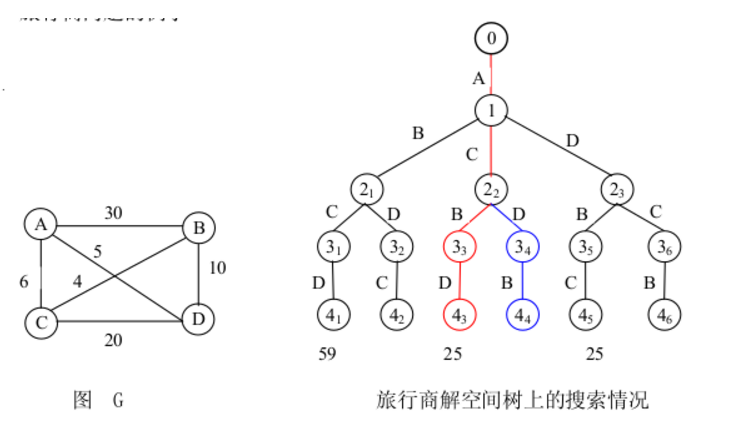
**基于回溯法的旅行商问题实验报告**

1. **算法原理**

如示例的带权图，给出这样的一个问题：“旅行商问题”常被称为“旅行推销员问题”，是指一名推销员要拜访多个地点时，如何找到在拜访每个地点一次后再回到起点的最短路径。

对于基于回溯法解决的TSP问题可以由下图简单说明：

一条边代表了选择的下一个城市，可以看到问题的本质是对解空间树的遍历搜索，找到一条这样的路径：从根结点到某一叶节点是输出路径，当且仅当这条路径耗费的代价最少。在本问题中，解空间的结点趋向N！，因此必须引入剪枝函数来减少不必要的搜索花费。

1. **问题描述**

给定一个带带权图和起点，要求输出一条这样的路径：从起点出发并返回起点，途径所有的除了起点之外的城市，并只途径一次。

要求输出路径和最小代价。

1. **算法过程分析**

本算法使用了和上一实验报告《kruskal最小生成树》相同的图结构，并引入优先队列来完成对解空间的搜索。对于优先队列的队头结点，首先出队，然后根据这个结点来判断接下来该入队的结点。

当队头结点满足这样的条件时可以输出一个候选解：此时积累的代价小于当前最小代价，此时所在的城市回到了起点，所有的城市都遍历了一遍。

如果队头结点满足下列条件，则可以不考虑入队处理，这样构成了算法的第一剪枝条件：结点的累积代价超过了当前最小代价。

对于即将入队的结点（与该结点邻接的结点），如果满足以下条件，则可以不入队，这也构成了算法的第二剪枝条件：与当前结点非邻接或此邻接点已经出现过了。

算法流程图可以简单的表示如下：

1. **算法复杂度分析**

旅行商本质是排列问题，即复杂度是O（N！），因为目的是找到一个排列，使得符合代价和最小的条件。

1. **可能的改进**

本实验所用的数据结构和算法均不够简洁，在结点入队时需要将之前的路径复制给入队结点，过程显得冗余，虽简化了寻找路径的过程，但是增大了空间上的开销。

1. **算法实现**
2. **数据结构和函数**

**注：本实验的图结构与《Kruskal算法实验报告》中的数据结构完全相同**

typedef struct MGraph{

int vexNum;

int arcNum;

int adjM[MAX\_VEXNUM][MAX\_VEXNUM];

GraphType type;

}MGraph;

struct queue\_node{

int last\_v; // 路径中上一个结点

int value; // 累积价值

int city; // 当前所在的城市

int check[MAX\_VEXNUM]; // 路径集合

};

void init( int check[],int len)

该方法初始化路径数组，使所有的结点为未经历状态。

bool all\_find(int path[],int path\_size,int start)

检查函数确保每一个城市都被遍历到且只遍历一次。

void find\_path(int path[],int path\_size,int start)

该方法为最后的输出路径算法，从最后一个结点迭代地倒推回第一个结点。

1. **算法核心流程**

用tempNode作为当前结点，问题的本质简化为带有剪枝的BFS搜索问题，是结点反复入队出队的过程。

if(tempNode.city == start\_v && all\_find(tempNode.check,path\_size,start\_v) && tempNode.value < mini\_tsp)

该条件保证了何时输出路径，即在遍历所有城市一次且回到起点的条件下，该路径成为一条候选路径。

for(int c = 0;c<=MAX\_VEXNUM-1;c++)

{

if(MG.adjM[tempNode.city][c] == MAX\_WEIGHT

|| tempNode.check[c] >= 0);

else

{

QUEUE\_NODE arr\_node;

memcpy(arr\_node.check,tempNode.check,path\_size\*sizeof(int));

arr\_node.city = c;

arr\_node.last\_v = tempNode.city;

arr\_node.value = tempNode.value + MG.adjM[tempNode.city][c];

arr\_node.check[c] = tempNode.city;

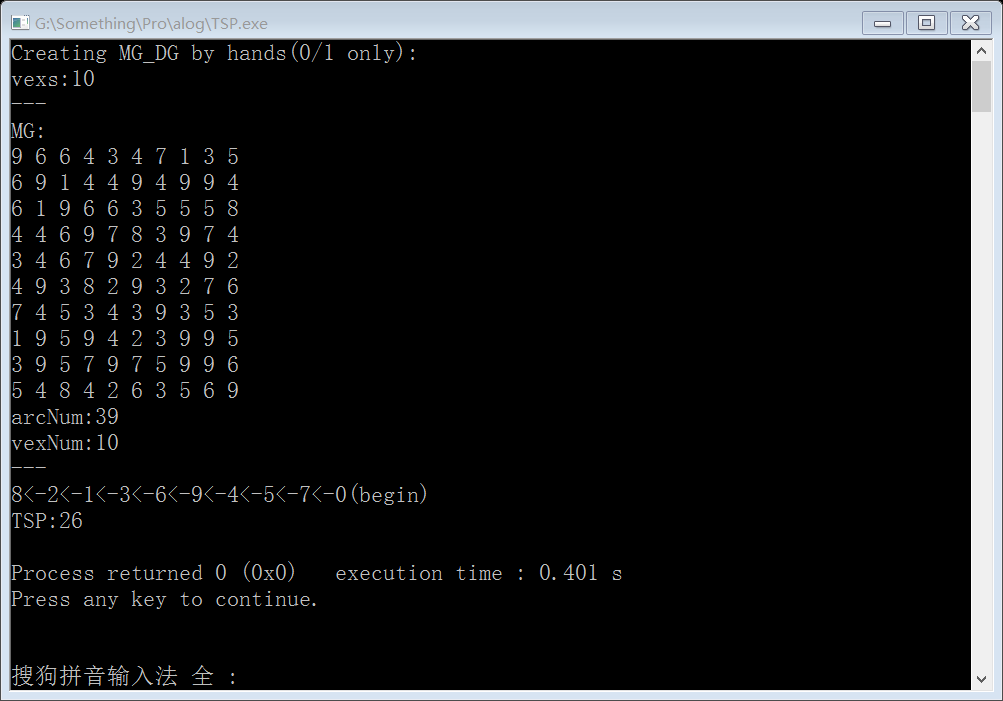
q.push(arr\_node);

}

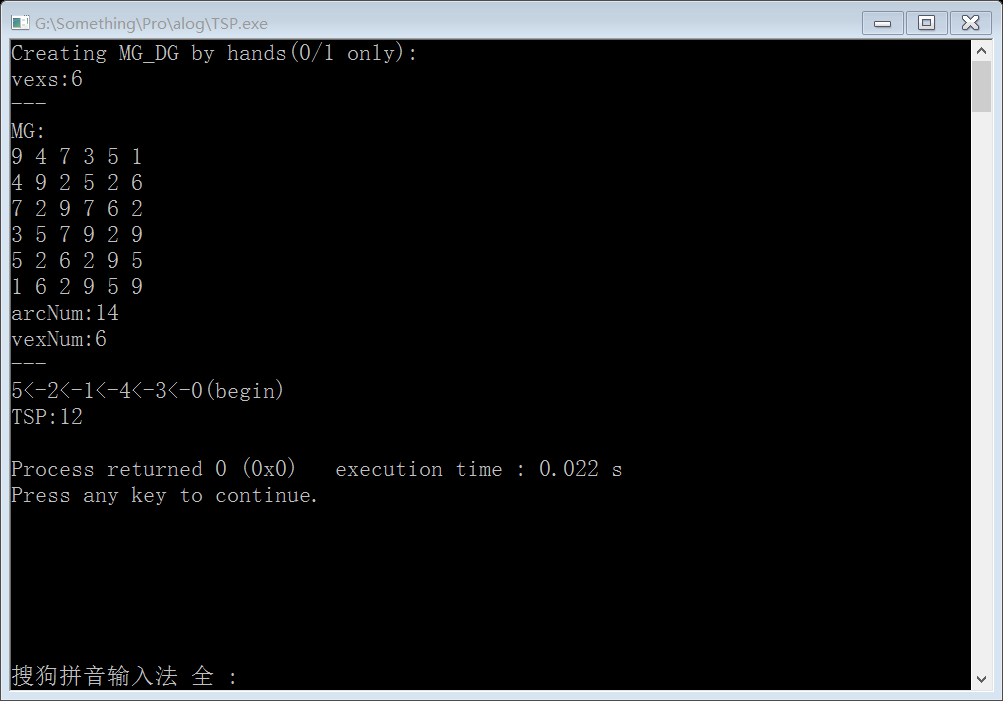
}

该循环for是结点入队的过程，所有可能入队的结点为当前结点的邻接城市，这些城市通过第二步剪枝筛选了可能入队的城市。

1. **输出结果示例与说明**



该实例给出了节点数为10的无向图，矩阵随机生成，请注意，权值为9代表两个结点不邻接，可以证明路径和最后的输出结果是正确的。



该实例给出了节点数为6的无向图，矩阵随机生成，请注意，权值为9代表两个结点不邻接，可以证明路径和最后的输出结果是正确的。

1. **总结和反思**

本次编码第一次涉足了时间复杂度为O（N！）的NP问题，在查阅资料学习的过程中发现对于该类问题的研究是积极的、富有生命力的，期间阅读到的启发式搜索、遗传算法等都是该问题的更好的解。当然，对于回溯法来说，这样的解决手段最方便，效率同时也最低下，但这不妨碍我对于TSP问题本质的认识和学习。

由于均可以抽象成图论问题，该实验的数据结构与《Kruskal算法实验报告》类似，这无疑大大加快了编码速度，可以将更多的时间投入到问题本质的研究上而不受数据结构的干扰。

#include<iostream>

#include<stdlib.h>

#include<queue>

#include<ctime>

#include<cstring>

using namespace std;

#define MAX\_VEXNUM 6

#define MAX\_WEIGHT 9

#define MAX\_VALUE 999

typedef enum{

DG,DN,UDG,UDN

}GraphType;

typedef struct MGraph{

int vexNum;

int arcNum;

int adjM[MAX\_VEXNUM][MAX\_VEXNUM];

GraphType type;

}MGraph;

bool initGraph(MGraph &MG)

{

MG.vexNum=MG.arcNum=0;

for(int i=0;i<=MAX\_VEXNUM-1;i++)

{

for(int j=0;j<=MAX\_VEXNUM-1;j++)

MG.adjM[i][j]=0;

}

}

bool createMG(MGraph &MG,int vexs)

{

int arc\_count=0;

srand(time(NULL));

cout<<"Creating MG\_DG by hands(0/1 only):"<<endl;

cout<<"vexs:"<<vexs<<endl;

MG.type = UDG;

MG.vexNum = vexs;

MG.arcNum = 0;

for(int row = 0; row <=MAX\_VEXNUM -1 ;row++)

{

for(int col = row;col <=MAX\_VEXNUM - 1;col++)

{

if(row == col)

MG.adjM[row][col] = MAX\_WEIGHT;

else

{

MG.adjM[row][col] = rand()%MAX\_WEIGHT + 1; // 随机数创建矩阵

MG.adjM[col][row] = MG.adjM[row][col];

if(MG.adjM[row][col]!=MAX\_WEIGHT)

MG.arcNum++;

}

}

}

}

struct queue\_node{

int last\_v; // 路径中上一个结点

int value; // 累积价值

int city; // 当前所在的城市

int check[MAX\_VEXNUM]; // 路径集合

};

typedef struct queue\_node QUEUE\_NODE;

void init( int check[],int len)

{

for(int i = 0;i<=len-1;i++)

{

check[i] = -1;

}

}

bool all\_find(int path[],int path\_size,int start)

{

for(int i=0;i<=path\_size-1;i++)

{

if(i == start) continue;

if(path[i]<0) return false;

}

return true;

}

int TSP(MGraph &MG,int path[],int path\_size,int start\_v)

{

queue<QUEUE\_NODE> q;

QUEUE\_NODE tempNode;

tempNode.last\_v = -1;

tempNode.value = 0;

tempNode.city = start\_v;

init(tempNode.check,path\_size);

int mini\_tsp = MAX\_VALUE;

int final\_last;

q.push(tempNode);

while(!q.empty())

{

tempNode = q.front();

q.pop();

if(tempNode.city == start\_v && all\_find(tempNode.check,path\_size,start\_v) && tempNode.value < mini\_tsp)

{

mini\_tsp = tempNode.value;

final\_last = tempNode.last\_v;

memcpy(path,tempNode.check,path\_size\*sizeof(int));

path[start\_v] = tempNode.last\_v;

//cout<<"find path"<<endl;

}

else

{

//cout<<"here"<<endl;

if(tempNode.value > mini\_tsp);

else

{

for(int c = 0;c<=MAX\_VEXNUM-1;c++)

{

if(MG.adjM[tempNode.city][c] == MAX\_WEIGHT

|| tempNode.check[c] >= 0);

else

{

QUEUE\_NODE arr\_node;

memcpy(arr\_node.check,tempNode.check,path\_size\*sizeof(int));

arr\_node.city = c;

arr\_node.last\_v = tempNode.city;

arr\_node.value = tempNode.value + MG.adjM[tempNode.city][c];

arr\_node.check[c] = tempNode.city;

q.push(arr\_node);

//cout<<"push"<<endl;

}

}

}

}

}

return mini\_tsp;

}

void printGraph(MGraph &MG)

{

cout<<"---"<<endl;

cout<<"MG:"<<endl;

for(int row=0;row<=MG.vexNum-1;row++)

{

for(int col=0;col<=MG.vexNum-1;col++)

{

cout<<MG.adjM[row][col]<<" ";

}

cout<<endl;

}

cout<<"arcNum:"<<MG.arcNum<<endl;

cout<<"vexNum:"<<MG.vexNum<<endl;

cout<<"---"<<endl;

}

void find\_path(int path[],int path\_size,int start)

{

for(int i = path[start];i != start;i = path[i])

cout<<i<<"<-";

cout<<start<<"(begin)"<<endl;

}

int main(void)

{

int tsp;

int path[MAX\_VEXNUM];

MGraph MG;

initGraph(MG);

createMG(MG,MAX\_VEXNUM); //可通过修改宏来改变顶点个数

printGraph(MG);

tsp = TSP(MG,path,MAX\_VEXNUM,0);

if(tsp != MAX\_WEIGHT)

find\_path(path,MAX\_VEXNUM,0);

cout<<"TSP:"<<tsp<<endl;

return 0;

}