**《算法设计与分析》**

**课程实验报告**



**专业： 计算机科学与技术**

**班级： 2021211307**

**姓名： 陈朴炎**

**学号： 2021211138**

目录

[1 实验说明 4](#_Toc154745227)

[1.1 内容 4](#_Toc154745228)

[1.1.1书面作业 4](#_Toc154745229)

[1.1.2 编程作业 4](#_Toc154745230)

[1.2 要求 4](#_Toc154745231)

[1.3 起点说明 6](#_Toc154745232)

[2 书面作业——分支限界法最小化问题算法框架 6](#_Toc154745233)

[3 回溯法 8](#_Toc154745234)

[3.1 回溯法原理 8](#_Toc154745235)

[3.2 TSP回溯法过程 10](#_Toc154745236)

[3.3 TSP回溯法算法流程 11](#_Toc154745237)

[3.4 算法实现 12](#_Toc154745238)

[4 分支限界法 13](#_Toc154745239)

[4.1 TSP问题的上界 13](#_Toc154745240)

[4.1.1 求上界思路 13](#_Toc154745241)

[4.1.2 求上界算法实现 15](#_Toc154745242)

[4.2 TSP问题的下界 16](#_Toc154745243)

[4.2.1 计算部分解的目标值下界lb 16](#_Toc154745244)

[4.2.2 求下界算法实现 18](#_Toc154745245)

[4.3 算法思路及步骤 19](#_Toc154745246)

[4.3.1 算法思路 19](#_Toc154745247)

[4.3.2 算法步骤 20](#_Toc154745248)

[4.4 算法实现 21](#_Toc154745249)

[4.4.1 变量说明 21](#_Toc154745250)

[4.4.2 核心分支限界函数实现 22](#_Toc154745251)

[5 执行结果 23](#_Toc154745252)

[5.1 结果表格 23](#_Toc154745253)

[5.2 执行结果截图 27](#_Toc154745254)

[5.2.1 回溯法 27](#_Toc154745255)

[5.2.2分支限界法 28](#_Toc154745256)

[6 程序源代码 29](#_Toc154745257)

[6.1 回溯法 29](#_Toc154745258)

[6.2 分支限界法 34](#_Toc154745259)

# 1 实验说明

## 1.1 内容

### 1.1.1书面作业

参照讲义PPT中（p26-28）给出的面向最大化问题（如0-1背包问题）的分支限界法算法框架，设计面向最小化问题，e.g. 旅行商问题，的分支限界法算法框架.

将算法框架附在实验报告中

### 1.1.2 编程作业

采用回溯法、分支限界法，编程求解不同规模的旅行商问题TSP，并利用给定数据，验证算法正确性，对比算法的时间复杂性、空间复杂性

## 1.2 要求

参照教科书，编程实现回溯法、分支限界法，求解旅行商问题，并对比2个算法对同一规模TSP问题的运行时间和内存空间占用，对比两个算法的时间、空间复杂性

说明： 图中顶点数目为>22个基站时，可能导致：回溯法运行时间较长，或分支限界法占用内存空间过多，无法求出最终解，故不考虑顶点数目过多的

修改完善回溯法、分支限界法求解TSP问题的程序，统计记录

1.搜索过程中扫描过的搜索树结点总数L

2.程序运行时间T

针对图1、图2、图3、图4，输出采用回溯法、分支限界法得到的

1. 从起始城市出发的最短旅行路径

2. 路径总长度

3. 扫描过的搜索树结点总数L

4. 程序运行时间T

并将结果记录在下列表格中：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 问题 |  | 求解算法 | 最短回路 | 路径总长度  （单位：  m） | 搜索过的结点总数 | 程序运行时间  （单位：s） |
| 15个基站 |  | 回溯 |  |  |  |  |
|  | 分支限界 |  |  |  |  |
| 20个基站 |  | 回溯 |  |  |  |  |
|  | 分支限界 |  |  |  |  |
| 22、30个基站  （可选） |  | 回溯法 |  |  |  |  |

说明：

如果时间充裕 （不做硬性要求），试着针对n=22个基站组成的无向图，采用回溯法求解最短回路。n=22、30时，采用分支限界法求解时，可能需要搜索和记录的活结点过多，导致内存溢出。

注意：

对同一个问题，如n=15个基站组成的图，采用回溯法、分支限界法得到的最短路径回路应当是一样的，或至少最短回路的长度是一样的。2种方法搜索的节点数目、运行时间有可能不同

## 1.3 起点说明

15个基站的起点基站ID：567443，城市20

20个基站的起点基站ID：567443，城市20

22个基站的起点基站ID：567443，城市20

30个基站的起点基站ID：567443，城市20

# 2 书面作业——分支限界法最小化问题算法框架

步骤1. 选择初始解对应的根节点v0，根据限界函数bound，估计根节点的目标函数上下界bound(v0)，确定目标函数的界[down, up]

步骤2. 将活结点表ANT初始化为空

步骤3. 生成根节点v0的全部子结点-宽度优先；

对每个子结点v，执行以下操作：

3.1 估算v的目标函数值（下界）bound(v)

3.2 若bound(v) <= up, 将v加入ANT表

因为对于最小化问题，要求沿着v分支搜索到的完全解的目标值估计，必须小于现有的已知的最优目标函数的上界up。若下界估计值lb超出完整解的上界，则该部分解对应死结点，可以剪枝。

步骤4. 循环，直到某个叶节点的目标函数值在表ANT中最小

这一步是用来找到一个具有最小值的完全解

4.1 从ANT中选择（下界）bound(vi)值最小的结点vi，扩展其子结点。

// 可以使用优先队列，将队列中最小目标值bound（v）作为扩展结点

4.2 对结点vi的每个子结点c，执行下列操作

4.2.1 估算c的目标函数值bound(c)-下界

4.2.2 如果bound(c) <= up，将c加入ANT表

因为子结点c有可能产生更优的解

4.2.3 如果c是叶节点并且bound(c)在表ANT中最小，则将结点c对应的完全解输出，算法结束

4.2.4 如果c是叶节点但是bound(c)在表ANT中不是最小，则说明结点c对应了1给新找到的完全解，但该完全解的目标函数值与已经找到的、或之后可能找到的完全解相比，并非最优

(i)令up=value(c)，利用新找到的完全解的实际目标vlaue(c)/bound(c)，来更新问题解的上界value(c)。叶节点c对应完全解的实际路径长度

(ii)对表ANT中所有bound(vj) > up = bound(c)的结点vj，从ANT表中删除该结点。这一步是利用新找到的完全解目标函数bound(c)，进行剪枝：从ANT表中去掉那些目标函数下界值不可能小于结点c的bound(c)的结点vj，即去掉那些目标函数下界值大于当前新找到的完全解c的目标值bound(c)的结点。

剪枝条件可参考下图：

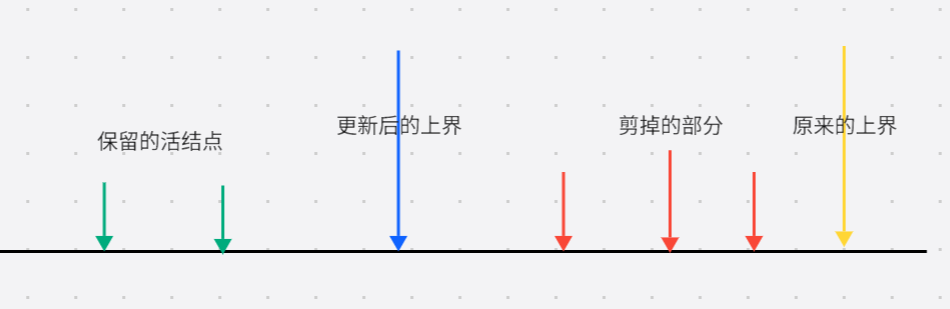


图2-1 剪枝条件示意图

# 3 回溯法

## 3.1 回溯法原理

1. 利用某种数据结构形式化表示问题及问题解

旅行商问题TSP

n个城市组成的带权无向图G=(V,E)，顶点V对应于城市，边E对应于城市间路径，要求找出一条旅行线路，每个城市只经历一次，且总路径长度最短

问题解/周游路径：n+1维向量，向量中元素为依次。经历的城市，例如<1,4,2,3,1>，或：n维向量<1,4,2,3>

2. 各种形式的解组成问题解空间

最优解，次优解/可行解，错误/不可行解，部分解

e.g. 最优解: <1,3,2,4,1>, <1,4,2, 3,1>，路径长度=25

次优解/可行解: <1,2,3,4,1>，路径长度=59

错误/不可行解：<1,2,4,2,1>, <1,3,3,4,1>,

部分解： <1,3,2,?,?>,

3. 问题解应满足的条件称为约束，包括

1）显约束：对解空间中分量xi的取值限定，e.g.

0-1背包为题xi只能取0、1

TSP中，解向量各分量只能为图中各个城市

2）隐约束：为满足问题的解而对不同分量之间施加的约束，e.g.

背包问题中，放入包中的物品总重量不能超过C

旅行商问题中，每个城市只能经过一次

4. 解空间中各种类型解根据相互间关系和解的构造顺序，组成**解空间树**

1个叶结点对应1个可行解、最优解、不可行解

非叶节点代表部分解

问题求解：解空间树搜索过程

从代表初始解（e.g. <?, ?, ?>）的根结点开始，向叶结点逐步搜索，扩展解空间树，搜索过程对应解的逐步、渐进构造过程

E.g. 1 3种物品的0-1背包问题，解表示<x1, x2, x3>, 解空间

{<?,?,?>, <0,0,0>, <0,1,0>, <0,0,1>, <1,0,0>, <0,1,1>, <1,0,1>, <1,1,0>, <1,1,1>, <0/1, ?, ?>, <0/1,0/1,?> }

根据背包容量和各个物体重量，判断上述各个解的性质：

最优、次优、不可行解、部分解

5. 针对组合优化问题，对解空间中的解引入定量指标，作为解搜索、优化依据

e.g. 1 0-1背包问题：放入背包中的物品价值最大化

e.g. 2 旅行商问题： 旅行回路总长最短

6. 解的构造过程

1）以深度优先的方式，从树根结点的空解<?,?,?>开始，依次扩展树结点，直到到达叶结点——搜索过程中动态产生解空间

深度优先目的：尽可能快地获得可行解

2）扩展过程中，碰到可行非叶结点（部分解），可进一步扩展

e.g. 结点C对应部分解<1,2,?,?,1>，可进一步扩展为:

F= <1,2,3,?,1>

G= <1,2,4,?,1>

3）碰到不可行非叶/叶结点（不可行（部分）解），需要回溯：

返回到上一层结点

e.g. 对C结点，下一步的扩展有4种可能选择：3、4、1、2，每种选择都可以继续扩展子树；但只有前2种选择是合理的，对后2种选择不再继续扩展，而是返回C结点。

4）为提高搜索效率，用剪枝函数（面向具体问题，关键！）避免无效搜索，即避免搜索不可行解对应的子树或结点

**7.** 问题求解过程体现为对解空间的带有回溯的深度优先树搜索

从算法实现角度，采用2种回溯控制策略

1. 递归回溯

2. 迭代回溯

## 3.2 TSP回溯法过程

对于下面给出的邻接矩阵：

1 2 3 4

1 99999 30 6 99999

2 99999 99999 5 10

3 6 99999 99999 20

4 4 10 99999 99999

1. 初始时，bestw=∞

2. 深度优先得到第一条路径，<1,2,3,4,1>

搜索该路径时，bestw不变。搜索完后，bestw变为59

3. 按照深度优先，搜索第二条路径<1,2,4,3,1>

搜索过程中，不断比较、判断部分路径的费用≥bestw=59，决定是否继续搜索下去。对部分路径<1,2,4,?> , 代价=30+10=40<bestw=59，可以搜索G下的分支。搜索完该分支后，路径总长度=30+10+20+6=66 ≥bestw=59。该路径不如以前找到的第一条路径，bestw不变

4. 该问题最优解: <1,3,2,4,1>, <1, 4,2,3,1>，对应的bestw=25

假设：搜索另一分支<1,3,4,?>, 当前路径对应的结点为**I**，长度=6+20=26 ≥bestw=25，结点I之下的路径被舍弃

## 3.3 TSP回溯法算法流程

设有以下变量：

n：总共的城市数目

bestx[1:n]：用来记录最佳路径。

cw：当前已经走过的部分路径的总长

x[1:n]：搜索过程中生成的部分路径

w：图的邻接矩阵

算法流程如下：

0. 对函数输入一个形参i，表示当前搜索树在第几层

1. 如果i==n，说明搜索到达叶节点，还没考虑的城市只有x[n]，因此只能选择最后一个城市x[n]

2. 如果 x[n-1]到x[n]有一条边，并且x[n]到起始城市也有一条边，并且cw+这两条边的长度小于bestw

3. 那就循环for(int j = 1; j <=n; j++)

bestx[j]=x[j]; 更新路径

4. bestw = cw + w(x[n-1], x[n])+w(x[n],1) 更新长度

5. else 如果没用搜索到叶节点，那就依次考察下一步x[i]的可能取值j

6. for j=i，j<=n，j++

7. 如果x[i-1]和x[j]有一条边，并且加上这条边后cw < bestw，那么说明成本小于最优回路长度，可以继续搜索

8. 将城市x[j]放到x[i]的位置上

9. cw += w(x[i-1], x[i]) 更新扩展后的路径代价

10. 调用本函数，输入层数为i+1的参数

11. cw -= w(x[i-1], x[i]) 搜索失败，回溯，回到原来的状态

12. 将调换过的i和j位置调换回来

## 3.4 算法实现

1. **void** backTrackTSP(**int** layer)
2. {
3. nodeNum+=1;
4. **if** (layer == n)
5. {
6. // cout << "layer == n" << endl;
7. **if** (!abs(w(x[n - 1], x[n]) - MAX\_DISTANCE) < 1e-4 && !abs(w(x[n], start) - MAX\_DISTANCE) < 1e-4 && cw + w(x[n - 1], x[n]) + w(x[n], start) < bestw)
8. {
9. // 找到更优的路径，更新
10. **for** (**int** j = 1; j <= n; j++)
11. {
12. bestx[j] = x[j];
13. }
14. bestw = cw + w(x[n - 1], x[n]) + w(x[n], start);
15. cout << "搜索到其中一个解 : " << bestw << endl;
16. }
17. }
18. **else**
19. {
20. **for** (**int** j = layer; j <= n; j++)
21. {
22. **if** (!abs(w(x[layer - 1], x[j]) - MAX\_DISTANCE) < 1e-4 && cw + w(x[layer - 1], x[j]) < bestw)
23. {
24. swap(x[layer], x[j]);
25. cw += w(x[layer - 1], x[layer]);
26. backTrackTSP(layer + 1);
27. cw -= w(x[layer - 1], x[layer]);
28. swap(x[layer], x[j]);
29. }
30. }
31. }
32. }

# 4 分支限界法

## 4.1 TSP问题的上界

### 4.1.1 求上界思路

初始上界：

利用贪心法+回溯计算上界up

以起始城市作为出发点，每次从当前出发城市发出的多条边中，选择没有遍历过的最短边连接的城市，作为下一步到达的城市。如果当前城市不能通过一段没走过的城市路径到达起始城市的话，就要回溯到上一个城市进行搜索。

具体步骤如下：

1. 输入

城市数量 n；起始城市 start；邻接矩阵 graph，表示城市间的距离；访问标记数组 isVisited，记录城市是否被访问过

2. 初始化

将当前城市添加到路径 path；标记当前城市为已访问；如果路径包含所有城市，检查是否存在从当前城市返回起始城市的路径，更新总成本 cost

3. 深度优先搜索

对于当前城市，从未访问的相邻城市中选择距离最近的城市作为下一个访问城市。递归访问下一个城市，更新路径和成本。如果找到了从当前城市返回起始城市的路径，返回 true。如果没有可访问的城市，回溯到上一个城市，撤销访问标记。

4. 返回

如果成功找到了从当前城市返回起始城市的路径，返回 true，否则返回 false。

### 4.1.2 求上界算法实现

1. **bool** getUpBound(vector<**int**>&path , **double**& cost, **int** current){
2. // 边界条件：当搜索到最后一个点时，查看和起始点有没有路径
3. **if**(path.size() == n){
4. **if**(graph[current][start] != MAX\_DISTANCE){
5. cost += graph[current][start];
6. **return** **true**;
7. }
8. **return** **false**;
9. }
10. vector<**bool**> isVisited(n, **false**);
11. **for**(**int** i =0; i < path.size();i++){
12. isVisited[path[i]] = **true**;
13. }
14. // 对于该点，进行一个 深度优先+贪心
15. **for**(**int** i = 0; i < n; i++){
16. // 这层循环用来找
17. **int** min = MAX\_INT;
18. **int** next = -1;
19. **for**(**int** j =0; j < n; j++){
20. **if**(!isVisited[j] && graph[current][j] != MAX\_DISTANCE && graph[current][j] < min){
21. min = graph[current][j];
22. next = j;
23. }
24. }
25. // 没有路径了，回溯
26. **if**(next == -1){
27. **return** **false**;
28. }
29. // 有路径，查找该路径
30. isVisited[next] = **true**;
31. path.push\_back(next);
32. cost += graph[current][next];
33. **if**(getUpBound(path, cost, next) == **true**){
34. **return** **true**;
35. }
36. // 该路径之后也没路了，返回原来状态，找下一个结点
37. path.pop\_back();
38. cost -= graph[current][next];
39. }
40. // 找寻失败了，没有通路
41. **return** **false**;
42. }

## 4.2 TSP问题的下界

在一条路径上，每个城市i有2条邻接边：进入该城市、离开该城市

第i个城市对应矩阵中第i行

将每一行最小的2个元素相加除以2，并向上取整，得到一个更合理的下界

解释

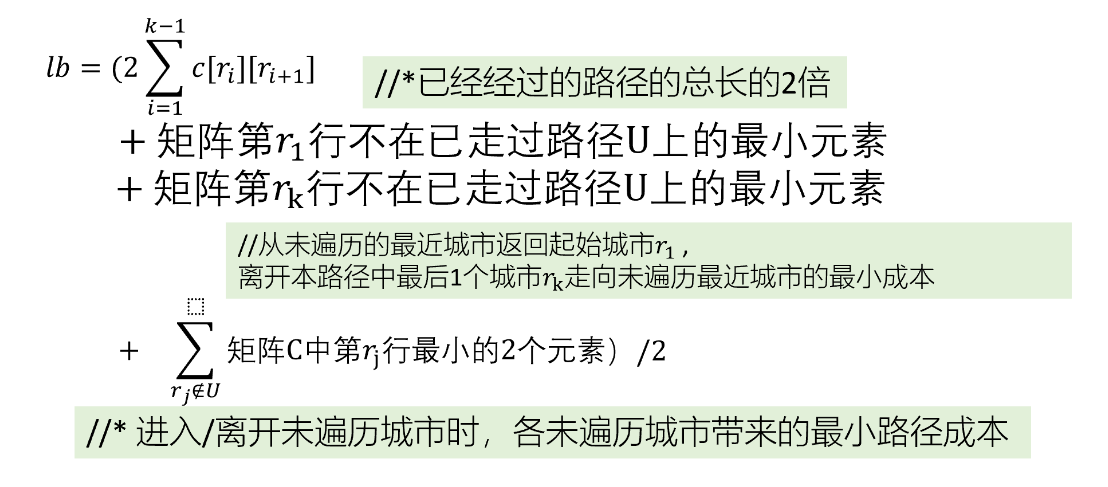
对第i个城市对应的矩阵第i行，2个最小元素为c(i, j1)、 c(i, j2)

由于c(i, j1)= c( j1, i)， c(i, j1)+ c(i, j2)= c( j1, i)+ c(i, j2)， 对应于从最近的上一个城市j1 到达城市i，再到下一个最近城市j2去，即j1→ i → j2，或者：j2→ i → j1

### 4.2.1 计算部分解的目标值下界lb

假设对于1条正在生成的路径/部分解，已经确定的城市顶点（已经经过/遍历的城市）集合为

U=(r1, r2, …, rk) //\* r1r2 …,rk , 该部分解的目标函数的下界*lb*为：





假设 正在生成的路径/部分解为1→4, U={1,4}，未遍历城市={2,3,5}，该部分解下界为：

lb ={ 2\*已经历过的路径总长

+ 从城市1到最近未遍历城市3的距离

+ 从城市4到最近未遍历城市5的距离

+ 进入/离开城市2带来的最小成本

+ 进入/离开城市3带来的最小成本

+ 进入/离开城市5带来的最小成本

} /2

= { 2\*5 + 1 + 3

+ (3+6) + (1+2) + (2+3) } / 2

= 16 (向上取整)

### 4.2.2 求下界算法实现

1. **double** getLb(**const** Node& node){
2. **double** lb = 2\*node.cc;
3. **for**(**int** i = 0; i < n; i++){
4. **if**(i == node.currentPath[0] || i == node.currentPath[node.level-1]){
5. // 到最近未遍历城市的距离
6. **double** min = MAX\_INT;
7. **for**(**int** j = 0; j < n; j++){
8. **if**(!isExist(node.currentPath, j) && graph[i][j] != MAX\_DISTANCE && min > graph[i][j]){
9. min = graph[i][j];
10. }
11. }
12. **if**(min == MAX\_INT) min = 0;
13. lb += min;
14. **continue**;
15. }
16. // 估计未到达城市的最小成本
17. **if**(!isExist(node.currentPath, i)){
18. **double** min1 = MAX\_INT;
19. **double** min2 = MAX\_INT;
20. **for**(**int** j = 0; j < n; j++){
21. **if**(graph[i][j] < min1){
22. min2 = min1;
23. min1 = graph[i][j];
24. }**else** **if**(graph[i][j] < min2){
25. min2 = graph[i][j];
26. }
27. }
28. lb += min1 + min2;
29. }
30. }
31. lb/=2;
32. **return** lb;
33. }

## 4.3 算法思路及步骤

### 4.3.1 算法思路

1. 在起始节点处，U为空，计算本问题的可能下界

down = lb(1)={

2\*已经走过的路径总长

+ 进入/离开第一个城市带来的最小成本

+ 进入/离开第二个城市带来的最小成本

+ 进入/离开第三个城市带来的最小成本

... ...

+ 进入/离开第n个城市带来的最小成本

}/2，取上界

此时lb < up，无需剪枝

2. 以起始节点为扩展结点，依次生成树节点2，3，4，5... ...n

3. 计算这n-1个节点的lb

将小于等于上界的结点加入活结点表ANT中2、3、4

对于下界大于问题上界的结点，抛弃5

4. 从当前活结点表ANT中，以lb为依据，并兼顾结点生成顺序，选取lb最小的结点作为扩展结点，继续生成结点6、7、8，计算着三个结点的lb，将lb大于up的舍弃，新小于up的加入活结点表。

5. 从ANT中，选择lb最小的结点，继续生成结点

6. 按照上述方法，依次扩展搜索树，得到最优解。

### 4.3.2 算法步骤

TSP问题的算法描述：

数组x[1:n]存储搜索路径上的树顶点

1. 采用贪心法，计算问题上界up——用于后续结点剪枝。根据目标函数公式，计算根节点/问题下界down

2. 将活结点表ANT初始化为空

3. 解向量初始化为0

4. 从起始出发，x[1] = start，k = 1

5. while(k >= 1) 遍历步骤，第k步，x[k]表示第k步走的城市

5.1 i=k+1 第k步已选定城市，考虑第k+1步

5.2 x[i] = 0，按照城市序号，首先选择第一个城市

5.3 while(x[i] <= n) 宽度优先，生成x[k]的子结点x[i]=1..2..n

5.3.1 如果路径上城市顶点不重复，则

5.3.1.1 计算x[i]的下界lb

5.3.1.2 if(lb <= up) 将城市和lb加入活结点表

5.3.2 x[i] = x[i] + 1 依次生成x[i]=1的各个兄弟结点

5.4 如果i==n，就将该叶子节点的目标函数值与ANT中所有叶节点、非叶结点的评估值lb相比，是最小的，那么就将该叶节点对应的最优解输出，算法结束

5.5 否则，若i==n，则从ANT中，找出具有最小lb值minlb的叶子节点，

5.5.1 令up = minlb，更新问题上界

5.5.2 删除ANT表中目标函数值lb超出up的结点

5.6 k=表ANT中lb最小的路径上的顶点个数，选lb最小的结点，作为第k+1步的扩展结点

## 4.4 算法实现

### 4.4.1 变量说明

#### 1. class Node

1. **class** Node{
2. **public**:
3. vector <**int**> currentPath;   // 已有路径
4. **double** lb;  // 下界
5. **int** city;   // 当前扩展结点
6. **int** level;  // 属于搜索树第几层，或处于路径中的第几号位置
7. **double** cc;  // currentPath的花销
8. };

该类型为存放在活结点表ANT中的结点，具体含义可以看注释

#### 2. 全局变量说明

1. vector<vector<**double**>> graph; // 图的邻接矩阵
2. **int** start;                    // TSP的起始点
3. **int** n;                        // 有多少个城市
4. **long** nodeNum = 0;             // 用来查看总共查询了多少结点

全局变量有四个，分别是存储图的邻接矩阵，起始城市，城市数目，以及搜索结点的数目。

#### 3. Node的比较函数

1. **bool** cmp(**const** Node&a, **const** Node&b){
2. **return** a.lb > b.lb;
3. }

该函数用来辅助构造小顶堆表ANT

#### 4. isExist函数

1. **bool** isExist(**const** vector<**int**>&cPath, **int** city){
2. **return** find(cPath.begin(), cPath.end(), city) != cPath.end();
3. }

该函数用来查看某个城市是否已经在当前路径中。

#### 5. 显示信息函数

1. **void** printNode(Node node){
2. cout <<endl<< "path : " << endl;
3. **for**(**int** i = 0; i < node.level; i++){
4. cout << node.currentPath[i] << "-->" ;
5. }
6. cout << endl << "city : " << node.city << endl << "lb : " << node.lb << endl;
7. cout << "current cost : " << node.cc << endl;
8. }

该函数用来查看node的信息。

### 4.4.2 核心分支限界函数实现

1. **void** branchLimitTSP(){
2. // 初始化优先队列，以lb小的优先
3. priority\_queue<Node, vector<Node>, decltype(&cmp)> ant(cmp);
4. Node root;
5. root.currentPath = {start};
6. root.lb = getLb(root);
7. root.level=1, root.cc = 0, root.city = start;
8. ant.push(root);
9. // 获取上界
10. vector<**int**>path = {start};
11. **double** up;
12. getUpBound(path, up, start);
13. Node current, result;
14. **while**(!ant.empty()){
15. nodeNum ++ ;
16. current = ant.top();
17. ant.pop();
18. **if**(current.level == n){
19. **if**(graph[current.city][start]+current.lb <= up){
20. up = current.lb + graph[current.city][start];
21. // 得到最优解了已经
22. current.cc += graph[current.city][start];
23. // printNode(current);
24. result = current;
25. }
26. }**else**{
27. // 生成子结点
28. **for**(**int** i = 0; i < n; i++){
29. **if**(isExist(current.currentPath, i) || graph[current.city][i] == MAX\_DISTANCE){
30. **continue**;
31. }
32. Node child;
33. child.city = i;
34. child.currentPath = current.currentPath;
35. child.currentPath.push\_back(i);
36. child.cc = current.cc + graph[current.city][child.city];
37. child.lb = getLb(child);
38. **if**(child.lb <= up){
39. child.level = current.level + 1;
40. ant.push(child);
41. }
42. }
43. }
44. }
45. // cout << "wrong" << endl;
46. printNode(result);
47. }

# 5 执行结果

## 5.1 结果表格

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 问题 | 求解算法 | 最短回路 | 路径总长度 | 搜索过的结点总数 | 程序运行时间（s） |
| 15 | 回溯 | 567443-->566783-->566750-->567238-->566631-->566993-->566967-->567547-->566798-->566751-->567439-->566802-->568098-->566742-->567260-->567443 | 5506.88 | 254516 | 0.028 |
| 分支限界 | 567443-->566783-->566750-->567238-->566631-->566993-->566967-->567547-->566798-->566751-->567439-->566802-->568098-->566742-->567260-->567443 | 5506.88 | 6574 | 0.086 |
| 20 | 回溯 | 567443-->567260-->566742-->568098-->566802-->567439-->567322-->566751-->33109-->566798-->567547-->566999-->566967-->567203-->565696-->566993-->566631-->567238-->566750-->566783-->567443 | 6987.51 | 76201709 | 9.776 |
| 分支限界 | 567443-->567260-->566742-->568098-->566802-->567439-->567322-->566751-->33109-->566798-->567547-->566999-->566967-->567203-->565696-->566993-->566631-->567238-->566750-->566783-->567443 | 6987.51 | 29903 | 0.672 |
| 22 | 回溯 | 567443-->567260-->566742-->568098-->566802-->567439-->566751-->567322-->566747-->566720-->33109-->566798-->567547-->566999-->566967-->567203-->565696-->566993-->566631-->567238-->566750-->566783-->567443 | 7690.8 | 486666893 | 64.719 |
| 分支限界 | 567443-->567260-->566742-->568098-->566802-->567439-->566751-->567322-->566747-->566720-->33109-->566798-->567547-->566999-->566967-->567203-->565696-->566993-->566631-->567238-->566750-->566783-->567443 | 7690.8 | 11068 | 0.328 |
| 30 | 回溯 | 565492-->565496-->565648-->565621-->565773-->565531-->567618-->567497-->565630-->565801-->565753-->565562-->566010-->565631-->565898-->565675-->567500-->567510-->567526-->565964-->567531-->566074-->565859-->565610-->565516-->565551-->565558-->565559-->567891-->565633-->565492 | 11426.6 | 6909104967 | 976.548 |
| 分支限界 | 跑不出来 |  |  |  |

## 5.2 执行结果截图

### 5.2.1 回溯法

15个基站，回溯法，起始点567443

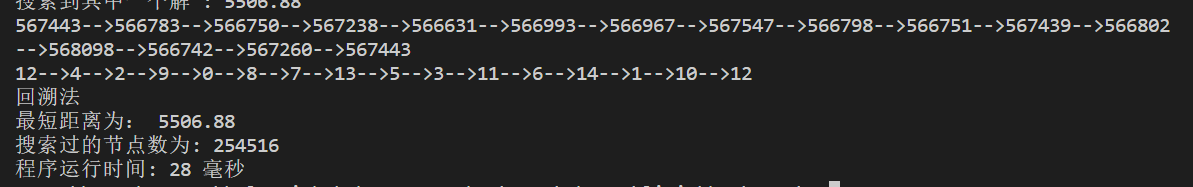


图5-1 15个基站回溯法执行结果

20个基站，回溯法，起始点567443

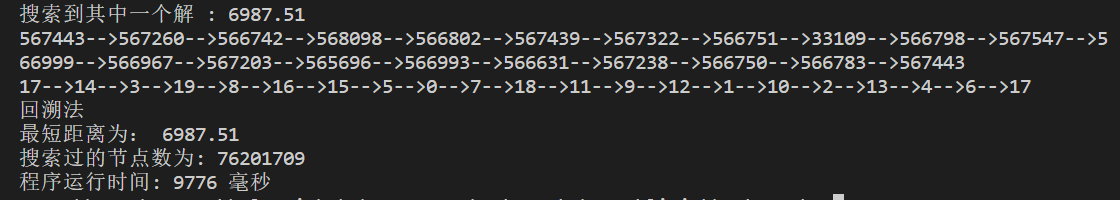


图5-2 20个基站回溯法执行结果

22个基站，回溯法，起始点567443

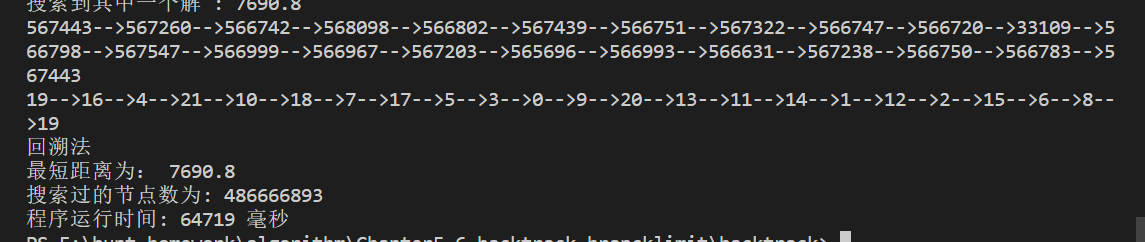


图5-3 22个基站回溯法执行结果

30个基站，回溯法，起始点565492

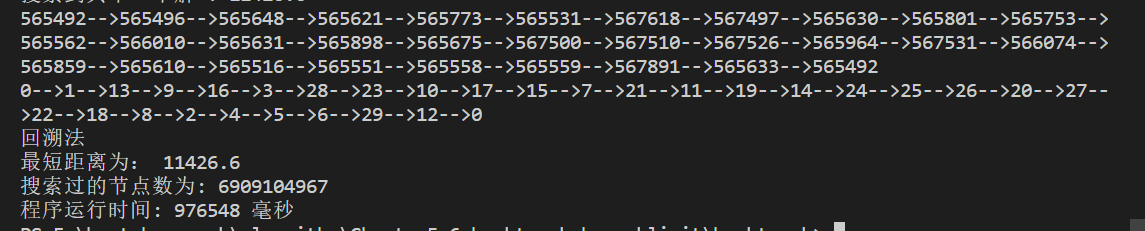


图5-4 30个基站回溯法执行结果

### 5.2.2分支限界法

15个基站，分支限界法，起始点567443

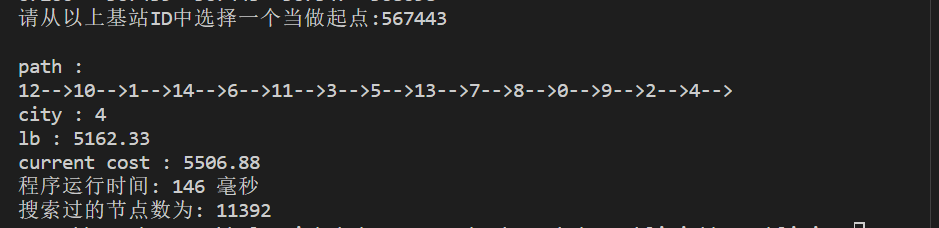


图5-5 15个基站分支限界法执行结果

20个基站，分支限界法，起始点567443

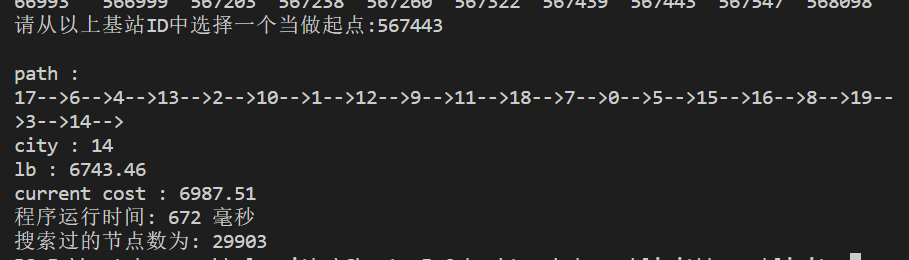


图5-6 20个基站分支限界法执行结果

22个基站，分支限界法，起始点567443

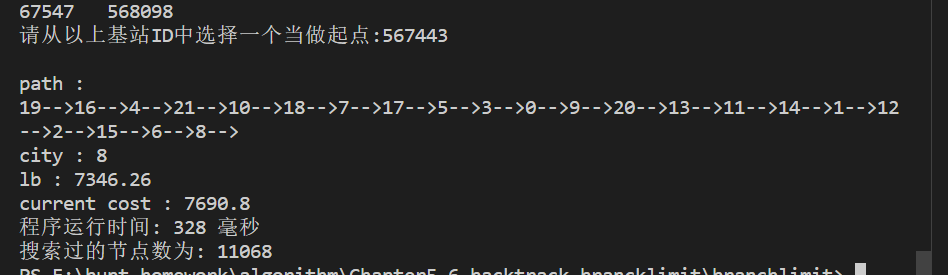


图5-7 22个基站分支限界法执行结果

# 6 程序源代码

## 6.1 回溯法

#include <iostream>

#include <vector>

#include <fstream>

#include <sstream>

#include <algorithm>

#include <chrono>

using namespace std;

#define MAX\_DISTANCE 99999

#define MAX\_INT 1000000000

// 定义全局变量

vector<int> bestx;            // 记录最佳路径

double cw;                       // 当前已经走过的部分路径的总长

vector<int> x;                // 当前已经生成的部分路径

double bestw;                    // 最优路径长度

vector<vector<double>> graph; // 图的邻接矩阵

int start;                    // TSP的起始点

int n;                        // 有多少个城市

long long nodeNum = 0;

// 将基站id转为数组的索引

int getIndexFromId(int id, vector<int> ids)

{

    for (int i = 0; i < ids.size(); i++)

    {

        if (id == ids[i])

            return i;

    }

    return -1;

}

// 获取用户输入的起点

int getInputStart(vector<int> ids)

{

    int id;

    for (int i = 0; i < ids.size(); i++)

    {

        cout << ids[i] << "\t";

    }

    cout << endl

         << "请从以上基站ID中选择一个当做起点:";

    cin >> id;

    while (true)

    {

        auto it = find(ids.begin(), ids.end(), id);

        if (it == ids.end())

        {

            cout << "输入不合法，请重新输入" << endl;

            cin >> id;

        }

        else

        {

            break;

        }

    }

    return getIndexFromId(id, ids);

}

double w(int a, int b)

{

    // cout << a << "--" << b << ":" << graph[a][b] << endl;

    return graph[a][b];

}

void swap(int &a, int &b)

{

    // cout << "swap" << a << " : " << b << endl;

    int temp = a;

    a = b;

    b = temp;

}

void backTrackTSP(int layer)

{

    nodeNum+=1;

    if (layer == n)

    {

        // cout << "layer == n" << endl;

        if (!abs(w(x[n - 1], x[n]) - MAX\_DISTANCE) < 1e-4 && !abs(w(x[n], start) - MAX\_DISTANCE) < 1e-4 && cw + w(x[n - 1], x[n]) + w(x[n], start) < bestw)

        {

            // 找到更优的路径，更新

            for (int j = 1; j <= n; j++)

            {

                bestx[j] = x[j];

            }

            bestw = cw + w(x[n - 1], x[n]) + w(x[n], start);

            cout << "搜索到其中一个解 : " << bestw << endl;

        }

    }

    else

    {

        for (int j = layer; j <= n; j++)

        {

            if (!abs(w(x[layer - 1], x[j]) - MAX\_DISTANCE) < 1e-4 && cw + w(x[layer - 1], x[j]) < bestw)

            {

                swap(x[layer], x[j]);

                cw += w(x[layer - 1], x[layer]);

                backTrackTSP(layer + 1);

                cw -= w(x[layer - 1], x[layer]);

                swap(x[layer], x[j]);

            }

        }

    }

}

int main()

{

    ifstream inputFile("./1-2-30.txt");

    if (!inputFile.is\_open())

    {

        cerr << "文件打开失败" << endl;

        return 1;

    }

    string line;

    getline(inputFile, line);

    stringstream ss(line);

    // 读取第一行基站数据

    int id;

    double value;

    vector<int> ids;

    while (ss >> id)

    {

        ids.push\_back(id);

    }

    // 跳过第一列读取距离数据

    while (inputFile >> id)

    {

        vector<double> row;

        for (int i = 0; i < ids.size(); i++)

        {

            inputFile >> value;

            row.push\_back(value);

        }

        graph.push\_back(row);

    }

    inputFile.close();

    for (int i = 0; i < graph.size(); i++)

    {

        for (int j = 0; j < graph[i].size(); j++)

        {

            cout << graph[i][j] << "\t";

        }

        cout << endl;

    }

    n = ids.size();

    start = getInputStart(ids);

    bestx.resize(n + 1);

    x.resize(n + 1);

    cw = 0;

    bestw = MAX\_INT;

    for (int i = 1; i <= n; i++)

    {

        x[i] = getIndexFromId(ids[i - 1], ids);

    }

    swap(x[start + 1], x[1]);

    auto startTime = chrono::high\_resolution\_clock::now();

    backTrackTSP(2);

    auto endTime = chrono::high\_resolution\_clock::now();

    for (int i = 1; i <=n; i++)

    {

        cout << ids[bestx[i]] << "-->";

    }

    cout << ids[start] << endl;

    for (int i = 1; i <=n; i++)

    {

        cout << bestx[i] << "-->";

    }

    cout << start << endl;

    auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime);

    cout << "回溯法"<<endl;

    cout << "最短距离为： " << bestw << endl;

    cout << "搜索过的节点数为: " << nodeNum << endl;

    cout << "程序运行时间: " << duration.count() << " 毫秒" << std::endl;

    return 0;

}

## 6.2 分支限界法

#include <iostream>

#include <vector>

#include <fstream>

#include <sstream>

#include <algorithm>

#include <chrono>

#include <queue>

using namespace std;

#define MAX\_DISTANCE 99999

#define MAX\_INT 1000000000

class Node{

public:

    vector <int> currentPath;   // 已有路径

    double lb;  // 下界

    int city;   // 当前扩展结点

    int level;  // 属于搜索树第几层，或处于路径中的第几号位置

    double cc;  // currentPath的花销

};

bool cmp(const Node&a, const Node&b){

    return a.lb > b.lb;

}

bool isExist(const vector<int>&cPath, int city){

    return find(cPath.begin(), cPath.end(), city) != cPath.end();

}

// 定义全局变量

vector<vector<double>> graph; // 图的邻接矩阵

int start;                    // TSP的起始点

int n;                        // 有多少个城市

long nodeNum = 0;             // 用来查看总共查询了多少结点

void printNode(Node node){

    cout << endl << "city : " << node.city << endl << "lb : " << node.lb << endl;

    cout << "current cost : " << node.cc << endl;

    cout << "path : " << endl;

    for(int i = 0; i < node.level; i++){

        cout << node.currentPath[i] << "-->" ;

    }

    cout << start << endl;

}

// 将基站id转为数组的索引

int getIndexFromId(int id, vector<int> ids)

{

    for (int i = 0; i < ids.size(); i++)

    {

        if (id == ids[i])

            return i;

    }

    return -1;

}

// 获取用户输入的起点

int getInputStart(vector<int> ids)

{

    int id;

    for (int i = 0; i < ids.size(); i++)

    {

        cout << ids[i] << "\t";

    }

    cout << endl

         << "请从以上基站ID中选择一个当做起点:";

    cin >> id;

    while (true)

    {

        auto it = find(ids.begin(), ids.end(), id);

        if (it == ids.end())

        {

            cout << "输入不合法，请重新输入" << endl;

            cin >> id;

        }

        else

        {

            break;

        }

    }

    return getIndexFromId(id, ids);

}

bool getUpBound(vector<int>&path , double& cost, int current){

    // 边界条件：当搜索到最后一个点时，查看和起始点有没有路径

    if(path.size() == n){

        if(graph[current][start] != MAX\_DISTANCE){

            cost += graph[current][start];

            return true;

        }

        return false;

    }

    vector<bool> isVisited(n, false);

    for(int i =0; i < path.size();i++){

        isVisited[path[i]] = true;

    }

    // 对于该点，进行一个 深度优先+贪心

    for(int i = 0; i < n; i++){

        // 这层循环用来找

        int min = MAX\_INT;

        int next = -1;

        for(int j =0; j < n; j++){

            if(!isVisited[j] && graph[current][j] != MAX\_DISTANCE && graph[current][j] < min){

                min = graph[current][j];

                next = j;

            }

        }

        // 没有路径了，回溯

        if(next == -1){

            return false;

        }

        // 有路径，查找该路径

        isVisited[next] = true;

        path.push\_back(next);

        cost += graph[current][next];

        if(getUpBound(path, cost, next) == true){

            return true;

        }

        // 该路径之后也没路了，返回原来状态，找下一个结点

        path.pop\_back();

        cost -= graph[current][next];

    }

    // 找寻失败了，没有通路

    return false;

}

double getLb(const Node& node){

    double lb = 2\*node.cc;

    for(int i = 0; i < n; i++){

        if(i == node.currentPath[0] || i == node.currentPath[node.level-1]){

            // 到最近未遍历城市的距离

            double min = MAX\_INT;

            for(int j = 0; j < n; j++){

                if(!isExist(node.currentPath, j) && graph[i][j] != MAX\_DISTANCE && min > graph[i][j]){

                    min = graph[i][j];

                }

            }

            if(min == MAX\_INT) min = 0;

            lb += min;

            continue;

        }

        // 估计未到达城市的最小成本

        if(!isExist(node.currentPath, i)){

            double min1 = MAX\_INT;

            double min2 = MAX\_INT;

            for(int j = 0; j < n; j++){

                if(graph[i][j] < min1){

                    min2 = min1;

                    min1 = graph[i][j];

                }else if(graph[i][j] < min2){

                    min2 = graph[i][j];

                }

            }

            lb += min1 + min2;

        }

    }

    lb/=2;

    return lb;

}

void branchLimitTSP(){

    // 初始化优先队列，以lb小的优先

    priority\_queue<Node, vector<Node>, decltype(&cmp)> ant(cmp);

    Node root;

    root.currentPath = {start};

    root.lb = getLb(root);

    root.level=1, root.cc = 0, root.city = start;

    ant.push(root);

    // 获取上界

    vector<int>path = {start};

    double up;

    getUpBound(path, up, start);

    Node current, result;

    while(!ant.empty()){

        nodeNum ++ ;

        current = ant.top();

        ant.pop();

        if(current.level == n){

            if(graph[current.city][start]+current.lb <= up){

                up = current.lb + graph[current.city][start];

                // 获得某个次优解

                current.cc += graph[current.city][start];

                result = current;

            }

        }else{

            // 生成子结点

            for(int i = 0; i < n; i++){

                if(isExist(current.currentPath, i) || graph[current.city][i] == MAX\_DISTANCE){

                    continue;

                }

                Node child;

                child.city = i;

                child.currentPath = current.currentPath;

                child.currentPath.push\_back(i);

                child.cc = current.cc + graph[current.city][child.city];

                child.lb = getLb(child);

                if(child.lb <= up){

                    child.level = current.level + 1;

                    ant.push(child);

                }

            }

        }

    }

    printNode(result);

}

int main()

{

    ifstream inputFile("./1-2-30.txt");

    if (!inputFile.is\_open())

    {

        cerr << "文件打开失败" << endl;

        return 1;

    }

    string line;

    getline(inputFile, line);

    stringstream ss(line);

    // 读取第一行基站数据

    int id;

    double value;

    vector<int> ids;

    while (ss >> id)

    {

        ids.push\_back(id);

    }

    // 跳过第一列读取距离数据

    while (inputFile >> id)

    {

        vector<double> row;

        for (int i = 0; i < ids.size(); i++)

        {

            inputFile >> value;

            row.push\_back(value);

        }

        graph.push\_back(row);

    }

    inputFile.close();

    for (int i = 0; i < graph.size(); i++)

    {

        for (int j = 0; j < graph[i].size(); j++)

        {

            cout << graph[i][j] << "\t";

        }

        cout << endl;

    }

    n = ids.size();

    start = getInputStart(ids);

    Node result;

    auto startTime = chrono::high\_resolution\_clock::now();

    branchLimitTSP();

    auto endTime = chrono::high\_resolution\_clock::now();

    auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime);

    cout << "程序运行时间: " << duration.count() << " 毫秒" << std::endl;

    cout << "搜索过的节点数为: " << nodeNum << endl;

    return 0;

}