TSP 剪枝效率实验报告

王资 18214668 计算机技术

1.基本思想

从旅行商售货问题（TSP）的描述中，可知该回溯问题为排列类问题，同时，由于有优化最小路程的目标，可以以该目标构造 bound 函数，即当前排列中需要消耗的总路程已经超过了已知的最优路程时，即可剪枝。

本实验旨在对比没有剪枝的全排序和带剪枝的排序回溯算法之间的效率，即前者没有 bound 函数（或永真）而后者有。

2.代码中的数据结构

int[][] cities 该结构为各个城市之间的连通图，是一个无向图，即该矩阵的转置和原矩阵相同，cities[i][j]代表从第 i 个城市到第 j 个城市所需的路程，NoEdge 表示两个城市不直接相连，其中 NoEdge 是一个大整数。

使用类 TravelingSale 来包装需要的数据结构，尽量避免使用大量全局变量，以下均为类 TravelingSale 的成员。

int n 代表共有 n 个城市。

Type** a 使用密集矩阵代表各个城市之间距离的图。（与 int[][] cities 功能相同。）

Type* x 表示旅行商遍历各个城市的顺序。

Type bestc 用于存储目前为止遍历所有城市所需的最优总路程。

Type* bestx 用于储存遍历所有城市所需的最优总路程对应的路径城市的路径。

Type cc 用于存储目前为止游历的城市已经走过的路程。

3.程序流程

1）将密集图和策略（剪枝与不剪枝）传入，创建类 TravelingSale 的实例。

2）在构造函数中根据对应的策略，执行排序型的回溯，最终得到 bestc 和 bestx.

3）计算回溯过程所需的时间，并输出 bestc 和 bestx，确认各个策略所得到的解是否一致。

4.调试过程出现的问题和解决方法

1）bestc 计算不正确（使用小图校验算法正确性），得到负数等情况。原因是课件提供的代码在排序完成后的判断使用的是 a[x[n]][1]，目的是查看第 n 个城市能否回到第一个城市，但第一个访问的城市不一定是城市 1，因此需要把这一系列关于初始城市的地方改成 x[1]才是正确的。

2）n 与图矩阵的宽或高度不一致，导致的一些问题，图矩阵的宽度 SIZE 应该比 n 大 1，因为第 0 行和第 0 列需要全部置零，表示初始状态下可以从任一城市开始游历，在实现过程中，有一些地方混用了 n 和 SIZE 导致了一些错误，如总路程的计算错误、段错误等问题。

5.运行结果

使用的图的矩阵表示如下所示：

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	2 NoEdge	NoEdge		6 NoEdge	NoEdge	NoEdge		1 NoEdge	NoEdge	NoEdge	NoEdge	NoEdge		5 NoEdge		3 NoEdge		2 NoEdge
0	2	0 NoEdge		2 NoEdge	NoEdge		4 NoEdge	NoEdge	NoEdge		3 NoEdge	NoEdge		7 NoEdge		1 NoEdge	NoEdge	NoEdge		2
0 NoEdge	NoEdge		0 NoEdge	NoEdge		2 NoEdge	NoEdge		1 NoEdge	NoEdge		1 NoEdge		9 NoEdge	NoEdge		4 NoEdge		6 NoEdge	
0 NoEdge		2 NoEdge		0	1 NoEdge	NoEdge		4 NoEdge		2 NoEdge	NoEdge		4 NoEdge	NoEdge		5 NoEdge	NoEdge	NoEdge		8
0	6 NoEdge	NoEdge		1	0 NoEdge		2 NoEdge		6 NoEdge	NoEdge		5 NoEdge	NoEdge		8 NoEdge		5 NoEdge	NoEdge		10
0 NoEdge	NoEdge		2 NoEdge	NoEdge		0 NoEdge		1 NoEdge	NoEdge		1 NoEdge	NoEdge		5 NoEdge		4 NoEdge		8 NoEdge	NoEdge	
0 NoEdge		4 NoEdge	NoEdge		2 NoEdge		0	1 NoEdge	NoEdge		2 NoEdge		7 NoEdge		5 NoEdge	NoEdge	NoEdge		6 NoEdge	
0 NoEdge	NoEdge	NoEdge		4 NoEdge		1	1	0 NoEdge		2 NoEdge	NoEdge		3 NoEdge	NoEdge		2 NoEdge		10 NoEdge	NoEdge	
0	1 NoEdge		1 NoEdge		6 NoEdge	NoEdge	NoEdge		0 NoEdge		3 NoEdge	NoEdge		4 NoEdge	NoEdge	NoEdge		6	7 NoEdge	
0 NoEdge	NoEdge	NoEdge		2 NoEdge	NoEdge	NoEdge		2 NoEdge		0	1 NoEdge		5 NoEdge	NoEdge		3 NoEdge	NoEdge	NoEdge		2
0 NoEdge		3 NoEdge	NoEdge	NoEdge		1	2 NoEdge		3	1	0	2 NoEdge	NoEdge		8 NoEdge	NoEdge		1 NoEdge		3
0 NoEdge	NoEdge		1 NoEdge		5 NoEdge	NoEdge	NoEdge	NoEdge	NoEdge		2	0	1 NoEdge	NoEdge		4 NoEdge		5 NoEdge		2
0 NoEdge	NoEdge	NoEdge		4 NoEdge	NoEdge		7	3 NoEdge		5 NoEdge		1	0	5 NoEdge		1 NoEdge		9 NoEdge		8
0 NoEdge		7	9 NoEdge	NoEdge		5 NoEdge	NoEdge		4 NoEdge	NoEdge	NoEdge		5	0	1 NoEdge		3 NoEdge	NoEdge		2
0	5 NoEdge	NoEdge	NoEdge		8 NoEdge		5 NoEdge	NoEdge	NoEdge		8 NoEdge	NoEdge		1	0	5 NoEdge		1 NoEdge		6
0 NoEdge		1 NoEdge		5 NoEdge		4 NoEdge		2 NoEdge		3 NoEdge		4	1 NoEdge		5	0	7 NoEdge		2 NoEdge	
0	3 NoEdge		4 NoEdge		5 NoEdge	NoEdge	NoEdge	NoEdge	NoEdge	NoEdge	NoEdge	NoEdge		3 NoEdge		7	0 NoEdge		1 NoEdge	
0 NoEdge	NoEdge	NoEdge	NoEdge	NoEdge		8 NoEdge		10	6 NoEdge		1	5	9 NoEdge		1 NoEdge	NoEdge		0 NoEdge	NoEdge	
0	2 NoEdge		6 NoEdge	NoEdge	NoEdge		6 NoEdge		7 NoEdge	NoEdge	NoEdge	NoEdge	NoEdge	NoEdge		2	1 NoEdge		0	4
0 NoEdge		2 NoEdge		8	10 NoEdge	NoEdge	NoEdge	NoEdge		2	3	2	8	2	6 NoEdge	NoEdge	NoEdge		4	0

得到的结果是

```
/home/wz/CLionProjects/TSP/cmake-build-debug/TSP
finish with pruning costs 60s
The best cost is 27
The best path is: 1 -> 9 -> 3 -> 12 -> 13 -> 16 -> 2 -> 20 -> 10 -> 4 -> 5 -> 7 -> 8 -> 6 -> 11 -> 18 -> 15 -> 14 -> 17 -> 19 -> 1
finish without pruning costs 10550s
The best cost is 27
The best path is: 1 -> 9 -> 3 -> 12 -> 13 -> 16 -> 2 -> 20 -> 10 -> 4 -> 5 -> 7 -> 8 -> 6 -> 11 -> 18 -> 15 -> 14 -> 17 -> 19 -> 1

Process finished with exit code 0
```

分析结果，剪枝和不剪枝得到的路径以及最终的总路程数是一样的，而剪枝的效率是不剪枝（穷举）的 ~175 倍，在我所设计的连通图下，剪枝需要 60s 找到最优的路径，而穷举的方法需要近 3h 才能结束程序。

6.总结

本实验旨在证明剪枝的效率对于回溯算法效率的提高，从树的结构已知，对于子集树，树的节点数是 O(m^n), 其中 m 代表 max(每个节点可以取的值的数量), n 代表需要解决的问题的大小；排列树中，树的节点数是 O(n!). 都是指数型增长的，因此如果没有剪枝的操作，而是要遍历这些树，对于规模稍大的问题就会很慢，而剪枝的操作可以避免访问大部分的节点，从该实验结果来看，对于 KSP, n=20 时，有 174/175 的节点没有访问，因此效率提高了 175 倍。

7.存在问题和改进设想

应用了剪枝算法后，还有一种更加快速的算法是在初始化时对节点排序，使得可能的取值数越少的节点排在越前面，这样如果在前部分发生剪枝，能够剪掉更多的节点，进而提高效率。但实验测试后发现时间并没有减少，反而增加了（从 60s 增加到 66s），这可能是排序的方法对于排列类回溯问题起的作用较小，因为要得到第一个最优值需要首先得到第一个符合显式约束的路径，这本身就需要遍历大量节点，其次可能是数据设计的原因，使得排序之后前面得到的当前最优值比较大，每次都需要得到完整的路径后和这个最优值对比并更新了最优值，最终，比默认排序的方法访问了更多的节点。因此该方法对于子集类回溯问题的效率应该会更高，而对于排列类回溯问题有效与否还要看具体的数据。