回溯法：旅行商问题

曹桢 2014211302 学号：2014211182 班内序号：28

1. **原理**

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”。

在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。

**二、问题描述**

旅行商问题（Traveling Salesman Problem,TSP）是旅行商要到若干个城市旅行，各城市之间的费用是已知的，为了节省费用，旅行商决定从所在城市出发，到每个城市旅行一次后返回初始城市，问他应选择什么样的路线才能使所走的总费用最短？

此问题可描述如下：设G=(V,E)是一个具有边成本cij的有向图，cij的定义如下，对于所有的i和j，cij>0,若<i,j>不属于E，则cij=∞。令|V|=n，并假设n>1。 G的一条周游路线是包含V中每个结点的一个有向环，周游路线的成本是此路线上所有边的成本和。

**三、算法**

旅行商问题的解空间是一颗排列树。对于排列树的回溯搜索与生成1,2,3，……，n的所有列的递归算法Perm类似。开始时x=[1,2,…,n]，则相应的排列树由x[1:n]的所有排列构成。在backTrack函数中算法的主要流程可以归结如下：

① i=n 时，当前扩展结点是排列树的叶节点的父节点。此时算法检测图G是否存在一条从顶点x[n-1]到顶点x[n]的边和一条从顶点x[n]到顶点1的边。如果这两条边都存在，则找到一条回路。此时算法还需要判断这条回路的代价是否优于已找到的当前最优回路的代价。如果是则更新当前解。

② i<n 当 时，当前扩展结点位于排列树的第i-1层。图G中存在从顶点x[i-1]到顶点x[i]的边时，x[1:i]构成图G的一条路径，且当x[1:i]的代价小于当前的最优值时，更新最优路径，否则剪去相应的子树。

**四、算法复杂度**

若解空间树种从根节点到叶节点的最长路径的长度为h(n)，则算法的时间复杂度为O(h(n))。如果不考虑更新bestx所需的计算时间，则算法backtrack需要O((n-1)!)计算时间。由于算法backtrack在最坏情况下可能需要更新当前最优解O((n-1)!)次，每次更新bestx需O(n)计算时间，从而整个算法的计算时间复杂性为O(n!)

**五、改进**

旅行商问题事实上是一个NP问题，用回溯法暴力求解，时间复杂度较高。不过可以用动态规划的方法把时间复杂度从O(n!)降低到O (2n)。

假设从顶点i出发，令d(i,V’)表示从顶点i出发经过V’中各个顶点一次且仅一次，最后回到出发点i的最短路径的长度，开始时，V’=V-{i}，于是，旅行商问题的动态规划函数为：d(i,V’) = min{cik+ d(k,V’-{k})} (k∈V’) 1) d(k,{}) = cki(k ≠ i)2)。

核心伪代码如下: {

for (i =1;i<n;i++) //初始化第0列

d[i][0]=c[i][0];

for( j=1;j<2^(N-1)-1;j++)

for(i=1 ; i<n ;i++) {

if(子集Vj中不包含i){

对Vj中的每个元素k，计算d[i][Vj] = min{c[i][k] + d[k][{Vj-k}] | 每一个k∈Vj};

}

}

对V[2^(n-1)-1]中的每个元素k,计算:

d[0][2^(n-1)-1] = min{c[0][k] + d[k][2^(n-1)-2]};

输出最短路径:

d[0][2^(n-1)-1];

}

**六、算法实现**

class Traveling{

private:

friend int TSP(int\*\*,int\*,int,int);

void backTrack(int i);

int city;

int\* x;//当前旅行策略

int\* bestx;//最优旅行策略

int\*\* a;

int cc;//当前代价

int bestc;//存放最小代价值

int NoEdge;//不连通的标志，这里设为-1

};

void Traveling::backTrack(int i)

{

if(i == city){

if(a[x[city-1]][x[city]] != NoEdge && a[x[city]][1] != NoEdge && (cc + a[x[city-1]][x[city]] + a[x[city]][1] < bestc || bestc == NoEdge))

{

bestc = cc + a[x[city-1]][x[city]] + a[x[city]][1];

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

bestx[j] = x[j];

}

}else{

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

if(a[x[i-1]][x[j]] != NoEdge && (cc +a[x[i-1]][x[i]]<bestc || bestc==NoEdge))

{

swap(x[i], x[j]);

cc += a[x[i - 1]][x[i]];

backTrack(i + 1);

cc -= a[x[i - 1]][x[i]];

swap(x[i], x[j]);

}

}

}

int TSP(int\*\* a,int\* bestPath,int city,int NoEdge)

{

Traveling Y;

Y.x = new int[city + 1];

for(int i = 1; i <= city; i++)

Y.x[i] = i;

Y.a = a;

Y.city = city;

Y.bestc = NoEdge;

Y.bestx = bestPath;

Y.cc = 0;

Y.NoEdge = NoEdge;

Y.backTrack(2);

delete[] Y.x;

return Y.bestc;

}

**七、总结**

这次实验，我更加深刻地理解了回溯法，理解了排列树，实现了旅行商问题的求解，提高了自己的代码能力，对计算机编程的兴趣愈发浓厚。

**八、附录：源代码**

#include <iostream>

#include <cstring>

using namespace std;

class Traveling{

private:

friend int TSP(int\*\*,int\*,int,int);

void backTrack(int i);

int city;

int\* x;//当前旅行策略

int\* bestx;//最优旅行策略

int\*\* a;

int cc;//当前代价

int bestc;//存放最小代价值

int NoEdge;//不连通的标志，这里设为-1

};

void Traveling::backTrack(int i)

{

if(i == city){

if(a[x[city-1]][x[city]] != NoEdge && a[x[city]][1] != NoEdge && (cc + a[x[city-1]][x[city]] + a[x[city]][1] < bestc || bestc == NoEdge))

{

bestc = cc + a[x[city-1]][x[city]] + a[x[city]][1];

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

bestx[j] = x[j];

}

}else{

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

if(a[x[i-1]][x[j]] != NoEdge && (cc +a[x[i-1]][x[i]]<bestc || bestc==NoEdge))

{

swap(x[i], x[j]);

cc += a[x[i - 1]][x[i]];

backTrack(i + 1);

cc -= a[x[i - 1]][x[i]];

swap(x[i], x[j]);

}

}

}

int TSP(int\*\* a,int\* bestPath,int city,int NoEdge)

{

Traveling Y;

Y.x = new int[city + 1];

for(int i = 1; i <= city; i++)

Y.x[i] = i;

Y.a = a;

Y.city = city;

Y.bestc = NoEdge;

Y.bestx = bestPath;

Y.cc = 0;

Y.NoEdge = NoEdge;

Y.backTrack(2);

delete[] Y.x;

return Y.bestc;

}

int main()

{

int \*\*map;

int \* bestpath;

int n;

cin >> n;

map = new int\*[n];

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

{

map[i] = new int[n];

}

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

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

cin >> map[i][j];

}

}

bestpath = new int[n];

memset(bestpath, 0, sizeof(bestpath));

cout << "The min cost: " << TSP(map, bestpath, n-1,-1) << endl;

return 0;

}