算法设计与分析实验报告

实验名称：推销员问题（ETS 算法）

一、问题陈述，相关背景、应用及研究现状的综述分析

问题陈述：有一推销员，欲到 n(n<=10)个城市推销产品。为了节省旅行费用，在出发前他查清了任意两个城市间的旅行费用，想找到一条旅行路线，仅经过每个城市一次，且使旅行费用最少。本问题已知城市 n，和 n*n 的表达任意两个城市间费用的矩阵。试求最短路径及其费用。

分析：把城市当作结点，默认任意两城市间都有路径（如果两城市间没有路径，可以认为两城市间的路径长度无穷大），把任意两城市间的旅行费用当作路径长度。那么这个城市+旅行费用就构成了一个带权有向完全图。现在的问题就转化成，找一条最短路径，使得每个城市都在路径上且只存在于路径上一次。

这个问题表述有两种情况：

1 是推销员会回到起点城市，那么推销员所走的路径就是由这些城市组成的图上的一条最短哈密顿回路。

2 是推销员不会回到起点城市，那么推销员所走的路径就是一条最短哈密顿通路。

现就第一种情况进行讨论：

查阅资料知道，求一个图上的哈密顿回路存不存在是 NP 完全问题[i]。可以断言，找完全图上的最短哈密顿回路是不是无穷大问题也是一个 NP 完全问题，因为可以通过加无穷大边的方法把普通图变成一个完全图，如果能够在此完全图上找到一个最短哈密顿回路，而且最短路径长度不是无穷大的话：那么原图的哈密顿回路就存在（因为在没有用到长度无穷大的路径的情况下已经找到了一个哈密顿回路）。反之，如果最短路径是无穷大的话，原图的哈密顿回路就不存在（如果存在，完全图的最短哈密顿回路的路径不可能是无穷大），所以这两个问题其实是一个问题，都是 NP 完全问题。当前问题比求证完全图上的最短哈密顿回路是不是无穷大还要更进一步，要求出最短值，所以可以断定其算法的时间复杂度至少是不会小于 NP 完全问题的时间复杂度的。

NP 完全问题的一个最主要特点就是时间复杂度是非多项式的复杂度[ii]。所以没有一个时间复杂度小于 O(n^p)的通用的算法。考虑到此题的城市数量 n 是小于等于 10 的，想出枚举的算法。因为路径上的城市只能出现一次，所以把一条路径上有序经过的城市编号记录下来，就形成了 n 个城市的一个排列。这样的排列最多有 A(n,n) = n! 种。在 n 等于 10 的情况下，最多有 10！=3 628 800 种排列，这是普通计算机可以计算的范围（<1e9）。得到每一种序列后，因为每个序列都是一条不同合法的路径，所以要计算每一条路径的长度后，取最小即是所求的最短路径长度了，同时记录一下当前最短路径的排列，就得到了路径。

在枚举计算过程中，发现这个枚举中有很多地方进行了重复的计算：例如在计算 n=5 情况下，12345 和 12354 这种情况下，1->2->3 这条路径的长度就被计算了两次，又例如在 n=9 的情况下，123******这种情况下，1->2->3 这条路径的长度就被计算了 A(6,6)=720 次。可以预见，在 n 更大的情况下，像这样的重复计算是越来越多的。在解决重复计算类问题的算法里，动态规划算法是非常有用的，所以考虑使用动态规划算法来优化这个枚举算法。

二、模型拟制、算法设计和正确性证明

模型拟制：

因为 n 的范围时小于 10 的，所以采用邻接矩阵的办法来存储这 n 个城市所构成的有向带权完全图。M[i][j]=k,就表示第 i 个城市到第 j 个城市的花费是 k。预定义无穷大为 INF，如果 k=INF 就说明 i 到 j 没有路径。

枚举的算法：

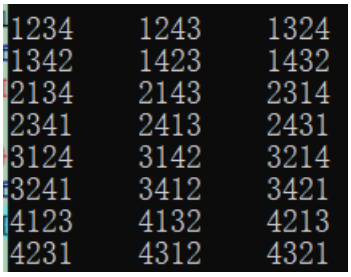
算法设计：相当于是找 1~n 所有的排列，考虑递归的办法。

递归的参数选用的是 th，表示当前已经枚举了排列的第 th 个数，将要枚举下一个数。

如果 th 已经是 n 了，表示现在已经找到了一个有效排列，计算出路径后更新答案和答案路径即可。

如果 th 小于 n，那么枚举下一个数，如果这个数没有被使用过(使用 used 数组记录)，就递归进入下一层 th+1。

正确性证明：在 n=4 的情况下，测试输出所有的排列截图如下：



在检查了数组大小后，认为递归的程序实现没问题。

因为是枚举出了所有的哈密顿回路，然后再取其中的最小值，所以总是可以找到最短的那一条哈密顿回路，这是枚举算法的特点，就是总能保证正确性，是可靠的算法。

动态规划的算法：

动态规划的基本思路就是用空间解决时间上的问题，简单的想法就是用内存空间储存计算结果，然后再往后递推。递推过程中如果要用到之前的结果，就直接读取内存中储存的结果，不必进行重复计算。如何储存数据，储存什么数据就成了首要的问题，其中储存什么数据是最重要的问题。

储存什么数据：要储存的信息有两个：一是计算的结果，二是这个结果所对应的状态。先说状态，状态选取的是当前走过的城市情况和起点城市的编号和最后一个到达城市（末站城市）的编号。计算的结果就是到达当前状态下的最短路径。注意这里舍去了之前的路径信息，也就是说，之前走过的城市顺序是不知道的，只知道第一个城市的编号和最后一个到达城市的编号。这样做是不影响枚举的正确性的，因为这样的状态已经包含了所有当前状态对后来状态有影响的因素：即当前城市情况和首站城市编号和末站城市编号。这样做的好处是节省了空间[ⅲ]，坏处是找最短路径时要麻烦一点，但还是有办法能找到。

如何储存数据：城市情况用 0 和 1 表示，0 表示没有走过，1 表示走过了。这样将 n 个城市的情况写一块，就形成了一个二进制数 i。例如 n=4 的情况下，i=1001 就表示第 1 和第 4 个城市走过，第 2 和第 3 个城市没走过。城市的编号用十进制表示，使用位运算来实现二进制和十进制的转换。这样一来，使用二维数组就可以存下所有要用到的信息了。

D[i][j][k]=len 表示到达 i 种城市情况下，首战城市为 j，末站城市为 k 的状态下所耗费的最短路径长度。

递推式子：在当前状态量 D[i][j][k]下，枚举下一个城市的编号 next_city，那么下一个城市状态就是 next_state = i+(1<<next_city), 其更新方程为：

D[next_state][j][next_city]=min (D[next_state][j][next_city] , D[i][j][k]+M[k][next_city])

最后要得到答案就是在最终状态量 tar=((1<<n)-1)下找 D[tar][i][j]，i，j 从 0 到 n-1 中的最小值就可以了。

最后要得到路径就是反递推刚才的递推过程，用记录下的最终状态量 tar 和首站城市 j 和末站城市 k，枚举前一个城市 pre，如果满足：

D[tar-(1<<pre)][j][pre]=D[tar][j][k]-M[pre][k]

的话，那 pre 就是要找到的前一站城市，在更新 tar 和 i，j 后就可以往前递推找再上一个的城市了。

正确性检验：

在随机出了 10 个数据后，发现枚举算法和动态规划算法所得到的答案一样：

```
n=9
动态规划答案：1339        5 9 3 7 4 1 8 2 6
动态规划使用时间：0
枚举答案：1339            9 3 7 4 1 8 2 6 5
枚举使用时间:32

n=10
动态规划答案：1367        6 9 5 1 2 8 4 10 3 7
动态规划使用时间：15
枚举答案：1367            10 3 7 6 9 5 1 2 8 4
枚举使用时间:313

n=11
动态规划答案：1372        10 8 6 7 1 9 4 3 5 2 11
动态规划使用时间：15
枚举答案：1372            11 10 8 6 7 1 9 4 3 5 2
枚举使用时间:3640
```

可以判断程序实现是对的。

可以看到所求的哈密顿回路是一条循环往复的回路，所以两种答案的路径顺序虽然不同，但本质是一样的。

本来预想的是可能同一最短路程上最短路径可能有很多条，所以两种方法所得到的路径会本质不同。但数据出得并不好，没有这种情况。如果有更多的数据就好了[iv]。

三、时间和空间复杂性分析

枚举算法的时间复杂度主要在两方面，1 是枚举出排列的时间复杂度是 O(n!)=O(n^n)的，2 是计算每个排列的路径长度的时间复杂度是 O(n)的，二者相乘得到最终算法的时间复杂度：O(n^n)=O(n^(1+n))。n 均指城市数量。

枚举算法的空间复杂度在两方面，1 是存图所用的邻接矩阵 O(n^2),2 是 used 数组和其他大小为 n 的数组，所以最终算法的空间复杂度是 O(n^2)的。

动态规划的时间复杂度：因为每一种状态只会递推一次，所以由总的状态数 2^n*n 得到递推的次数是 O(2^n*n)的，然后每一次递推会用 O(n)的时间枚举下一个城市的编号。最后的用回溯法倒推路径的时间复杂度是 O(n^2)的，所以最终算法的时间复杂度是 O(2^n)=O(2^n*n)*O(n)+O(n^2)。

动态规划的空间复杂度：主要占用的空间在于 D 数组的使用，共用了 n^2*n 的空间。正如动态规划的最显著特点:用空间换取时间，其空间复杂度远远大于枚举算法。所以最终算法的复杂度是 O(n^2)。

复杂度小结如下：

算法	枚举算法	动态规划算法
时间复杂度	$O(n^n)$	$O(2^n)$
空间复杂度	$O(n^2)$	$O(2^n)$

动态规划的时间复杂度比枚举算法的时间复杂度略小，空间复杂度比枚举算法大得多。虽然空间复杂度要大得多，但认为是必要的，因为 n 在 13~20 这个范围类的计算，枚举算法就不行了，而动态规划还可以一战。

```
n=11
动态规划答案：1372          10 8 6 7 1 9 4 3 5 2 11
动态规划使用时间：16
枚举答案：1372              11 10 8 6 7 1 9 4 3 5 2
枚举使用时间:3639

n=12
动态规划答案：1722          11 7 3 6 2 8 10 1 4 9 5 12
动态规划使用时间：32
枚举答案：1722              12 11 7 3 6 2 8 10 1 4 9 5
枚举使用时间:46857

n=13
动态规划答案：1572          9 10 6 2 4 3 12 13 7 8 1 5 11
动态规划使用时间：78
```

枚举法在计算 n=13 这种情况就会运行超过 3min。

而动态规划可以在 3s 内算出 n=20 的情况：

```
n=18
动态规划答案：929           动态规划使用时间：581
n=19
动态规划答案：1260          动态规划使用时间：1281
n=20
动态规划答案：894           动态规划使用时间：2816
```

上述时间单位均为 ms

四、程序实现和实验测试过程

枚举算法：用 get()函数和 fi(th,ans)函数实现，get()函数初始化和启动递归，fi()函数实现递归操作，并且储存答案到 ans。

动态规划算法：用 DP(start,tar,path[])，init()函数实现，分别是起始城市状态、最终城市状态，和路径数组，函数本身返回最短路径长度。

有一个 getdata()函数专门用来出随机的测试数据。

数据保存在 in.txt 文件中，测试时用 freopen 函数读取文件中的数据。

```
n=4
动态规划答案：1685        1 2 3 4
动态规划使用时间：0
枚举答案：1685            4 1 2 3
枚举使用时间:0

n=5
动态规划答案：2198        2 3 5 1 4
动态规划使用时间：0
枚举答案：2198            5 1 4 2 3
枚举使用时间:0

n=6
动态规划答案：1357        5 4 3 2 1 6
动态规划使用时间：0
枚举答案：1357            6 5 4 3 2 1
枚举使用时间:0
```

和黄、莫等同学核对了数据的答案。

五、总结

在解决问题后可以多想想怎么优化解法。

动态规划可以优化重复计算的问题，但面对 NP 完全问题还是有点力不从心，因为其时间复杂度是 2^n，仍然是非多项式的时间复杂度，换句话说如果 n 再大一点，n>=23 时，动态规划算法就无法在 3min 中内算出。

---

[i] https://en.wikipedia.org/wiki/Hamiltonian_path 哈密顿回路；

[ii] https://en.wikipedia.org/wiki/NP-completeness NP 完全问题；

[iii] 这里的节省空间是相对于动态规划算法下其他储存的信息来说的，不是相对于枚举的算法。举一个其他储存信息的例子：考虑用一个十进制数来存储走过的城市信息，按第 i 位存第 i 个城市编号来存放，例如 n=5 的情况下，12300 表示当前已按顺序走过了 1、2、3 城市。那么这个方法所占的空间是 10^n，是大于正文方法所占的空间 2^n*n 的，具体可以看第三大点，空间复杂度分析；

[iv] 本来南开中学的 oj 上有一个非常适合测试程序的题目，http://oi.nks.edu.cn/zh/Problem/Details?id=1752 ，但是它没有开放注册，就没有办法使用它的数据集来测试代码。