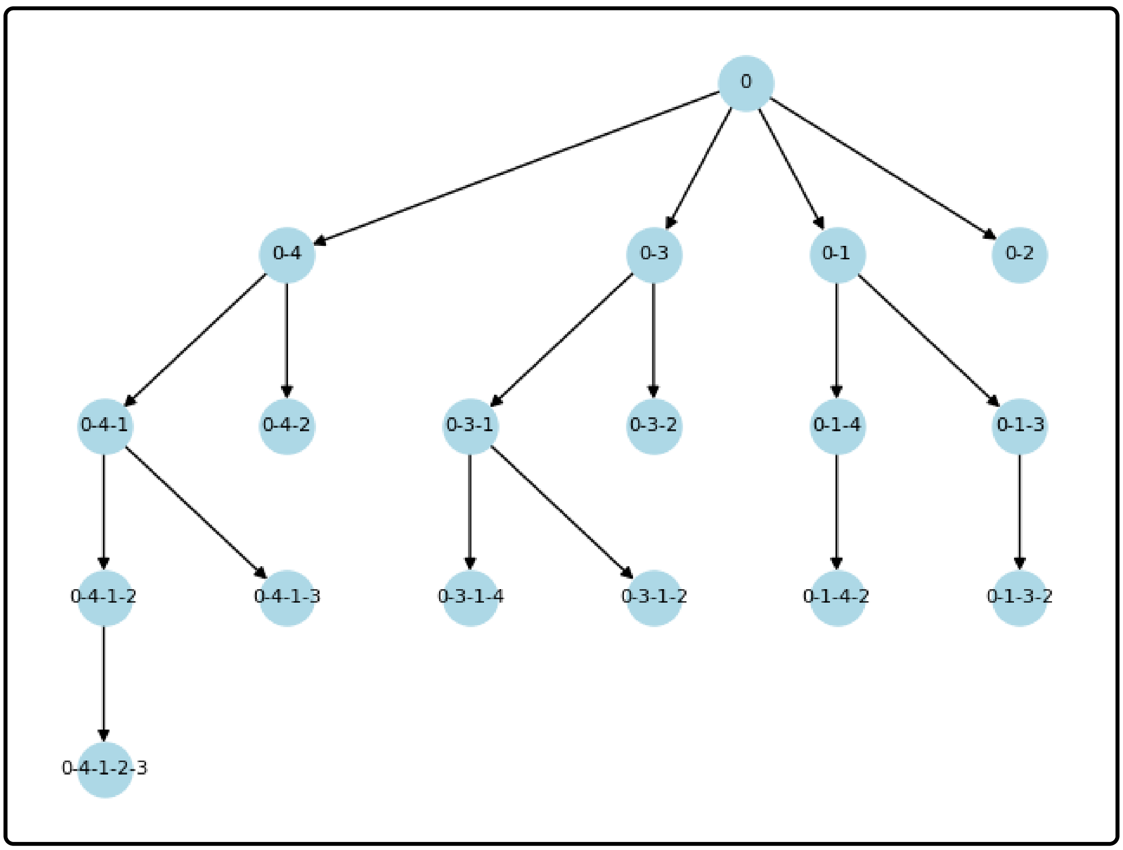
202328015926048-丁力

1.假设对称旅行商问题的邻接矩阵如图 1 所示, 试用优先队列式分枝限界算法给出最短环游。画出状态空间树的搜索图, 并说明搜索过程。

$$\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ & \infty & 16 & 4 & 2 \\ & & \infty & 6 & 7 \\ & & & \infty & 12 \\ & & & & \infty \end{pmatrix}$$

空间树搜索图如下， 其中索引从0开始，所以最短环游为：

```
1->5->2->3->4
```



1. **初始阶段**：先建立一个优先级队列，然后创建一个以第一个节点为起点的树的根节点，计算其成本，并把该节点放入优先级队列中。队列按照节点的成本从低到高排序。
2. **开始搜索阶段**：从优先级队列中取出一个代价最低的节点作为当前节点。如果这个节点的级别等于节点总数（n），那么我们就找到了一条完整的路径，并打印出路径和总成本。如果这个节点的级别小于节点总数，那么对于每一个尚未访问的节点，我们生成一个新节点并加入优先级队列中。
3. **生成新节点阶段**：首先，我们复制当前节点的代价矩阵，并从中删除所有指向当前节点的路径和尚未访问的节点。然后，我们执行矩阵的行和列表减。接着，我们计算新节点的成本，包括从当前节点到新节点的距离、约简矩阵的成本以及当前节点的成本。最后，我们把新节点添加到优先级队列中。
4. **结束阶段**：重复步骤2和3，直到优先级队列为空或者我们已经找到了一条包括所有节点的路径。如果我们找到了这样的路径，我们将结束算法并输出结果。否则，我们将输出没有找到解。

---

2. 试写出 0/1 背包问题的优先队列式分枝限界算法程序, 并找一个物品个数至少是 16 的例子检验程序的运行情况。

算法思路

1. **状态表示**：每个状态表示当前考虑到的物品和当前的总价值和总重量。
2. **队列选择**：使用优先队列（基于价值的最大堆）来保证每次都先考虑价值最大的状态。
3. **分枝**：对于每个物品，我们可以选择“放入背包”或者“不放入背包”两种状态。
4. **限界**：使用贪心策略来估算剩余物品的最大价值上界，如果当前状态加上上界仍然无法超过当前最优解，则舍弃这个状态。

代码示例（Python）

```
import queue

class State:
    def __init__(self, level, profit, weight, bound):
        self.level = level
        self.profit = profit
        self.weight = weight
        self.bound = bound

    def __lt__(self, other):
        return self.bound > other.bound

def bound(state, W, n, items):
    if state.weight >= W:
        return 0
    profit_bound = state.profit
    j = state.level + 1
    totweight = state.weight
    while j < n and totweight + items[j][1] <= W:
        totweight += items[j][1]
        profit_bound += items[j][0]
        j += 1
    if j < n:
        profit_bound += (W - totweight) * (items[j][0] / items[j][1])
```

```python
    return profit_bound

def knapsack(W, items):
    items = sorted(items, key=lambda x: x[0]/x[1], reverse=True)
    Q = queue.PriorityQueue()
    n = len(items)
    maxProfit = 0
    Q.put(State(-1, 0, 0, 0))
    while not Q.empty():
        state = Q.get()
        if state.level == n-1:
            continue
        nextLevel = state.level + 1
        nextWeight = state.weight + items[nextLevel][1]
        nextProfit = state.profit + items[nextLevel][0]
        if nextWeight <= W and nextProfit > maxProfit:
            maxProfit = nextProfit
        bound_value = bound(state, W, n, items)
        if bound_value > maxProfit:
            Q.put(State(nextLevel, state.profit, state.weight, bound_value))
            Q.put(State(nextLevel, nextProfit, nextWeight, bound_value))
    return maxProfit

# 测试数据
W = 50
items = [(60, 10), (100, 20), (120, 30), (70, 10), (50, 5), (30, 5), (40, 10), (35, 15), (25, 10), (55, 20), (65, 25), (75, 30), (85, 35), (95, 40), (105, 45), (115, 50)]
print(knapsack(W, items))
```

输出结果：

```
310
```

运行该代码后，得到的最大价值是 310。这意味着在不超过背包容量 50 的情况下，通过合理选择这 16 个物品中的部分，可以达到的最大价值为 310。而相应的放置策略包括以下物品：

1. 价值 50，重量 5
2. 价值 70，重量 10
3. 价值 60，重量 10
4. 价值 30，重量 5
5. 价值 100，重量 20

3. 最佳调度问题: 假设有 n 个任务要由 k 个可并行工作的机器来完成, 完成任务 i 需要的时间为 t_i 。试设计一个分枝限界算法, 找出完成这 n 个任务的最佳调度, 使得完成全部任务的时间 (从机器开始加工任务到最后停机的时间) 最短。

策略1：

可以按照如下策略安排这几个任务：

- 将任务所需的时间由大到小排序
- 计算n个任务所需要的总时间平均到k个机器上的时间
- 将大于平均时间的任务各分配一个机器，找到最大完成时间
- 将其他的任务顺序安排在一台机器上，如果超出最大时间，则交给下一个机器，直到所有任务都不能在小于最大完成时间的情况下安排。
- 安排下一台机器直到所有任务完成。
- 所有可能安排某些任务找不到小于最大完成时间，那么重新扫描各个机器再加上任务时间最小，按照此方法完成所有任务。

策略2：

- 将任务所需时间又打大小排序
- 将n个任务中的前k个任务分配给当前的k个机器，然后将k+1任务分配给最早完成已分配任务的机器，一次进行，最后找出这些机器最终分配任务所需时间最长的， 以此时间作为分支界限函数，如果一个扩展节点所需的时间大于这个最优解，那么删掉此节点为根的树，否则更新最优值。