# 算法设计与分析实验报告

# 实验三 旅行商问题

|  |  |
| --- | --- |
| **院系：** | **信息工程学院** |
| **班级：** | **15级网络工程2班** |
| **学号：** | **2015551621** |
| **姓名：** | **王 康** |
| **任课教师：** | **王 婷** |
| **成绩：** |  |

**湘 潭 大 学**

**2018年6月实验三 旅行商问题**

1. **实验内容**

分别编程实现回溯法和分支限界法求 TSP 问题的最优解，分析比较两种算法的时间复杂度并验证分析结果。

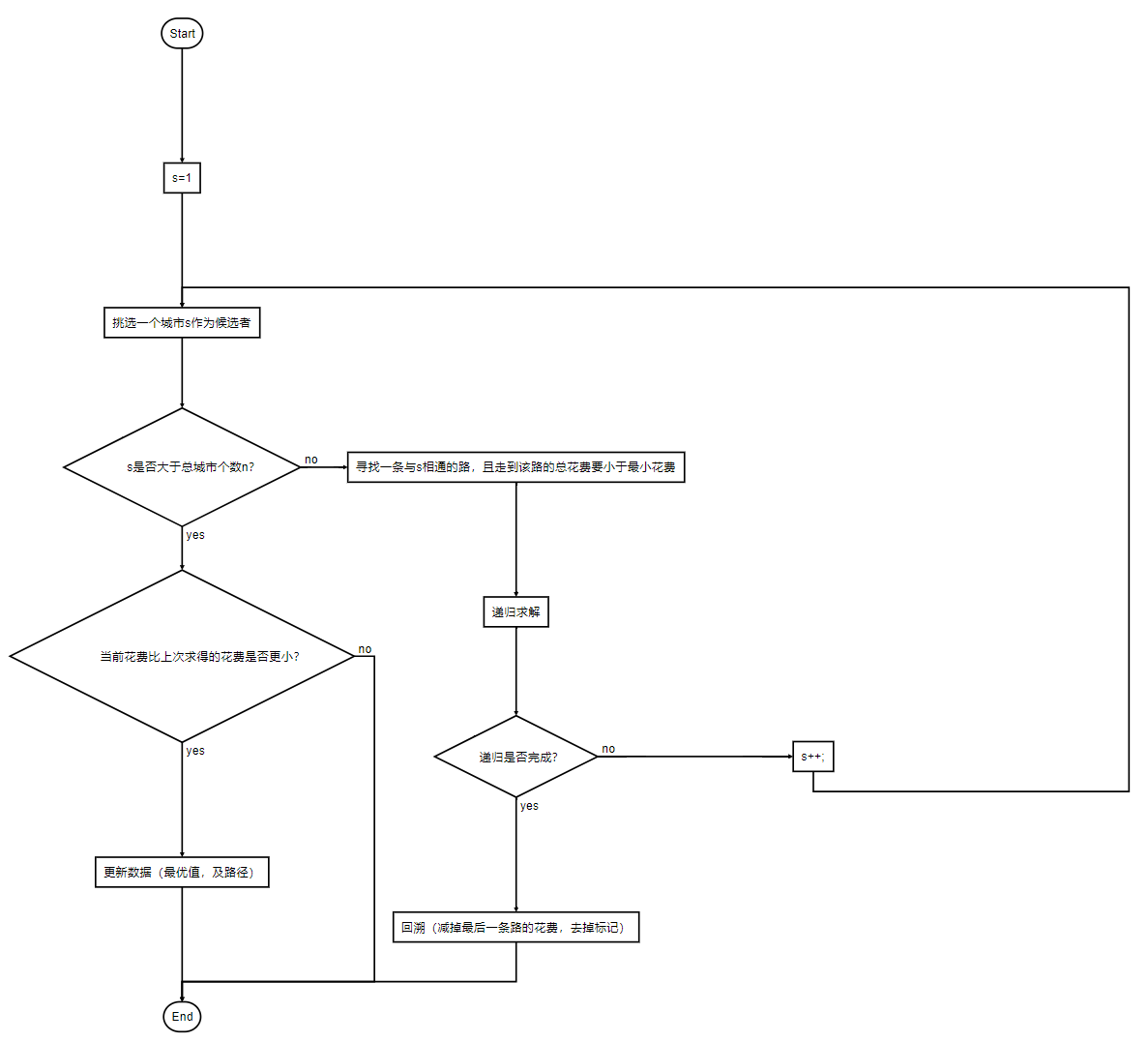
**二．实验目的**

1、掌握回溯法和分支限界法解决问题的一般步骤，学会使用回溯法和分支限界法解决实际问题；

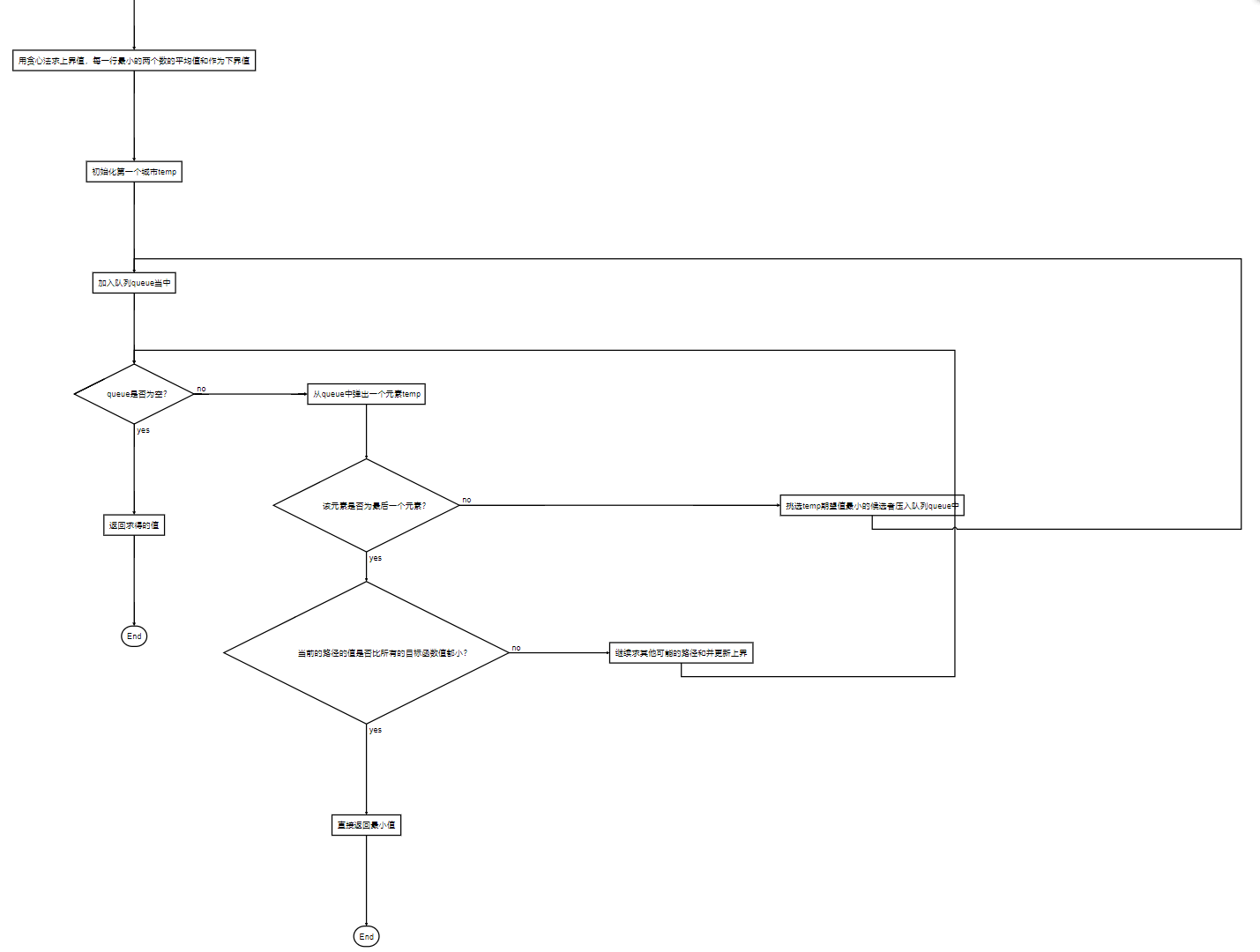
2、掌理解回溯法和分支限界法的异同及各自的适用范围。

**三. 算法描述**

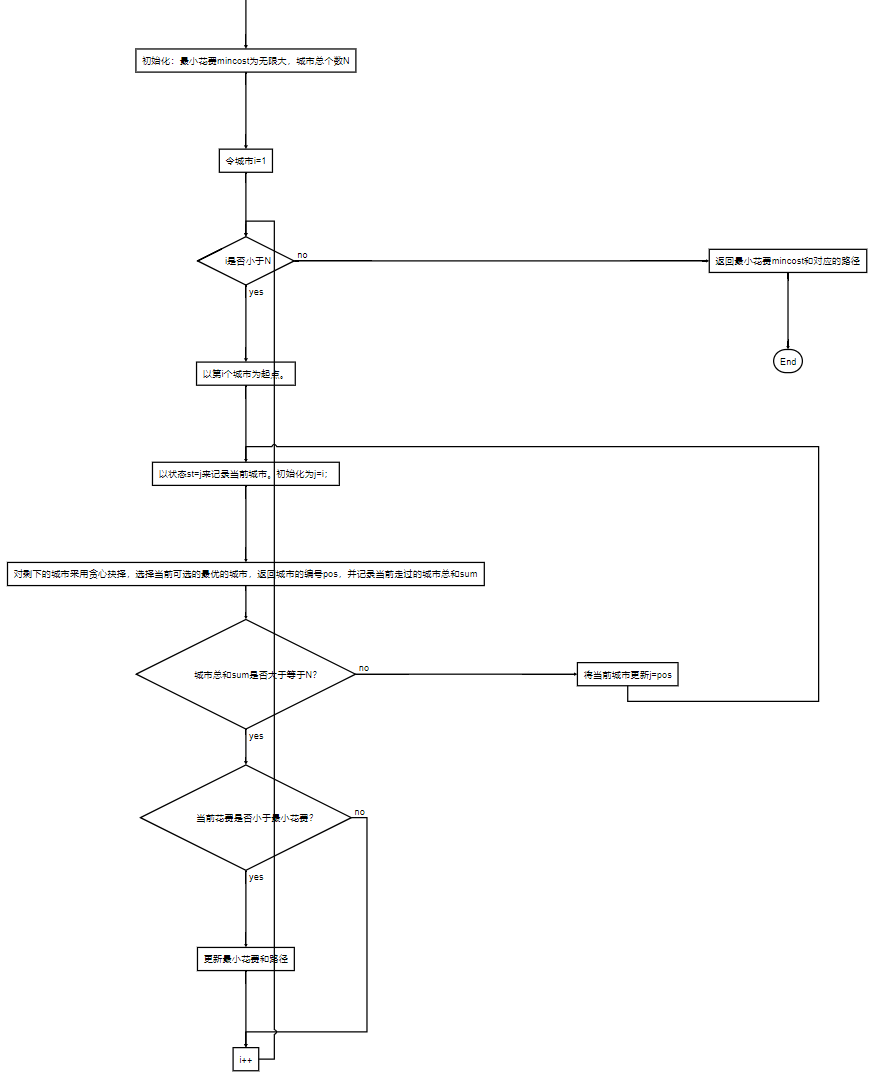
1、回溯法算法描述



1. 分支限界法算法描述



1. 贪心法算法描述（另加）



**四. 算法实现**

1. **数据结构及函数说明**

**（1）回溯法数据结构与函数说明**

**Ⅰ、数据结构**

二维数组邻接矩阵Arc储存边的信息

三个一维数组bestPath、curPath、mark分别用来存储最佳路径、当前路径和对城市做标记

**Ⅱ、函数说明**

\*\*

\* @author Swking

\* @method Tsp\_Backtrack

\* @parame

\* int t 当前测试结点

\* int n 结点个数

\* int\*\* Arc 邻接矩阵（包含边的信息）

\* int\* bestPath 最佳路径

\* int\* curPath 当前路径

\* int minCost 最小花费

\* int curCost 当前花费

\* bool\* mark 标记数组

\* @return void

\*/

void Tsp\_Backtrack(int t, int n, int\*\* Arc, int\* bestPath, int\* curPath, int &minCost, int curCost, bool\* mark)

1. **分支限界法数据结构与函数说明**

**Ⅰ、数据结构**

struct node

{

bool vis[20];//标记

int st;//路径的起点

int ed;//路径的终点

int k;//走过的点数

int sumv;//经过路径的距离

int lb;//目标函数的值

bool operator<(const node &p)const

{

return lb>p.lb;

}

};

二维数组邻接矩阵Arc储存边的信息

**Ⅱ、函数说明**

/\*\*

\* @author Swking

\* @method get\_up

\* @parame

\* int\*\* graph 邻接矩阵，边的信息（对称）

\* int n 城市（节点）个数

\* int\* uesd 记录表

\* @return mcost req城市的最短路径（req数组中第一个元素的期望值）

\*/

int get\_up(int\*\* graph, int n, int\* used)；

/\*\*

\* @author Swking

\* @method get\_low

\* @parame

\* int\*\* graph 邻接矩阵，边的信息（对称）

\* int n 城市（节点）个数

\* @return low

\*/

int get\_low(int\*\* graph, int n)；

/\*\*

\* @author Swking

\* @method Tsp\_BAB

\* @parame

\* int\*\* graph 邻接矩阵

\* int n 结点个数

\* int\* used 记录表

\* int low 下界

\* int up 上界

\* queue<int>& path 记录路径

\* @return ret 最小花费

\*/

int Tsp\_BAB(int\*\* graph, int n, int used[], int low, int up, queue<int>& path)

**（3）贪心法数据结构与函数说明**

**Ⅰ、数据结构**

struct GNode{

int node\_num;//节点编号

bool isselect;//是否被选择

}

二维数组邻接矩阵Arc储存边的信息

**Ⅱ、函数说明**

/\*\*

\* @author Swking

\* @method TSP\_Greedy

\* @parame

\* GNode\* Np 城市节点

\* int\*\* G 邻接矩阵（包含边的信息）

\* int n 结点个数

\* int\* minPath 最佳路径

\* @return mincost

\*/

int TSP\_Greedy(GNode\* Np, int\*\* G, int n, int minpath[])

1. **源程序代码**

字体：Times New Roman，字号：五号，行距：18磅，段前段后距：0行。

#include<bits/stdc++.h>

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<fstream>

#include<ctime>

using namespace std;

#define Max 9999

struct node

{

bool vis[20];

int st;//路径的起点

int ed;//路径的终点

int k;//走过的点数

int sumv;//经过路径的距离

int lb;//目标函数的值

bool operator<(const node &p)const

{

return lb>p.lb;

}

};

priority\_queue<node> q;

struct GNode{

int node\_num;

bool isselect;

};

/\*\*

\* @author Swking

\* @method TSP\_Greedy

\* @parame

\* GNode\* Np 城市节点

\* int\*\* G 邻接矩阵（包含边的信息）

\* int n 结点个数

\* int\* minPath 最佳路径

\* @return mincost

\*/

int TSP\_Greedy(GNode\* Np, int\*\* G, int n, int minpath[]){

int mincost = Max; //最短路径

for(int p=1; p<=n; p++){

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

Np[i].isselect = false; //每个节点都需要将之前的数据清除

}

int path[n+2]; //记录以srcp源点出发的路径

int srcpcost = 0; //以srcp源点出发的总开销

int sump = 1; //找到的节点总数

int srcp = p; //记录当前点

Np[p].isselect = true;

path[sump] = Np[p].node\_num;

while(sump<n){

int cost = Max; //找出最小cost的比较对象

int tempp; //记录临时点

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

if(!Np[i].isselect && G[srcp][i] < cost){

cost = G[srcp][i];

tempp = i;

}

}

sump++; //每次循环必定找到一个点

srcp = tempp; //将当前点更新成找到的点

srcpcost += cost;

path[sump] = Np[tempp].node\_num;

Np[tempp].isselect = true;

}

path[sump+1] = p;

srcpcost += G[path[1]][path[sump]]; //闭合回路的花费

if(mincost > srcpcost){

mincost = srcpcost;

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

minpath[i] = path[i];

}

}

}

return mincost;

}

/\*\*

\* @author Swking

\* @method Tsp\_Backtrack

\* @parame

\* int t 当前测试结点

\* int n 结点个数

\* int\*\* Arc 邻接矩阵（包含边的信息）

\* int\* bestPath 最佳路径

\* int\* curPath 当前路径

\* int minCost 最小花费

\* int curCost 当前花费

\* bool\* mark 标记数组

\* @return void

\*/

void Tsp\_Backtrack(int t, int n, int\*\* Arc, int\* bestPath, int\* curPath, int &minCost, int curCost, bool\* mark){

if(t>n) //当搜索完所有结点时，更新最小花费和最优路径

{

if(Arc[curPath[n]][1]!=Max && (curCost+Arc[curPath[n]][1])<minCost) //当前花费比上次求得的花费更小，则更新数据

{

minCost=curCost+Arc[curPath[n]][1]; //更新最优值

for(int j=1;j<=n;j++){ //将最优路径更新为当前求得的更好的路径

bestPath[j]=curPath[j];

}

}

}

else

{

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

{

//寻找一条与t相通的路，且走到该路的总花费要小于最小花费（否则舍弃【剪枝】），同时该结点未做标记（没有走过）

if(Arc[curPath[t-1]][j]!=Max && curCost+Arc[curPath[t-1]][j]<minCost && !mark[j])

{

curPath[t]=j; //将找到的节点加入当前路径

curCost+=Arc[curPath[t-1]][j]; //更新当前花费

mark[j]=1; //选择该路，做上标记

Tsp\_Backtrack(t+1, n, Arc, bestPath, curPath, minCost, curCost, mark); //递归求解

//求到最后，回溯到上一节点

curCost-=Arc[curPath[t-1]][j]; //减掉最后一条路的花费

mark[j]=0; //去掉标记，表示可以被选择

}

}

}

}

int get\_up\_helper(int\*\* graph, int n, int\* used, int v,int j,int len)

{

if(j==n) return len+graph[v][1];

int minlen=Max,pos;

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

{

//采用贪心算法取权值最小的边

if(used[i]==0&&minlen>graph[v][i])

{

minlen=graph[v][i];

pos=i;

}

}

used[pos]=1;

return get\_up\_helper(graph,n,used,pos,j+1,len+minlen);

}

/\*\*

\* @author Swking

\* @method get\_up

\* @parame

\* int\*\* graph 邻接矩阵，边的信息（对称）

\* int n 城市（节点）个数

\* int\* uesd 记录表

\* @return mcost req城市的最短路径（req数组中第一个元素的期望值）

\*/

int get\_up(int\*\* graph, int n, int\* used)

{

used[1]=1;

return get\_up\_helper(graph,n,used,1,1,0);

}

//部分解的目标函数的下界

int get\_lb(int\*\* graph, int n, node p)

{

int ret=p.sumv\*2;

int min1=Max,min2=Max;

//从起点到最近未遍历城市的距离

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

{

if(p.vis[i]==0&&min1>graph[p.st][i])

{

min1=graph[p.st][i];

}

}

ret+=min1;

//从终点到最近未遍历城市的距离

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

{

if(p.vis[i]==0&&min2>graph[p.ed][i])

{

min2=graph[p.ed][i];

}

}

ret+=min2;

//进入并离开每个未遍历城市的最小成本

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

{

if(p.vis[i]==0)

{

min1=min2=Max;

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

{

if(min1>graph[i][j])

min1=graph[i][j];

}

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

{

if(min2>graph[j][i])

min2=graph[j][i];

}

ret+=min1+min2;

}

}

//向上取整

return ret%2==0?(ret/2):(ret/2+1);

}

/\*\*

\* @author Swking

\* @method get\_low

\* @parame

\* int\*\* graph 邻接矩阵，边的信息（对称）

\* int n 城市（节点）个数

\* @return low

\*/

//计算下界

int get\_low(int\*\* graph, int n)

{

int low=0;

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

{

int temp[20];

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

{

temp[j]=graph[i][j];

}

sort(temp+1,temp+1+n);

low=low+temp[1]+temp[2];

}

low=low/2;

return low;

}

/\*\*

\* @author Swking

\* @method Tsp\_BAB branch and bound

\* @parame

\* int\*\* graph 邻接矩阵

\* int n 结点个数

\* int\* used 记录表

\* int low 下界

\* int up 上界

\* queue<int>& path 记录路径

\* @return ret 最小花费

\*/

int Tsp\_BAB(int\*\* graph, int n, int used[], int low, int up, queue<int>& path)

{

up = get\_up(graph, n, used);

low = get\_low(graph, n);

node start;start.st=1;start.ed=1;start.k=1;

for(int i=1;i<=n;i++) start.vis[i]=0;

start.vis[1]=1;start.sumv=0;start.lb=low;

int ret=Max;q.push(start);node next,temp;

while(!q.empty())

{

temp=q.top();

q.pop();

if(temp.k==n-1)

//如果只剩最后一个点了

{

int pos=0;

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

{

if(temp.vis[i]==0)

{

pos=i;

break;

}

}

if(pos==0) break;

int ans=temp.sumv+graph[pos][temp.st]+graph[temp.ed][pos];

node judge=q.top();

//如果当前的路径和比所有的目标函数值都小则跳出并直接输出最优解

if(ans<=judge.lb)

{

ret=min(ans,ret);

break;

}

//否则继续求其他可能的路径和并更新上界

else

{

up=min(up,ans);

ret=min(ret,ans);

continue;

}

}

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

{

if(temp.vis[i]==0)

{

next.st=temp.st;

next.sumv=temp.sumv+graph[temp.ed][i];

next.ed=i;

next.k=temp.k+1;

for(int j=1;j<=n;j++) next.vis[j]=temp.vis[j];

next.vis[i]=1;

next.lb=get\_lb(graph,n,next);

if(next.lb>=up) continue;

q.push(next);

}

}

}

return ret;

}

int main()

{

fstream fin, fout;

fin.open("Arc.txt",ios::in|ios::out);

fout.open("out.txt",ios::out);

/\*\*

\* 生成数据

\* 前三个样例手动模拟,后面为随机生成

\*/

int data1[18] = {4,

0, 2, 5, 6,

2, 0, 4, 4,

5, 4, 0, 2,

6, 4, 2, 0,

};

int data2[18] = {4,

0, 30, 6, 4,

30, 0, 5, 10,

6, 5, 0, 20,

4, 10, 20, 0,

};

int data3[38] = {6,

0, 2, 4, 5, 6, 1,

2, 0, 3, 2, 3, 7,

4, 3, 0, 1, 2, 3,

5, 2, 1, 0, 1, 3,

6, 3, 2, 1, 0, 3,

1, 7, 3, 3, 3, 0,

};

fin << data1[0] << endl;

for(int i=0; i<data1[0]; i++){

for(int j=1; j<=data1[0]; j++){

fin << data1[i\*data1[0]+j] << ' ';

}

fin << endl;

}

fin << data2[0] << endl;

for(int i=0; i<data2[0]; i++){

for(int j=1; j<=data2[0]; j++){

fin << data2[i\*data2[0]+j] << ' ';

}

fin << endl;

}

fin << data3[0] << endl;

for(int i=0; i<data3[0]; i++){

for(int j=1; j<=data3[0]; j++){

fin << data3[i\*data3[0]+j] << ' ';

}

fin << endl;

}

//随机生成数据

int exampleSum=12; //样例个数

srand(time(NULL));

while(exampleSum--){

int elemN = 5 + rand()%10; //城市个数

fin << elemN << endl;

int element[elemN][elemN];

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

for(int j=i; j<elemN; j++){

if(i==j){

element[i][j] = 0;

}else{

element[i][j] = 10+rand()%50;

element[j][i] = element[i][j];

}

}

}

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

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

fin << element[i][j] << ' ';

}

fin << endl;

}

}

fin.close();

cout << "reading the file..." << endl;

fin.open("Arc.txt", ios::in|ios::out);

cout << "Computing..." << endl;

while(!fin.eof()){

int n=0;

fin >> n;

if(n!=0){

GNode\* Np = new GNode[n+1];

int\*\* Arc = new int\* [n+1];

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

Arc[i] = new int[n+1];

}

for(int i=1; i<=n; i++){ //初始化与读取边的信息

Np[i].node\_num = i;

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

fin >> Arc[i][j];

}

}

queue<int> path;

/\*\*

\* Tsp\_Backtrack求解

\* 回溯

\*/

int bestPath[n+2]; //最优解

int curPath[n+2]; //记录当前路径

int minCost = Max; //记录最小花费

int curCost = 0; //记录当前花费

bool mark[n+1];//访问标记

//从1号结点开始

mark[1] = 1;

curPath[1] = 1;

time\_t s = clock();

Tsp\_Backtrack(2, n, Arc, bestPath, curPath, minCost, curCost, mark);

time\_t e = clock();

fout << "cityNum:" << n << endl;

fout << "Tsp\_Backtrack:"<<endl;

fout << "minCost:" << minCost << endl;

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

fout << bestPath[i] << "->";

path.push(bestPath[i]);

}

fout << 1 << endl;

fout<< "time:"<<(double)(e-s)\*1000/CLOCKS\_PER\_SEC<<"ms"<<endl;

fout << endl;

/\*\*

\* Tsp\_BAB求解

\* 分支限界

\*/

int low, up;

int used[n+1];

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

{

Arc[i][i]=Max;

used[i]=0;

}

int road;

time\_t ss = clock();

int mCost = Tsp\_BAB(Arc, n, used, low, up, path);

time\_t ee = clock();

fout << "TSP\_BAB:"<<endl;

fout << "mincost:"<< mCost << endl;

while(!path.empty()){

road = path.front();

path.pop();

fout << road << "->";

}

fout << 1 << endl;

fout<< "time:"<<(double)(ee-ss)\*1000/CLOCKS\_PER\_SEC<<"ms"<<endl;

fout << endl;

/\*\*

\* 贪心求解

\*

\*/

int minpath[n+2]; //记录路径

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

Arc[i][i] = 0;

}

time\_t sss = clock();

int mincost = TSP\_Greedy(Np, Arc, n, minpath);

time\_t eee = clock();

fout << "TSP\_Greedy:"<<endl;

fout << "mincost:"<<mincost << endl;

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

fout << minpath[i] << "->";

}

fout << minpath[n+1] << endl;

fout<< "time:"<<(double)(eee-sss)\*1000/CLOCKS\_PER\_SEC<<"ms"<<endl;

fout << "---------------------------------------" << endl<<endl;

}

}

fout.close();

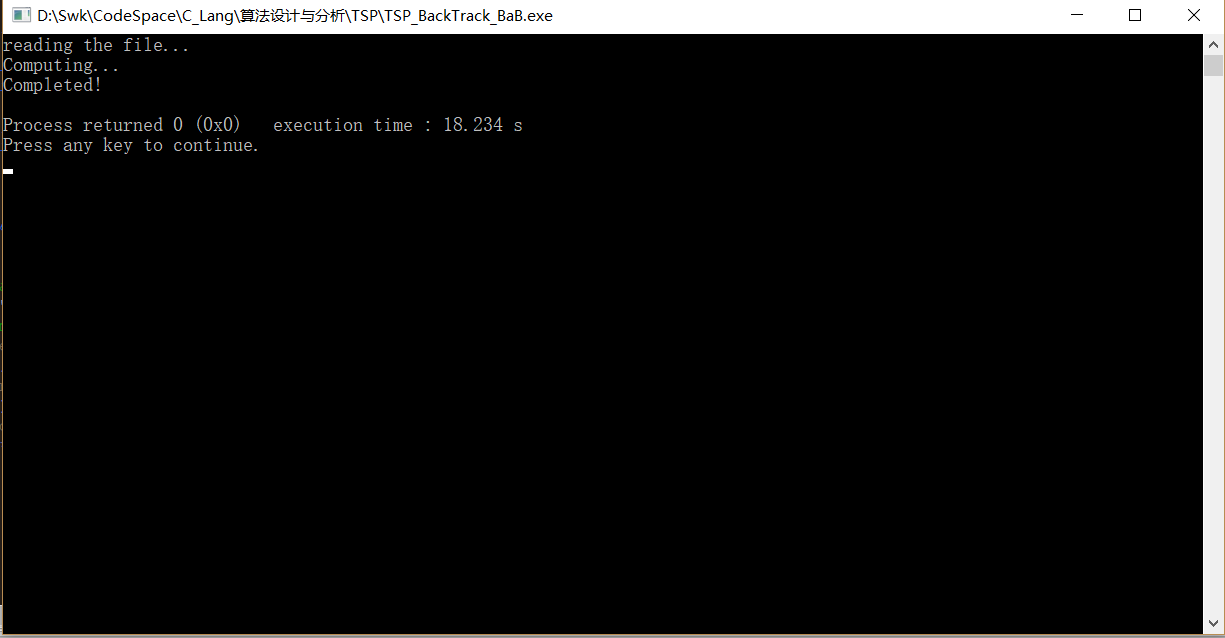
fin.close();

cout << "Completed!" << endl;

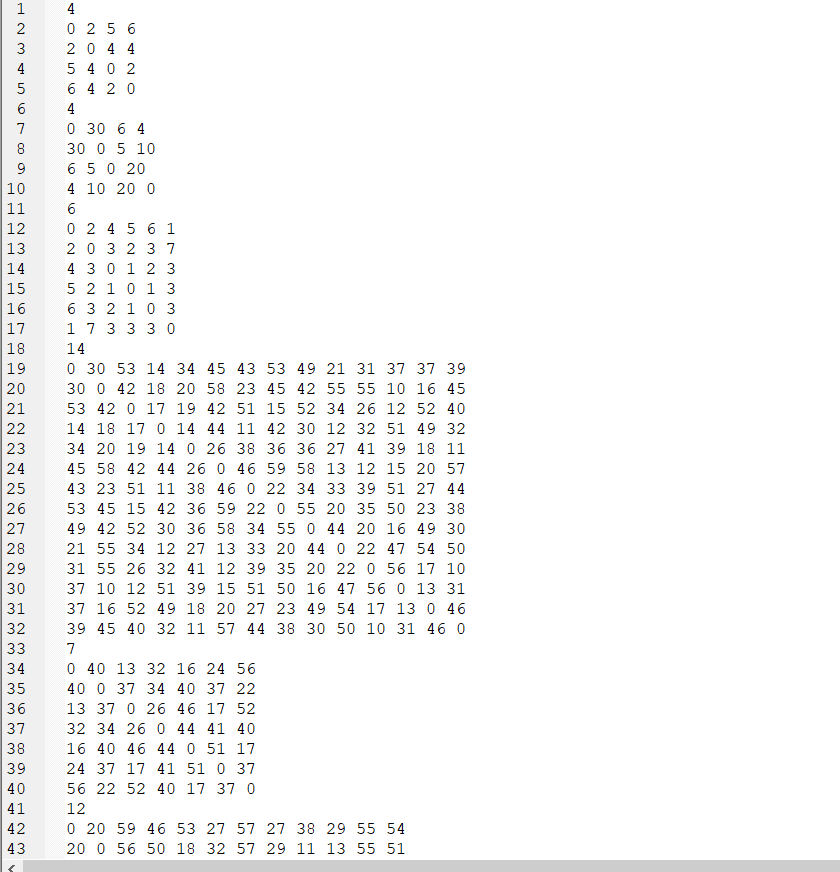
return 0;

}

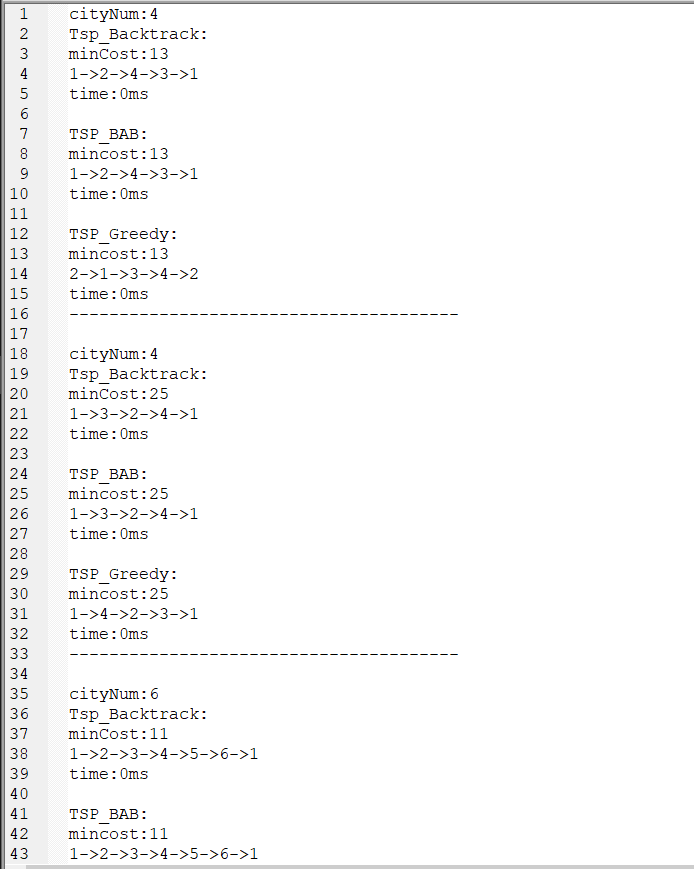
1. **程序运行结果**
2. 控制台输出信息



1. 数据文件Arc.txt前三组数据为了对比为手动模拟自动写入文件中，后面的数据为随机模拟自动写入文件。其中第一行为城市个数n，后面是城市的n\*n邻接矩阵。



1. 运行程序将结果输出到out.txt文件中，如图：以分割线分隔每个样例的结果，其中cityNum为城市个数，Tsp\_Backtrack为回溯法求得的结果，TSP\_BAB为分支限界法求得的结果，TSP\_Greedy是贪心法求得的结果。每个方法求得的结果下面分别有最小花费minCost，路径和运行时间。



1. **实验结果分析**
2. 回溯法求解TSP
   1. 贪回溯法其实是对问题的解构成解空间，每次都需要遍历到树的尾节点才往前回溯，且在未遍历完所有节点之前不能得到最优解，只有遍历完整个解空间，然后通过比较得出最优解，所以时间复杂度为：

T(n) = O(n!)

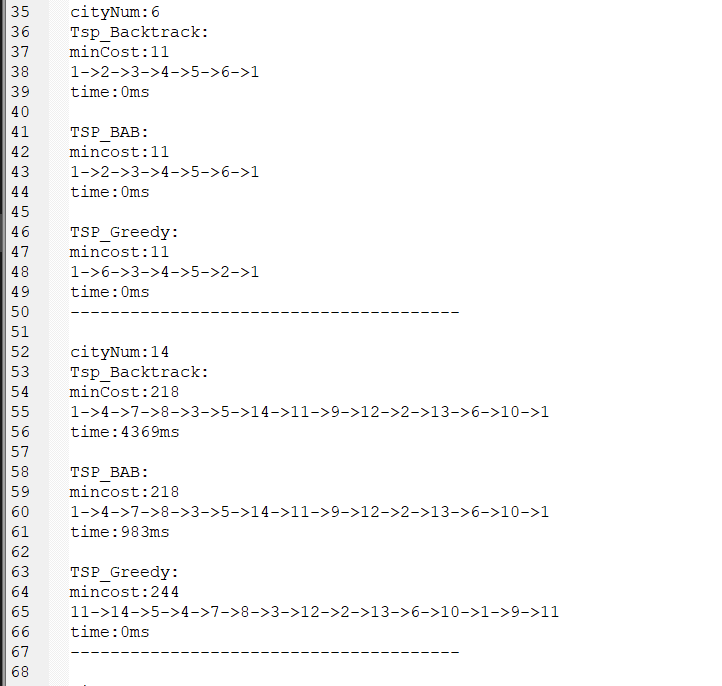
1. 分支限界法求TSP
   1. 分支同样需要遍历解空间，但是方式不同，分支限界采用广度搜索的方式寻找城市评价值最低的点进行决策，具体时间复杂度取决于采用的剪枝函数及其上界，最坏情况下时间复杂度同样要搜索整个解空间：

T(n) = O(n!)

1. 贪心法求TSP
   1. 贪心法是分别对每个节点进行一次贪心选择，选择当前最优结果后再直接封闭回路，然后选择最小花费的那条路径，所以时间复杂度为：

T(n) = O(n3)

1. 回溯法、分支限界法与贪心法求TSP的比较
   1. 其中回溯法和分支限界法能够求得最优解，但是分支限界法在求解过程中采用了剪枝函数，所以效率比回溯法要高，但分支限界法的时间复杂度还是指数级别。
   2. 贪心法运行时间最短，只有O(n3) ，但是有时候并不能求得最优解，所以贪心法不适合求TSP的精确解。



1. **结论**
2. 回溯法是将一个解空间树的过程，以深度搜索的方式对树进行遍历，当搜索到尾节点时，记录当前解，然后回溯当上一节点，一直到没有节点可选择，在搜索过程中通过比较每次搜索的解，选出最优的那一个，直至结束。所以回溯法求解TSP的时间复杂度为指数级的O(n!)，当城市大于16个以后，基本就难以求得结果。
3. 分支限界法与回溯法不同的是，它以广度优先搜索的方式对空间树进行遍历，而且每到一个候选者时，计算其评价值，通过评价值来评判是否扩展当前节点。而且分支限界法主要是在于剪枝函数的设计，好的剪枝方法可以将大量无用的解排除掉，从而提高算法效率，但时间分支限界的时间复杂度还是指数级别，但是比回溯法要好很多。
4. 贪心法是分别对每个节点进行一次贪心选择，选择当前最优结果后再直接封闭回路，然后选择最小花费的那条路径，虽然贪心法时间复杂度很低，但是很多时候是求不到最优解。我们可以用贪心法作为反之限界法里面的一个评判标准，做为其解的上界来使用，若是不要求求得精确解，贪心法效率较前两种方法是有极大的提高。