算法课堂实验报告（五）——python回溯法与分支限界法（旅行商TSP问题）

原创

Campsisgrandiflora 于 2018-08-27 16:03:55 发布 7255 收藏 51
分类专栏： 数据结构与算法分析 python 文章标签： python 算法

本文链接：https://blog.csdn.net/Campsisgrandiflora/article/details/82114198


数据结构与算法分析 同时被 2 个专栏收录
5 篇文章 0 订阅
订阅专栏


python
12 篇文章 0 订阅
订阅专栏

python实现回溯法与分支限界

一、开发环境

开发工具：jupyter notebook 并使用vscode，cmd命令行工具协助编程测试算法,并使用codeblocks辅助编写C++程序
编程语言：python3.6

二、实验目标

1. 请用回溯法求对称的旅行商问题（TSP问题）

2. 请用分支限界法求对称的旅行商问题（TSP问题）

三、实验内容

旅行商问题的简单说明：

旅行商问题（TSP问题）是一个经典的组合优化问题。经典TSP问题可以描述为：一个商品推销员要去若干个城市推销商品，该推销员从一个城市出发，需要经过所有城市后，回到出发地。应如何选择行进路线，以使总的行程最短

从图论角度看：

该问题实质是在一个带权完全无向图中，找一个权值最小的Hamilton回路。由于该问题的可行解是所有顶点的全排列，随着顶点数的增加，会产生组合爆炸，它是一个NP完全问题。

TSP的数学模型为：

$$\min F=\sum_{i\neq j} d_{ij}\times x_{ij}$$

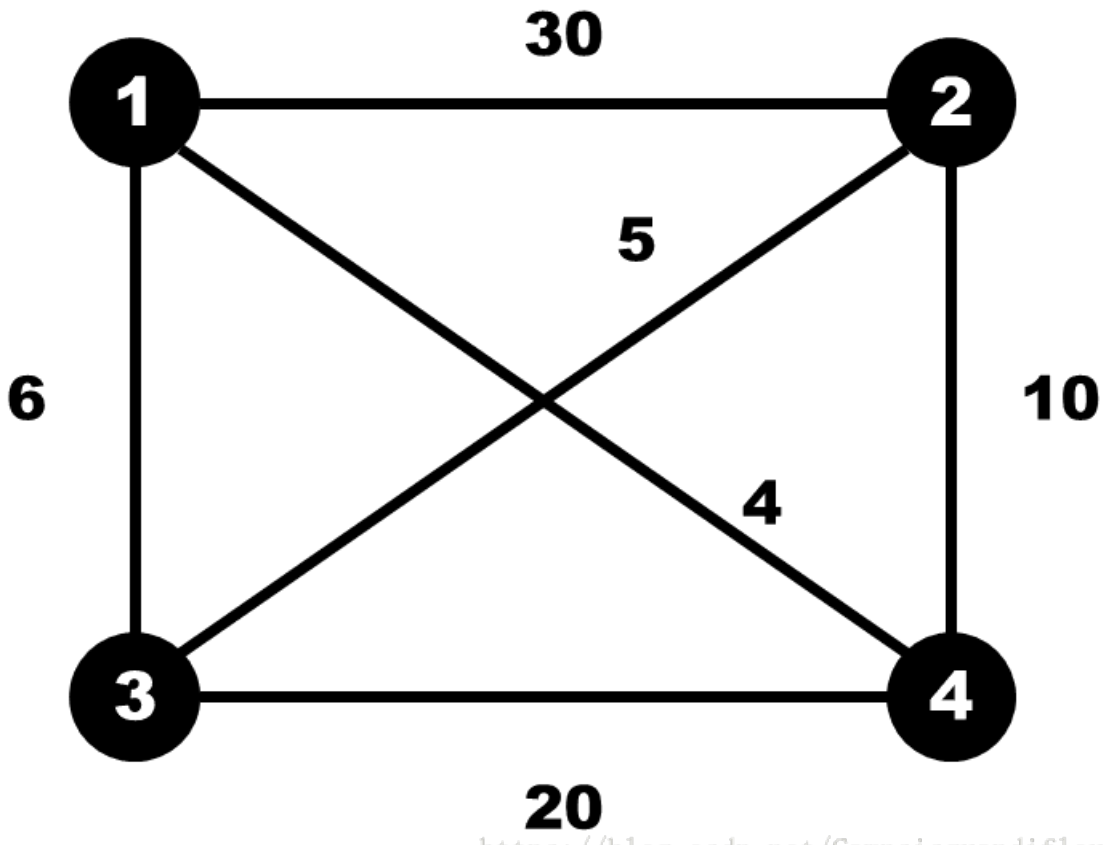
$$st.x_{ij}=\begin{cases}1, \text{边 } e_{ij} \text{ 在最优路径上}\\ 0, \text{边 } e_{ij} \text{ 不在最优路径上}\end{cases}$$

$$\sum_{i\neq j} x_{ij}=1, \quad i\in V$$

$$\sum_{i\neq j} x_{ij}=1, \quad j\in V$$

$$\sum_{i,j\in S} x_{ij}=|S|, \quad S \text{ 为 } G \text{ 的子图}$$

使用的测试数据：



使用字典的形式表示：

```
# 定义图的字典形式
G = {
    '1': {'2': 30, '3': 6, '4': 4},
    '2': {'1': 30, '3': 5, '4': 10},
    '3': {'1': 6, '2': 5, '4': 20},
    '4': {'1': 4, '2': 10, '3': 20}
}
```

使用数据的形式表示：

```
# 定义图的数组形式
graph = [
    [0, 30, 6, 4],
    [30, 0, 5, 10],
    [6, 5, 0 , 20],
    [4, 10, 20, 0]
]
```

（一）使用回溯法求解旅行商（TSP）问题

回溯法的概念：

回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

许多复杂的，规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。

在我看来，回溯法有点像图遍历中的深度优先算法，不过多了剪枝这一过程，当发现现有的解的值大于最优解的时候，就不再深入遍历下去，算法的性能取决于剪枝的多少。

下面给出回溯法的代码：

```
# 回溯法求解旅行商问题
import math
n = 4
x = [0, 1, 2, 3]
# 定义图的字典形式
G = {
    '1': {'2': 30, '3': 6, '4': 4},
    '2': {'1': 30, '3': 5, '4': 10},
    '3': {'1': 6, '2': 5, '4': 20},
    '4': {'1': 4, '2': 10, '3': 20}
}
# 定义图的数组形式
graph = [
    [0, 30, 6, 4],
    [30, 0, 5, 10],
    [6, 5, 0 , 20],
    [4, 10, 20, 0]
]

# bestcost = 1<<32 # 这里只要是一个很大数就行了 无穷其实也可以
bestcost = math.inf # 好吧 干脆就无穷好了
nowcost = 0    # 全局变量，现在的花费

def TSP(graph, n, s):
    global nowcost, bestcost
    if(s == n):
        if (graph[x[n-1]][x[0]] != 0 and (nowcost +graph[x[n-1]][x[0]]<bestcost)):
            print('best way:', x)
            bestcost = nowcost + graph[x[n-1]][x[0]]
            print('bestcost', bestcost)

    else:
        for i in range(s, n):

            # 如果下一节点不是自身 而且 求得的值小于目前的最佳值
            if (graph[x[i-1]][x[i]] != 0 and nowcost+graph[x[i-1]][x[i]] < bestcost):
                x[i], x[s] = x[s], x[i]    # 交换一下

                nowcost += graph[x[s - 1]][x[s]]    # 将花费加入
                TSP(graph, n, s+1)
                nowcost -= graph[x[s - 1]][x[s]]    # 回溯上去还需要减去

                x[i], x[s] = x[s], x[i]    # 别忘记交换回来

TSP(graph, n, 1)
```

运行的结果为：

```
best way: [0, 1, 2, 3]
bestcost 59
best way: [0, 2, 1, 3]
bestcost 25
```

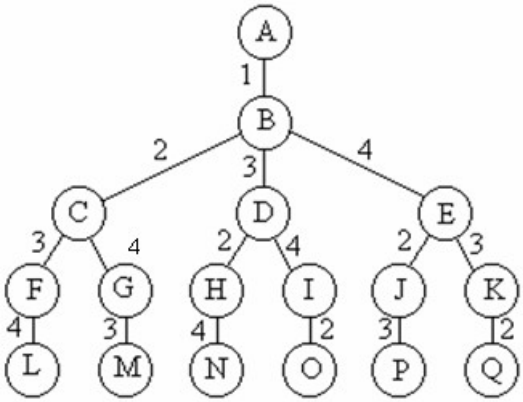

一共输出了两个结果，说明完整路径搜索了两次，其余情况下，均在搜索到中途的时候进行了剪枝操作。

（二）使用分支限界法求解旅行商（TSP）问题

分支限界法概念：

回溯法的求解目标是找出T中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

我觉得分支限界法像图的广度遍历，从上往下进行层级的遍历，一步一步直到最终找到解，不过，不同的地方在于，分支限界法采用了优先队列的方式进行了优化，首先找寻可能满足目标函数的解，再利用剪枝进行了优化处理



写算法的时候，利用上面这张图，进行了自定义节点类的构造和编写，并利用优先队列存储各个节点，代码量大大减少，不同过存在的问题是，如果不是一个完全图进行输入，结果会报错，没有做到很好的容差性与健壮性，同时新生成的对象太多了，会占用很大的空间，不过及时的删除节点将会解决这一问题。

分支限界法的代码如下：

```
# 分支限界法求解旅行商问题
import math
from heapq import *

n=4
x = [1, 2, 3, 4]
# 定义图的字典形式
G = {
    '1': {'2': 30, '3': 6, '4': 4},
    '2': {'1': 30, '3': 5, '4': 10},
    '3': {'1': 6, '2': 5, '4': 20},
    '4': {'1': 4, '2': 10, '3': 20}
}
# 定义图的数组形式
graph = [
    [0, 30, 6, 4],
    [30, 0, 5, 10],
    [6, 5, 0 , 20],
    [4, 10, 20, 0]
]

# bestcost = 1<<32 # 这里只要是一个很大数就行了 无穷其实也可以
bestcost = math.inf # 好吧 干脆就无穷好了
nowcost = 0    # 全局变量，现在的花费

# 设置节点类
class Node:
    # 构造函数，现在的花费，到目前的路径
    def __init__(self, w=math.inf, route=[], cost=0):
        self.weight = w
        self.route = route
        self.cost = cost
```

```
    # 重载比较，用于堆的排序
    def __lt__(self,other):
        return int(self.weight) < int(other.weight)

    # 打印
    def __str__(self):
        return "节点的权重为"+str(self.weight)+" 节点的路径为"+str(self.route)+" 花费为"+str(self.cost)


def BBTSP(graph, n, s):
    global bestcost, bestroute
    heap = []

    start = Node(route=[str(s)])

    heap.append(start)

    # 当堆中有数的时候，循环继续
    while heap:
        nownode = heappop(heap)    # 取出权重最大的那个数
        # 生成权重最大的那个数的下结点，并且把下结点加入堆中
        for e in [r for r in graph if r not in nownode.route]:
            node = Node(w=graph[nownode.route[-1]][e], route=nownode.route+[e], cost=nownode.cost+graph[now
            # 如果现在的值大于最优值，剪枝操作
            if node.cost >= bestcost:
                continue
            # 如果到了最后一个点，加上回去的路，并计算最小值
            if len(node.route)==4:
                wholecost = graph[node.route[-1]][s]+node.cost
                if wholecost < bestcost:
                    bestcost = wholecost
                    bestroute = node.route
                    print("最佳花费为:"+str(bestcost))
                    print("最佳路径为:"+str(bestroute))

            heap.append(node)


    return bestcost


BBTSP(G, n, '1')
```

实验结果的如下所示：

```
最佳花费为:59
最佳路径为:['1', '2', '3', '4']
最佳花费为:25
最佳路径为:['1', '4', '2', '3']
```

结果只是恰好与上面一样，但是其实里面内部的机制是完全不一样的，为什么使用优先队列进行优化还是会出现这样的结果呢？我自己分析了一下，发现优先队列使用的步骤只是找到三个节点，而从最后一个节点到第一个节点是没有考虑在内的，所以会出现前面最优而总体结果并不是最优的情况。