实验二 回溯法VS分支定界法

姓名： 张椿旭 **学号：**2017303024 **班级：**14011702

一．问题描述

回溯法可以处理货郎担问题，分支定界法也可以处理货郎担问题，回溯法和分支定界法哪个算法处理货郎担问题效率更高呢?

实现回溯法、分支定界法，以及不同的界值函数（课上讲过的或者自己新设计的），通过随机产生10个不同规模的算例（城市数量分别为10，20，40，80，100，120，160，180，200，500，或者其它规模），比较回溯法和分支定界法在相同界值函数下的执行效率。另外，分别比较回溯法和分支定界法在不同界值函数下的执行效率。（说明：本次实验课内容较多，如果不能完成，最后一次试验课的2个课时，可以继续）

二．实验目的及要求

1.理解回溯法和分支定界算法，并实现回溯法和分支定界法算法；

2.掌握不同的界值设计方法，通过随机案例，比较界值设计的好坏；

3.比较回溯法和分支定界法的执行效率，分析回溯法和分支定界法的优缺点。

三．实验分析

Travelling salesman problem(TSP) 是这样一个问题：给定一系列城市和每对城市之间的距离，求解访问每一座城市一次并回到起始城市的最短回路。它是组合优化中的一个NP困难问题，在运筹学和理论计算机科学中非常重要。该问题被划分为NP完全问题。已知TSP算法最坏情况下的时间复杂度随着城市数量的增多而成超多项式（可能是指数）级别增长。

1. 回溯法实现 回溯法是暴力搜寻法中的一种。对于某些计算问题而言，回溯法是一种可以找出所有（或一部分）解的一般性算法，尤其适用于约束满足问题（在解决约束满足问题时，我们逐步构造更多的候选解，并且在确定某一部分候选解不可能补全成正确解之后放弃继续搜索这个部分候选解本身及其可以拓展出的子候选解，转而测试其他的部分候选解）。 具体而言, 在本题中我们可以定义在一次回溯迭代中状态为走过的且仅一次城市数组. 那么在走过的城市数量为总数量时, 这就是一个解. 最后对所有解进行一次统计最小花费, 即可得到最优解.
2. 分支界限法实现 分支定界（BB）是一种针对离散和组合优化问题以及数学优化的算法设计范例。分支定界算法包括通过状态空间搜索对候选解进行系统枚举的方法：候选解的集合被认为是形成一个以树的整个集合为根的有根树。 该算法探索该树的分支，这些分支代表解决方案集的子集。 在枚举分支的候选解之前，先根据最佳解上的估计上限和下限检查分支，如果该分支不能产生比算法到目前为止找到的最佳解更好的解，则将其丢弃。 在TSP问题中, 我们选择使用优先队列进行计算下限和比较上限的广度搜索, 来解决. 最终找到一个解或者最优解. 在分支界限法中最重要的是设计计算上下界的界限函数. 对于TSP问题, 我采用课上的一种设计(简称为`TwoMinCost`), 同时自己设计了一个`基于剩余最小出边边权的下界界限函数`, (简称`RestMinCost`)
    1. `TwoMinCost`设计

➢**部分解的下界：U表示已选节点的集合，假设U中有k个元素<r_1,r_2,...r_k>**

$$db = (2\sum_{i=1}^{k-1} c[r_i][r_{i+1}] + \sum_{r_i \in U} r_i\text{行不在路径上的最小元素} + \sum_{r_j \notin U} r_j\text{行最小的两个元素}) / 2$$

2. `RestMinCost`设计

   这是一个贪心的设计, 在每次迭代中的状态节点包含了一个`restMinCost`和`lowerBound`属性.

   其中`restMinCost`指剩余最小花费, 是由未走过的节点的出边边权之和确定.

   而`lowerBound`指下界, 由贪心思想设计为`cost + restMinCost`.

四．算法伪代码或流程图

1. 回溯法

```
function backtrack(int depth, int cost, int path):
    if depth > n then
        cost += graph[path.last][1]
        if cost < minCost then
            minCost = cost
            answerPath = path + [1]
        end
    else:
        for j = i -> n:
            if check(i): continue
            if cutting(i): continue

            backtrack(depth+1, cost + graph[path.last][j])
    end
end
```

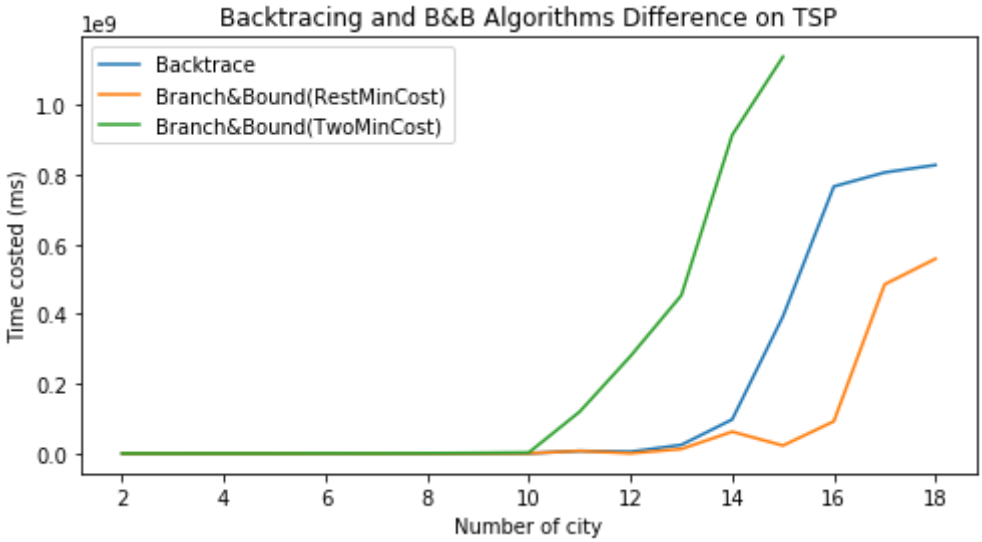
2. 分支界限法

```
function solve():
    answer = computeUpperBound()
    startNode = [1 .. n]
    que.add(startNode)

    while !que.isEmpty():
        node = que.poll()

        if node.size == n:
            int backCost = graph[node.state[n-1]][1]
            if (backCost == NO_EDGE) continue;
            ansVector = node.state.clone()
            answer = node.cost + backCost
            break
        end
```

```
        for i = node.size -> n:
            if graph[state[node.size-1]][node.state[i]] != NO_EDGE:
                next = Node.nextFrom(node, i)

                if (next.lowerBound >= answer) continue
                que.add(next)
            end
        end
    end
return answer

function nextFrom(Node from, int nextPosition):
    newState = from.state
    newState[nextPosition] = from.state[from.size]
    newState[from.size] = toElement

    lowerBound = from.cost + outer.graph[fromElement][toElement] +
from.restMinCost - outer.minOutEdgeCost[fromElement]

return new Node(newState, lowerBound)
```

五．算法时间复杂性分析

1. 理论分析
    1. 回溯法 暴力求出所有可行解, 最后统计最优解. 由于给定的图不一定是完全图, 所以有些边不存在, 那么分析时间复杂度比较困难. 但是我们假设图为完全图, 那么依据解空间树可知最坏情况下复杂度为O(n!). 若带剪枝, 那么复杂度会更优秀, 但难以定量计算复杂度.
    2. 分支界限法 若不带分析上下解的界限函数和剪枝的话, 显然复杂度为O((n-1)!log(n-1)!). 因为存在每次迭代都做一次优先队列的维护, 而找到一个解或最优解, 对于解空间树只需要计算n-1层即可. 若带分析上下解的界限函数和剪枝函数, 那么时间花费将大大减少, 但同样难以定量计算复杂度.
2. 实验分析 将回溯法和分支界限法用不同的界限函数和问题规模, 对计算所用时间花费做度量. 问题求解所需的临界矩阵是随机生成的, 其中边不存在是有1/3的概率. 为了防止特殊的实验对结果的影响, 我对每个规模生成了25次随机数据, 对每种算法进行测试, 得到如下实验结果:



六．问题思考与总结

1. 对于`n = 18`的问题规模, 基于分支界限的最快解法已经需要耗费分钟级别的时间. 用更大的规模进行计算是不明智的. 难以想象题目要求中对于`n = 500`规模需要几年时间来计算? 使用这种基于搜索的算法处理TSP问题是显然低效的, 同时求得的较优解是否称得上`优`? 事实上我认为对于大于20的规模, 不应该用基于搜索的算法, 而是尝试最优化算法(常见的退火, 粒子群, nn等).
2. 实验中给出的代码模板实在难以维护, 甚至难以观看. 我用了一些方法将代码重写了. 总结下来, 有关算法的代码不是不需要维护和二次编写的, 算法也是工程, 不能把代码写得太简陋.

七．实验中出现的问题及总结

1. 实验中给出的代码模板实在难以维护, 甚至难以观看. 我用了一些方法将代码重写了. 总结下来, 有关算法的代码不是不需要维护和二次编写的, 算法也是工程, 不能把代码写得太简陋. 有不少编码原则没有体现和实施, 希望代码能够更加易于维护, 或者不要提供样例代码. 这样学生也不至于处于一直迷惑的状态和不断契合原代码的多余工作.