最短巡视路线

数学系　04061012　邓宇善

2007 年 6 月 10 日

摘　要　本文利用启发式算法求得近似最短巡视路线，后用*tspsol*程序求得精确最优解。对于本巡视问题，两种方法求得的最优解是相等的，最短路线长度均为 185.1。

§1. 巡视问题

1998 年夏天某县遭受水灾，为考察灾情、组织自救，县领导决定，带领有关部门负责人到全县各乡（镇）、村巡视。巡视路线指从县政府所在地出发，走遍各乡（镇）、村，又回到县政府所在地的路线。请设计出总路程最短的路线（乡镇、村的公路网示意图见图 1）。 要求：给出算法、实现程序和计算结果（包括路线和路线长度）。

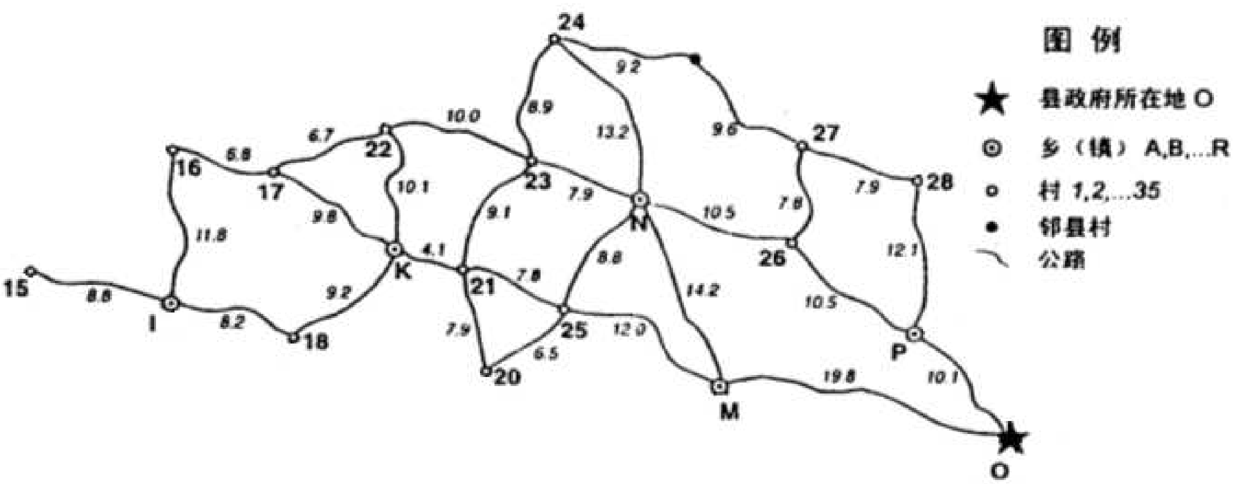

图 1

§2. 一般观察

若把图 1 中的乡镇、村看成是节点，它们之间的路线看成边，则可构成一无向连通图 G。从县政府所在地出发，走遍各乡（镇）、村，最后回到县政府所在地，我们得到一条遍历图 G 所有节点的路线，要

使该路线最短，显然该路线应尽量少重复或不重复已经访问过的乡镇或村。如果，G 存在 Hamilton 回路，则最短的 Hamilton 回路显然就是最短的巡视路线。但是，图 G 不存在 Hamilton 回路，原因是节点 15（即 15 村，记为 v_{15}）的度 $deg(v_{15}) = 1$，故要访问 15 村，I 乡（镇）（记为 v_I）必需被访问 2 次。事实上，去掉 15 村后，图 $G - \{v_{15}\}$ 就是一个 Hamilton 图，可找到它的一条最短 Hamilton 回路 H，在 H 的基础上插入访问 15 村的路径 $v_I \to v_{15} \to v_I$，由此得到的遍历路线，我们估计这就是最短的巡视路线，它的总长度为 185.1。尽管如此，一方面为了证明我们的估计是正确的，另一方面要给出解决此类问题的一般方法，我们必须给出严格的算法及相关的证明。

§3. 求解初始化

设图 $G =< V, E >$，$|V| = n$。我们分两种情况：(a) 图 G 是 Hamilton 图；(b) 图 G 不是 Hamilton 图，也不一定如上述般－－只要去掉极少数节点后得到的生成子图是 Hamilton 图。对于 (a) 我们只寻找它的最短 Hamilton 回路即可解决问题；(b) 是我们关注的情况，为从某一点 a 转移到另一个点 b，a 和 b 是不相邻的，若在中间我们不得不要重复访问已访问过的点，则我们可以做的只能是尽量减少 $a \to b$ 的路程，故我们需要 a 间 b 的最短路径。因此，我们把图 G 中所有的非相邻节点都补上一条边，该边的权是两节点间的最短路径，于是我们得到完全图 K_n，对于 K_n 我们便可采用 (a) 的解决办法。由于判断一个图 G 是否为 Hamilton 图不是个简单问题，故我们不管它是否为 Hamilton 图，在初始化时都按 (b) 的解决办法，统一把它们补全成完全图 K_n。

对于巡视问题，图 G 有 19 个节点，故应把它初始化成 K_{19}。由于 19 个节点在原问题中有它特定的名字，为方便图 G 的矩阵表示和计算机处理，把它们对应到一些整数，对应关系如下：

$$
\begin{array}{ccccccccccccccccccc}
\bigstar & M & P & 28 & 27 & 26 & 25 & N & 24 & 20 & 21 & 23 & K & 22 & 18 & 17 & I & 16 & 15 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18
\end{array}
$$

于是，可用边权矩阵 $D = (D_{ij})_{19\times 19}$ 表示 G，不相邻点间的边权用计算机最大的字长表示[1]。有了矩阵 D 后，使用 Warshall 算法求解任意两点间的最短路径，Warshall 算法如下 [3]：

（1）输入 D；

（2）$k := 1$；

（3）$i := 1$；

（4）$d_{ij} := \min(d_{ij}, d_{ik} + d_{kj})$，$j = 1, 2, \ldots, n$；

（5）$i := i + 1$，若 $i \le n$，转（4）；

（6）$k := k + 1$，若 $k \le n$，转（3）；否则停止.

之所以使用 Warshall 算法，首先是因为它的性能较好，它的时间复杂度为 $O(n^3)$；另外，在求解当中，我们并不关心两个非相邻节点的间最短路径具体如何走，我们只关心其路径的大小。故 Warshall 算法满足我们的需求，在程序**sh.c**中函数 $warshall()$ 是算法的具体实现。由 $warshall()$ 得到 K_{19} 的边权矩阵 D'，初始化工作完成。

§4. 求解 STSP

[1]在程序中，为加快求解速度，我们始终使用整数运算，所以使 G 的边权都乘 10，使它变成整数，直到最后才把最终结果化成浮点数。在文章中，我们则使用原本的数据。

接下来我们就是要在 K_n 中寻找最短的 Hamilton 回路，问题归结为求解对称的推销员问题（STSP: Symmetric Travelling Salesman Problem）。求解 STSP（以下简称 TSP）的方法有两种：一种是求近似最优解；另一种是求精确最优解。限于时间，作者编出的程序只使用了近似求解方法，至于精确求解，使用的是现成的工具*tspsol*[4]。事实上，对于该巡视问题，用近似算法就可以求出最优解。

§4.1. 启发式算法（Heuristics）

启发式算法是近似算法（Approximation Algorithms），它不一定能得出问题的精确最优解，但它却有很高的性能，能在可接受的时间内算出复杂问题（例如有成千上万个节点的 TSP）的近似最优解，使用此类算法得出的解应有一定的品质保证（Performance Guarantee）。大量的实验表明，用好的启发式算法求出的解在只有 2 – 3% 的情况出现较大地偏离最优解 [1]。

本文使用两种求解 TSP 的常用启发式算法，这两种方法是结合在一起使用。首先，我们使用搜刮方法（Scratch Methods）粗略地找出一个近似解作为初始回路，这些初始解可达到最优解的期望为 10%[2]；然后，使用改良方法（Improvement Method），或者叫后续优化（Post-optimization），对初始解进行多次的改良，直到无法改良为止，所得到的就是最好的近似解。在实际操作中，由于启发式算法的求解速度较快，所以我们会尝试从多个不同的方向出发，求出多个近似解，然后从中挑出最好的一个作为最终解。

使用搜刮法得出初始解，往往会关心这些初始解的品质如何，因为好的初始解会减少改良时所花费的时间。为列举几种常用搜刮算法的性能，我们引入一些记号和概念：设 T 是用某算法产生的一个回路，它的长度记为 $\ell(T)$，而问题的精确最优解记为 ℓ_{opt}；图 $G<V,E>$ 的边权满足所谓的三角不等式是指：

$$\ell(v_i,v_k)\leq \ell(v_i,v_j)+\ell(v_j,v_k)\quad \forall v_i,v_j,v_k\in V \tag{1}$$

其中 $\ell(v_i,v_j)$ 是指连接 v_i 和 v_j 的边的权。

- **最近邻算法（Nearest Neighbour）**

最近邻算法是最简单的启发式算法，但它的性能是最差的。它的过程是这样的：从任意选取的一个节点 $v_1\in V$ 开始。到达某一阶段，得到序列 $(v_1,v_2,\ldots,v_k)$，如果 $k=n$，则得到回路 $T=(v_1,v_2,\ldots,v_n,v_1)$，搜索结束；否则，我们找到与 v_k 最近邻的且不在序列当中节点 v_{k+1}，把它添加到序列中，令 $k:=k+1$，重复上述过程。

最近邻算法是一种贪心算法，当图 G 满足三角不等式 (1) 时，最近邻算法在最好的情况下可以使得 $\ell(T)/\ell_{opt}=(\log_2 n)/3$，在一般情况下，该算法也可保证 $\ell(T)/\ell_{opt}\leq(\log_2 n)/2$。[2]

回到巡视问题的图 K_{19}，可以检验该 K_{19} 的边权是满足三角不等式 (1) 的。我们用最近邻算法找出近似解 T，在最好情况下有 $\ell(T)/\ell_{opt}=(\log_2 19)/3=1.416$，一般情况下有 $\ell(T)/\ell_{opt}\leq(\log_2 19)/2=2.124$，我们再看看其他搜刮法的性能 [2]：

Scratch Method	Performance Guarantee
最近插入（Nearest Insertion）	$\ell(T)/\ell_{opt}\leq 2$
Double Tree	$\ell(T)/\ell_{opt}\leq 2$
Christofide’s Heuristic	$\ell(T)/\ell_{opt}\leq 3/2$

由此可见，对于只有 19 个节点的巡视问题，最近邻算法的性能与上述更优的算法相比，差别并不大，在最好情况时甚至可媲美 Christofide 算法——目前最好的搜刮算法 [2]。所以，在知道最近邻算法对

该巡视问题的效果并不差的情况下，我们当然就选用最近邻算法，因为它的时间复杂度比上述的三种算法要低。当然，这只是具体问题具体分析，但当 n 继续增大时，最近邻算法的表现将会非常差劲。

在程序**sh.c**中，函数 *init_cycle*() 是最近邻算法的实现。我们分别从 19 个节点出发，用最近邻算法分别得出 19 条初始路线，但事实只有 17 条是不同的，最短的一条是：

$$27 \to 28 \to 26 \to P \to \bigstar \to M \to 25 \to 20 \to 21 \to K \to 18 \to I \to 15 \to 16 \to 17 \to 22 \to 23 \to N \to 24$$

它的长度为 192.7，这比我们观察得到的结果 185.1 还差一点，显然不是最优解。因此，还要继续用改良圈法求解。

- **2 次逐边法**（$2-opt$）**与 3 次逐边法**（$3-opt$）

k 次逐边法（$k-opt$）的思想是从一个回路中的 n 条边（假设其有 n 个节点）中任意取出 k 条边构成 k 元集 $(v_1v_2, v_3v_4, \ldots, v_{2k+1}v_{2k+2})_i$，显然，这些 k 元集共有 $\binom{n}{k}$ 个，所以 $1 \le i \le \binom{n}{k}$。对每个 k 元集，检查回路是否能从它的变换中得到改良。通常，k 会取 2 或 3，在巡视问题中，我们分别都用 2 和 3 次逐边法求解，最终获得的最优解是相等的，故在这个问题上，3 次逐边不见得要比 2 次逐边要好。

以下具体讨论 2 次和 3 次逐边法的变换。由最近邻算法，我们已经得到回路 T，从 T 中任意取出 2 条边 v_1v_2 和 v_3v_4 构成二元集 $(v_1v_2, v_3v_4)_i$，$1 \le i \le \binom{n}{2}$，不妨把 T 记成 $(v_1, v_2, P_1, v_3, v_4, P_2, v_1)$，其中 P_1 是路径 $v_2 \to v_3$ 中依次经过的点构成的序列（不包括 v_2、v_3），同理，P_2 是 $v_4 \to v_1$ 间的点序列，P_1 的逆序列记为 $\overline{P_1}$（即 $v_3 \to v_2$ 间的点序列）。于是，2 次逐边算法可表述为：

（1）$i := 1$；

（2）取二元集 $(v_1v_2, v_3v_4)_i$，

$\ell := \ell(v_1v_2) + \ell(v_3v_4)$，

$\ell_1 := \ell(v_1v_3) + \ell(v_2v_4)$，

如果 $\ell > \ell_1$，则转到（3）；否则转到（4）；

（3）T 变换成 $(v_1, v_3, \overline{P_1}, v_2, v_4, P_2, v_1)$，转到（1）；

（4）如果 $i == \binom{n}{2}$，则结束；否则，$i := i+1$，转到（2）.

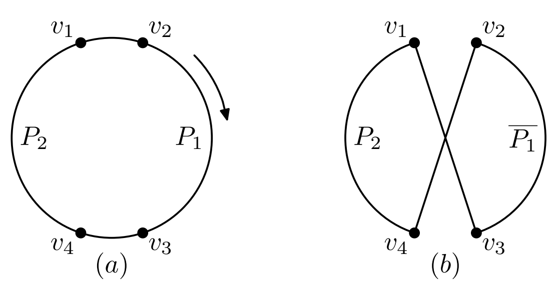

图 2 从（a）到（b）是 $2-opt$ 执行的一次变换.

同样道理，3 次逐边的算法是类似的，只是在 3 次逐边中，对同一个三元集，要检查它的三种变换（如图 3 所示），但对于 STSP，由于边的方向不影响求解，所以变换（b）与（d）是等价的，取其中的任一个即可。类似 $2-opt$ 的描述，$3-opt$ 的算法为：

（1）$i := 1$；

（2）取三元集 $(v_1v_2, v_3v_4, v_5v_6)_i$，

$\ell := \ell(v_1v_2) + \ell(v_3v_4) + \ell(v_5v_6)$,

$\ell_1 := \ell(v_1v_5) + \ell(v_4v_2) + \ell(v_3v_6)$,

$\ell_2 := \ell(v_1v_4) + \ell(v_5v_2) + \ell(v_3v_6)$,

$\ell_3 := \ell(v_1v_3) + \ell(v_2v_5) + \ell(v_4v_6)$,

如果 $\ell_1 == \min(\ell, \ell_1, \ell_2, \ell_3)$，则转到（3.1），

如果 $\ell_2 == \min(\ell, \ell_1, \ell_2, \ell_3)$，则转到（3.2），

如果 $\ell_3 == \min(\ell, \ell_1, \ell_2, \ell_3)$，则转到（3.3），

否则转到（4）；

（3.1）T 变换成 $(v_1, v_5, \overline{P_2}, v_4, v_2, P_1, v_3, v_6, P_3, v_1)$，转到（1）；

（3.2）T 变换成 $(v_1, v_4, P_2, v_5, v_2, P_1, v_3, v_6, P_3, v_1)$，转到（1）；

（3.3）T 变换成 $(v_1, v_3, \overline{P_1}, v_2, v_5, \overline{P_2}, v_4, v_6, P_3, v_1)$，转到（1）；

（4）如果 $i == \binom{n}{3}$，则结束；否则，$i := i + 1$，转到（2）.

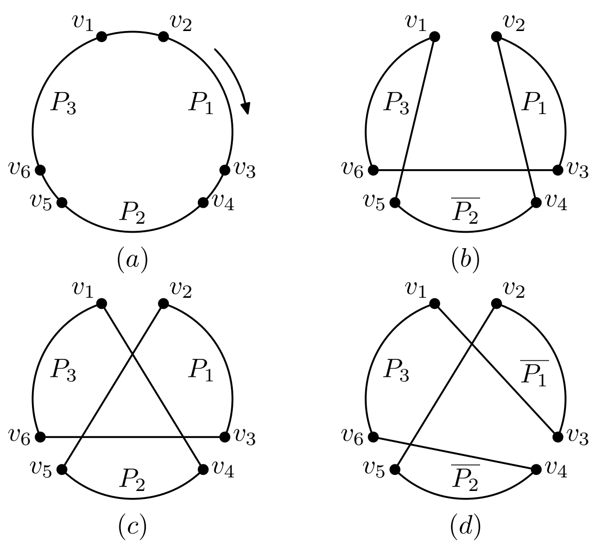

图 3 （b）、（c）、（d）分别是 $3-opt$ 的三种变换.

上述算法是程序设计中主要的思路，至于具体实现，还有一些细节需要处理，例如算法中的变换，需要对点序列 P 进行取逆 $\overline{P}$，程序**sh.c**中的函数 *array_inv*() 是该操作的实现，虽然不难，但在数组上操作时还是要小心处理的。程序**sh.c**中的函数 *two_opt*() 和 *three_opt*() 分别是对 2 次逐边法和 3 次逐边法的实现。

最终我们得到巡视问题的 K_{19} 的一条近似最短 Hamilton 回路 T_m：

$$\bigstar \to P \to 28 \to 27 \to 26 \to N \to 24 \to 23 \to 22 \to 17 \to 16 \to 15 \to I \to 18 \to K \to 21 \to 20 \to 25 \to M$$

且 $\ell(T_m) = 185.1$，这与我们的估计是一样的。但是，这并不一定是最有解，所以，最后我们使用现成的软件*tspsol*求得精确最优解来证实我们的估计与检验近似算法的性能，结果，我们的估计结果是正确的，近

似算法解出的也是最优解。

§4.2. 精确算法（Exact Algorithms）

限于时间，作者未能实现 TSP 的精确算法（事实上，对于本巡视问题不必使用精确算法也可求出最优解），但对在最后使用到的工具*tspsol*，简单介绍一下其原理：对图 $K_n < V, E >$，W 是点集 V 的一个真子集，引入记号：

$$d(W) = \{(i,j)|\ i \in W \text{且} j \notin W \text{或者} i \notin W \text{且} j \in W\}$$

即 $d(W)$ 是有且只有一个端点属于 W 的边集，若 $W = v$，我们简单记成 $d(v)$。结果 TSP 可等价为解以下 $0-1$ 整数规划问题：

$$\begin{aligned}
\text{minimize:} \quad & \sum_{i,j \in V} \ell(i,j) x_{ij} && && (2)\\
\text{subject to:} \quad & \sum_{(i,j) \in d(v)} x_{ij} = 2 && \forall v \in V && (3)\\
& \sum_{(i,j) \in d(W)} x_{ij} \geq 2 && \forall W \subset V,\, W \neq \phi \text{且} W \neq V && (4)\\
& x_{ij} = 0, 1 && \forall i, j. && (5)
\end{aligned}$$

$0-1$ 变量的意义是：若边 $(i,j) \in E(T)$（T 为 Hamilton 回路，$E(T)$ 是它的边集），则 $x_{ij} = 1$，否则 $x_{ij} = 0$。约束（3）是度约束，因为对于一 Hamilton 回路 T，其每一个节点的度都为 2；约束（4）是排斥约束，它是用来排除子回路的。

对于上述整数规划问题的具体解法，是精确算法讨论的问题，限于水平和时间，本文不进行讨论，可参考 [5 ,6]。

§5. 最后处理

上述两种方法求得的最短回路 T_m，是巡视问题的图 K_{19} 的最短 Hamilton 回路，问题最终结果应回到它原本的图 $G < V, E >$ 上。转换方法很简单：在 $T_m = (\ldots, v_i, v_{i+1}, \ldots)$ 中，若 $v_i v_{i+1} \in E$，则不需作任何替换；否则，将 $v_i v_{i+1}$ 换成 v_i、v_{i+1} 间的最短路径，因为这需要最短路径的具体走法，所以可用 Dijkstra 算法求出。对于 T_m，与我们在开始时估计的一样，只需将 $v_{16} \to v_{15}$ 替换成 $v_{16} \to v_I \to v_{15}$ 即可。由于转换很简单，故没有给出 Dijkstra 算法的具体实现。

§6. 结论

对于巡视问题，使用近似算法也可获得精确最优解。从另一方面看，本文并没有使用特别优秀的启发式近似算法也可获得问题的完美解决，有可能是问题本身的复杂性并不大，或数据还不够典型而导致的——通过观察也大概知道最优解。因此，不应满足于本问题的解决，我们需要更多更复杂的问题或数据来检验该启发式算法的性能。

参考文献

[1] Answers.com: *http://www.answers.com/topic/travelling-salesman-problem*.

[2] Luis A. Goddyn: *TSP Heuristics*, Math 408, CA, 2004.

[3] 殷剑宏，吴开亚：图论及其算法，中国科学技术大学出版社，合肥，2005.

[4] TSPLIB95: *http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/*.

[5] Y Narahari: *Optimal Solution for TSP using Branch and Bound*, Data Structures and Algorithms, Bangalore, 2006.

[6] Jens Clausen: *Branch and Bound Algorithms–Principles and Examples*, *http://www.imada.sdu.dk/Courses/DM85/TSPtext.pdf*, Denmark, 2006.

```
/*----------------sh.c--------------------*/
#include<stdio.h>
#define MAX 65535
#define INTWORD unsigned short int

char name[19][10] =
    { {"\\bigstar"}, {"M"}, {"P"}, {"28"}, {"27"}, {"26"}, {"25"},
{"N"}, {"24"}, {"20"}, {"21"}, {"23"}, {"K"}, {"22"}, {"18"},
{"17"}, {"I"}, {"16"}, {"15"}
};

inline void array_inv(INTWORD start, INTWORD end, INTWORD n, INTWORD a[])
{
    INTWORD c, k, tmp;
    c = ((end <
          start) ? (n - start + end + 1) / 2 : (end - start + 1) / 2);
    for (k = 0; k < c; k++) {
        tmp = a[start];
        a[start] = a[end];
        a[end] = tmp;
        start = (start + 1) % n;
        if (end == 0)
            end = n - 1;
        else
            end = (end - 1) % n;
    }
}

inline INTWORD array_min(INTWORD n, INTWORD a[])
{
    INTWORD i, m = a[0], k = 0;
    for (i = 1; i < n; i++) {
        if (a[i] < m) {
            m = a[i];
            k = i;
        }
    }
    return k;
```

```
}

void warshall(INTWORD n, INTWORD D[][n])
{
    INTWORD i, j, k;
    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            if (D[i][k] == MAX)
                continue;
            for (j = 0; j < n; j++) {
                if (D[i][j] - D[i][k] <= D[k][j])
                    continue;
                else
                    D[i][j] = D[i][k] + D[k][j];
            }
        }
    }
}

INTWORD
init_cycle(INTWORD start, INTWORD n, INTWORD cycle[n], INTWORD D[][n])
{
    INTWORD i, j = n - 1, k, tmp, min, dis = 0;
    INTWORD a[n];
    for (i = 0; i < n; i++)
        a[i] = i;
    for (i = 0; i < n - 1; i++, j--) {
        cycle[i] = a[start];
        tmp = a[j];
        a[j] = a[start];
        a[start] = tmp;
        start = 0;
        min = D[a[j]][a[0]];
        for (k = 1; k < j; k++) {
            if (D[a[j]][a[k]] < min) {
                start = k;
                min = D[a[j]][a[k]];
            }
```

```
        }
        dis += D[a[j]][a[start]];
    }
    cycle[n - 1] = a[0];
    dis += D[cycle[n - 1]][cycle[0]];
    return dis;
}

INTWORD
two_opt(INTWORD n, INTWORD length, INTWORD cycle[n], INTWORD D[][n])
{
    //if(n<4)return length;
    INTWORD i, j, v, w, x, y, z;
  here:i = 0;
    while (i < n) {
        j = 0;
        while (j < n - 3) {
            v = (i + 1) % n;
            w = (i + 2 + j) % n;
            x = (w + 1) % n;
            y = D[cycle[i]][cycle[v]] + D[cycle[w]][cycle[x]];
            z = D[cycle[i]][cycle[w]] + D[cycle[v]][cycle[x]];
            if (y <= z) {
                j += 1;
            } else {
                length -= (y - z);
                array_inv(v, w, n, cycle);
                goto here;
            }
        }
        i += 1;
    }
    return length;
}

INTWORD
three_opt(INTWORD n, INTWORD length, INTWORD cycle[n], INTWORD D[][n])
{
```

```
  //if(n<6)return length;
  INTWORD i, j, k, v, w, t, u, x, l, scheme[4], s;
here2:i = 0;
  while (i < n) {
      j = 0;
      v = (i + 1) % n;
      while (j < n - 5) {
          k = 0;
          w = (v + 1 + j) % n;
          x = (w + 1) % n;
          while (k < n - 5 - j) {
              t = (x + 1 + k) % n;
              u = (t + 1) % n;
              scheme[0] = D[cycle[i]][cycle[v]] + D[cycle[w]][cycle[x]] +
                  D[cycle[t]][cycle[u]];
              scheme[1] = D[cycle[i]][cycle[t]] + D[cycle[x]][cycle[v]] +
                  D[cycle[w]][cycle[u]];
              scheme[2] = D[cycle[i]][cycle[x]] + D[cycle[t]][cycle[v]] +
                  D[cycle[w]][cycle[u]];
              scheme[3] = D[cycle[i]][cycle[w]] + D[cycle[v]][cycle[t]] +
                  D[cycle[x]][cycle[u]];
              s = array_min(4, scheme);
              if (s == 0) {
                  k += 1;
              } else {
                  if (s == 1) {
                      length -= (scheme[0] - scheme[1]);
                      array_inv(v, t, n, cycle);
                      l = ((t < x) ? (n - x + t + 1) : (t - x + 1));
                      array_inv((v + l) % n, t, n, cycle);
                  } else if (s == 2) {
                      length -= (scheme[0] - scheme[2]);
                      array_inv(v, w, n, cycle);
                      array_inv(x, t, n, cycle);
                      array_inv(v, t, n, cycle);
                  } else {
                      length -= (scheme[0] - scheme[3]);
                      array_inv(v, w, n, cycle);
```

```
                        array_inv(x, t, n, cycle);
                    }
                    goto here2;
                }
            }
            j += 1;
        }
        i += 1;
    }
    return length;
}

int main()
{
    INTWORD i, j, n, min;
    scanf("%d", &n);
    INTWORD D[n][n], init_length[n], cycle[n][n];
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            scanf("%d", &D[i][j]);
    }
    warshall(n, D);
    for (i = 0; i < n; i++) {
        init_length[i] = init_cycle(i, n, cycle[i], D);
    }
    for (i = 0; i < n; i++) {
        init_length[i] = two_opt(n, init_length[i], cycle[i], D);
        printf("length[%d]=%d\n", i, init_length[i]);
    }
    for (i = 0; i < n; i++) {
        init_length[i] = three_opt(n, init_length[i], cycle[i], D);
        printf("length[%d]=%d\n", i, init_length[i]);
    }
    min = array_min(n, length);
    for (i = 0; i < n; i++) {
        printf("%s--> ", name[cycle[min][i]]);
    }
    printf("\n");
```

```
    return 0;
}
```