“数据结构与算法”

课程设计报告

设计题目 树搜索之分支界限法

姓　　名 XX

学　　号 XXXXXXXXXX

专　　业 XXXXX

班　　级 XXXXXXXX

完成日期 XXXXXXXXXX

目录

（一） 需求和规格说明

一个 8 数码问问题是将如图 6-69（a）所示的初始格局中的小格子中的数字经过若干步移动后（只能水平和垂直移动到一个空白处）形成如图 6-69（b）所示的初始格局。

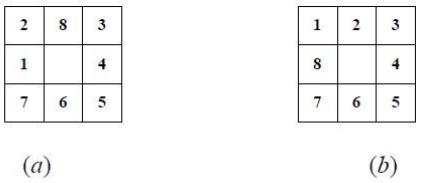


图6-69 8数码问题

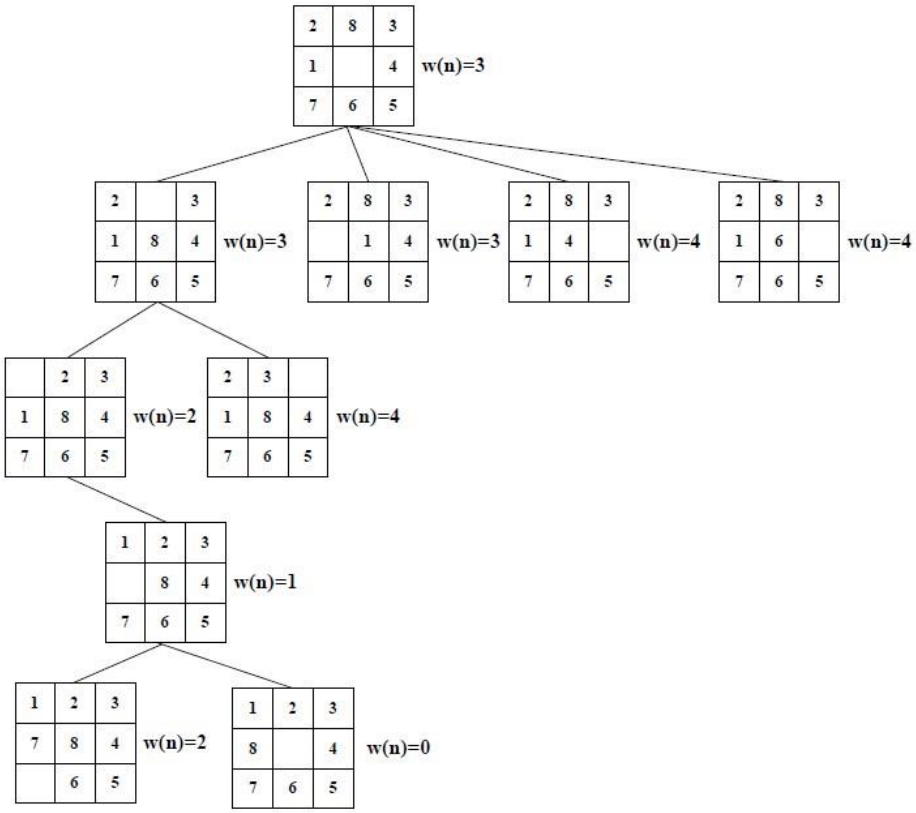


图6-70 用爬山法解决8数码问题

可用树搜索来解决 8 数码问题，如采用爬山法来解决该问题，爬山法的基本思想是根据一个评价准则贪心的选择一个节点进行所有可能的展开（枚举下一部的所有可能情况），这样处理直到找到解（到达目标格局）。对于上面的 8 数码问题，定义评价函数 w（n）=错误放置的节点个数，显然贪心准则就是选取 w（n）最小的节点展开。用爬山法解决上面的 8 数码问题的过程如图 6-70 所示。

有一点需要注意的是上面的爬山法给出的是一个解，并不一定是最优解，即移动次数最少的解，在很多情况下需要得到最优解，那么就需要将所有的节点都进行展开，即穷举搜索，此时爬山法没有任何帮助。接下来给出一种分支限界策略，该策略可以较好的解决这些优化问题，其基本思想是通过优化解的界限来修剪可行解从而减少搜索。如对于图 6-71 所示的最短路径搜索问题，可以用树搜索办法解决。首先用爬山法对该问题建立一个可行解，结果如图 6-72（a）-图 6-72（c）所示。可以求出该可行解对应的路径长度为 5，所以在展开其他节点时，如果发现了其他路径已经大于等于 5 时就不用进一步展开了，如图 6-72（d）所示，只有小于该长度的节点采用必要进一步展开，即图 6-72（d）中的阴影节点，从而修剪了许多分支，提高了算法效率这就是分支限界法的基本思想。

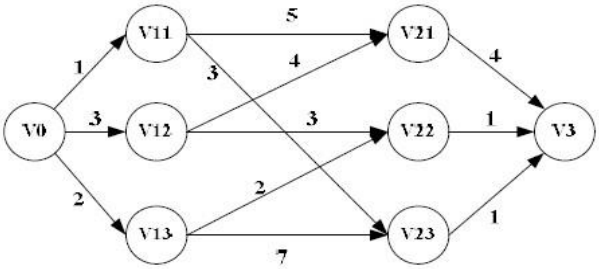


图 6-71 最短路径搜索问题

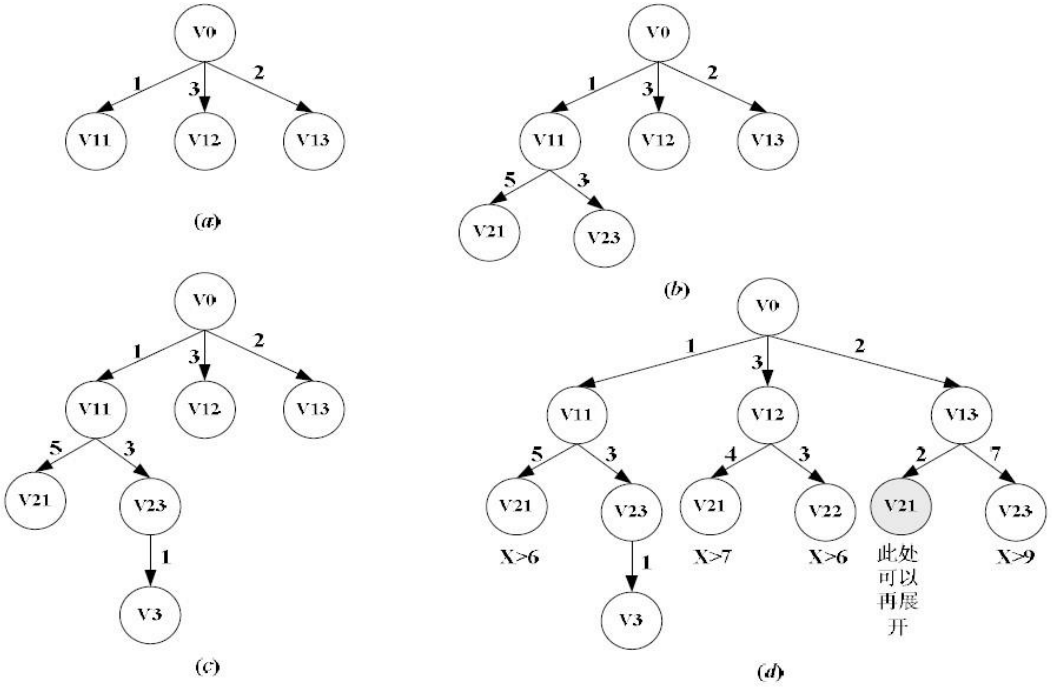


图 6-72 分支界限方法实例

可以用分支限界法解决旅行商问题（travelingsalepersonproblem 或 TSP），旅行商问题是一个在 n 个点的完全有向（或无向）加权图上寻找一个包含所有顶点的代价最小的回路，该问题已被证明是一个 NPC 问题，穷举搜索的时间复

杂性是指数函数，可以用分支限界法来避免穷举搜索，减少搜索次数。用分支限界法解决 TSP 问题的方法由两个部分组成。

① 用一种方法划分解空间（从而形成一棵搜索树）。

② 对每一类解预测其下界（需建立一种预测方法），用爬山法得到最优解的上界（即存在一个代价为次上界的解），如果某个分支的下界大于该上界，则终止这个分支。以下面的实例来说明这一过程，设一旅行商问题的代价矩阵（初始情况就是邻接矩阵）为。

	1	2	3	4	5	6	7
1	∞	3	93	13	33	9	57
2	4	∞	77	42	21	16	34
3	45	17	∞	36	16	28	25
4	39	90	80	∞	56	7	91
5	28	46	88	33	∞	25	57
6	3	88	18	46	92	∞	7
7	44	26	33	27	84	39	∞

① 分支限界法需将解分成两组：一组是包含某条特定的弧的解，另一组是不包含该弧的解。

② 按如下方法分析下界：首先注意到从代价矩阵中的任一行和任一列中减去一常数是不会改变最优解的。从该代价矩阵中的每一行分别减去 3，4，16，7，25，3，26。然后再从第 3，4 和 7 列中分别减去 7，1，4 使得代价矩阵中的每一行和每一列都至少包含一个 0。共减去的代价是 3+4+16+7+25+3+26+7+1+4=96，所以该旅行商问题解的代价的下界是 96。而经过这样处理后的代价矩阵为：

	1	2	3	4	5	6	7
1	∞	0	83	9	30	9	50
2	0	∞	66	37	17	12	26
3	29	1	∞	19	0	12	5
4	32	83	66	∞	49	0	80
5	3	21	56	7	∞	0	28
6	0	85	8	42	89	∞	0
7	18	0	0	0	58	13	∞

③ 选取将解分成两组时选取的弧是 4-6，选取 4-6 有三个原因：一是该弧代价为 0，选取包含该弧的解的代价应该较小的代价（贪心法）；另一个原因是当该弧代价为 0 时，如果解不包含该弧时，必定包含一条从 4 出发的弧和一条进入 6 的弧，选取两条代价最小的这样的弧 4-1 和

5-6，其代价为 32 和 0，所以这样解的代价的下界应该为 96+32=128；第三个原因是所有的这样的权值为 0 的弧中，不引入这条弧引起的下界的增加中选 4-6 是最大的，如选弧 3-5 进行划分，则引入的下界的增加为 1+17=18。

④ 选取弧 4-6 后，由于选取了该弧，所以从代价矩阵中删除第 4 行和第 6 列（已经找到了一条边，其他边没有用了），同时弧 6-4 也没有必要再包含了，即设置 C[6，4]= ∞。此时代价矩阵变为：

	1	2	3	4	5	7
1	∞	0	83	9	30	50
2	0	∞	66	37	17	26
3	29	1	∞	19	0	5
5	3	21	56	7	∞	28
6	0	85	8	∞	89	0
7	18	0	0	0	58	∞

此时发现第 5 行不包含 0，所以对第 5 行减去 3，所以包含弧 4-6 的解的下界应该增 3。

⑤ 用这样的方法继续处理，直到得到一个解，如图 6-73 所示。可得出该解的代价，为 126，所以所有的下界大于 126 的节点称为终止节点。

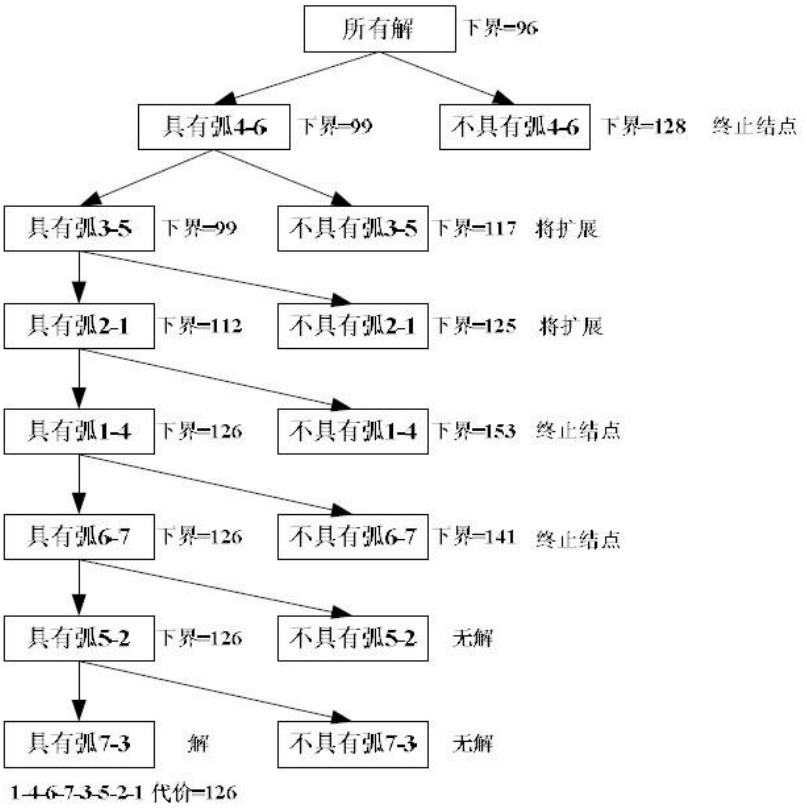


图 6-73 旅行商问题的分支限界解法过程

课程设计目的

对树搜索建立一定的认识，通过编程实现分支限界方法掌握该方法，并能用该方法解决一些实际的优化问题。

基本要求

① 针对 8 数码问题应用爬山法和分支限界法分别解决，体会二者的区别。

② 针对 TSP 问题，实现上述的分支限界法。

③ 产生一个 TSP 问题实例（多余 10 个节点），该实例可以从平面上随机产生 n 个点，权值就是两个点之间的笛卡尔距离，用分支限界法解决该问题，并输出其中间过程。

(二) 设计

八数码爬山法搜索过程：

搜索采用广度搜索方式，利用待处理队列辅助，逐层搜索（跳过劣质节点）。搜索过程如下：

① 把原棋盘压入队列；

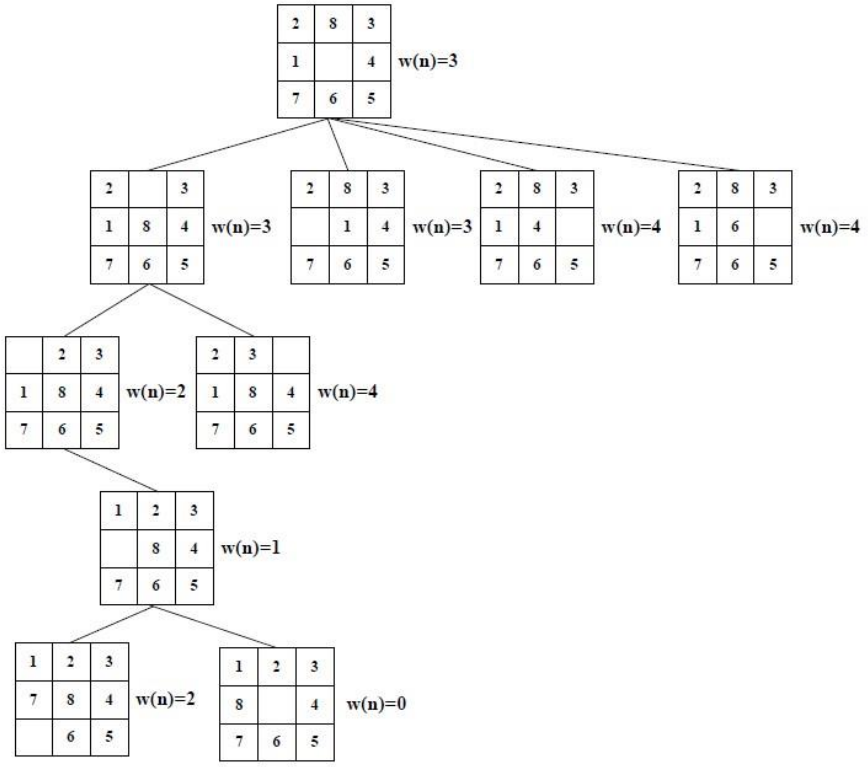
② 从棋盘取出一个节点；

③ 判断棋盘估价值，为零则表示搜索完成，退出搜索；

④ 扩展子节点，即从上下左右四个方向移动棋盘，生成相应子棋盘；

⑤ 对子节点作评估，是否为优越节点（子节点估价值小于或等于父节点则为优越节点），是则把子棋盘压入队列，否则抛弃；

⑥ 跳到步骤（2）。



八数码分支界限法搜索过程：

先用爬山法对该问题建立一个可行解，再采用广度搜索方式，利用待处理队列辅助，逐层搜索（跳过劣质节点）。搜索过程如下：

① 把原棋盘压入队列；

② 从棋盘取出一个节点；

③ 判断棋盘估价值，为零则表示搜索完成，退出搜索；

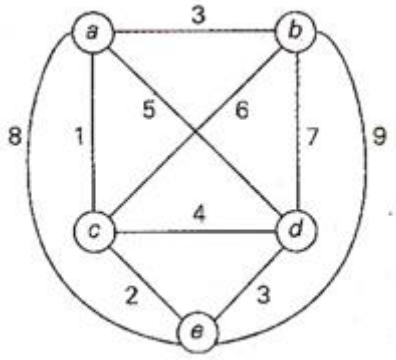
④ 扩展子节点，即从上下左右四个方向移动棋盘，生成相应子棋盘；

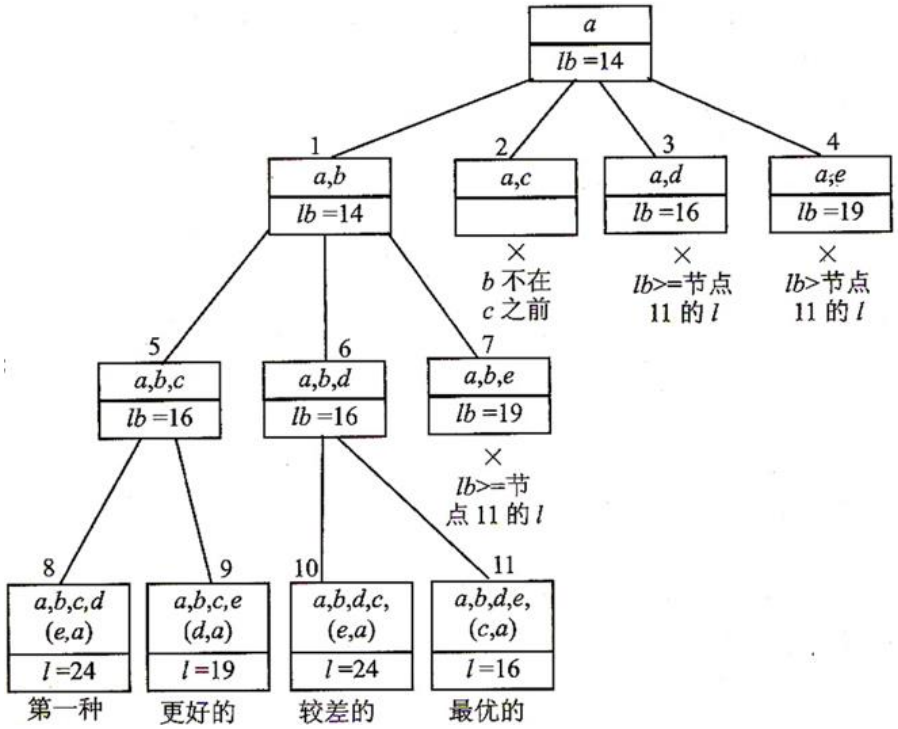
⑤ 对子节点作评估，是否为优越节点（子节点估价值小于或等于父节点则为优越节点），是则把子棋盘压入队列，否则抛弃；

⑥ 如果当前移动步数大于爬山法得到的移动步数，停止搜索，否则跳到步骤（2）。

旅行推销商问题分支界限法搜索过程：

对于 TSP，需要利用上界和下界来对 BFS 进行剪枝，下界的预测方法为题目所给的规约矩阵法，上界的选择也可以有多种方法。通过不断更新上界和下界，尽可能的排除不符合需求的子节点，以实现剪枝，最终可以获得最优的 BFS 解，以解决 TSP 问题。





八数码问题类属性和方法定义

类名	成员类别	类型	成员名	描述
Chess	属性	int[][]	cell	八数码棋盘
		int	value	评估值
		Direction	blockedDirection	屏蔽方向
		Chess *	parent	棋盘上一状态
	方法	int	evaluate(Chess *)	评价函数，返回错误的位置个数
		void	setChess()	设置棋盘
		void	printChess()	输出棋盘
		Chess *	moveChess(Direction)	移动空格

旅行推销商问题类属性和方法定义

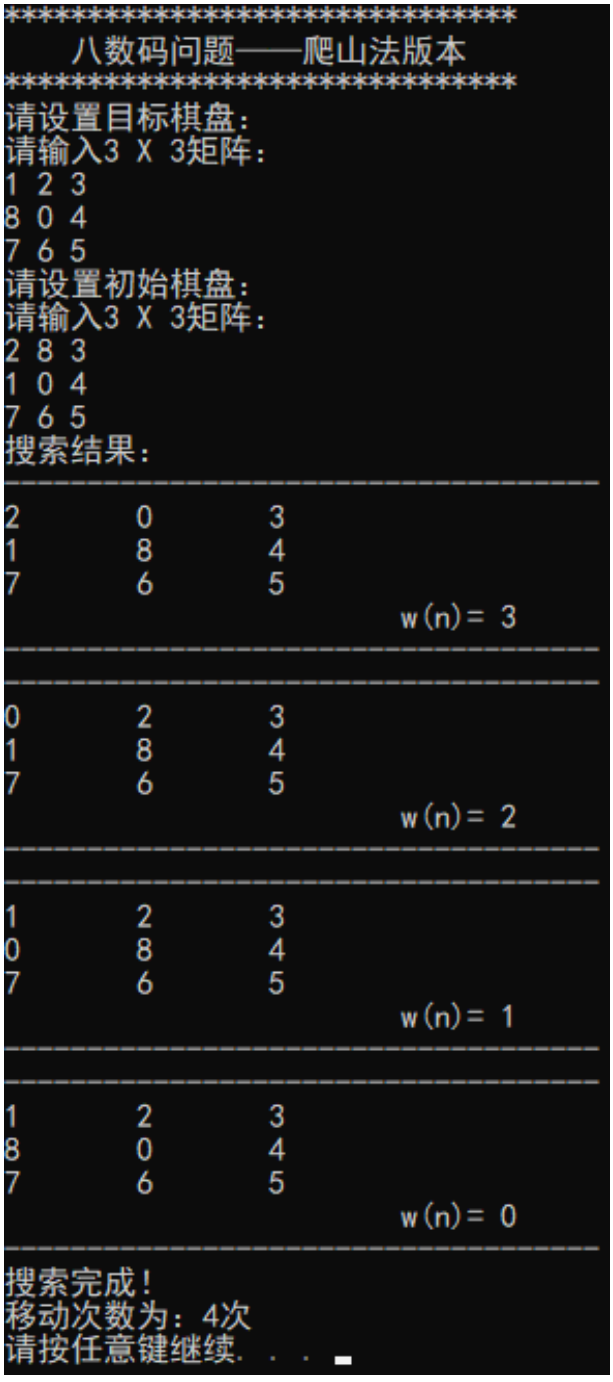
类名	成员类别	类型	成员名	描述
Node	属性	vector<vector<int>>	matrix	邻接矩阵
		vector<int>	currentPath	当前路径
		int	wage	规约代价值
		int	level	当前节点所处层数
		int	currentNode	当前节点编号
	方法	void	setChess()	设置棋盘
		void	printChess()	输出棋盘
		Chess *	moveChess(Direction)	移动空格

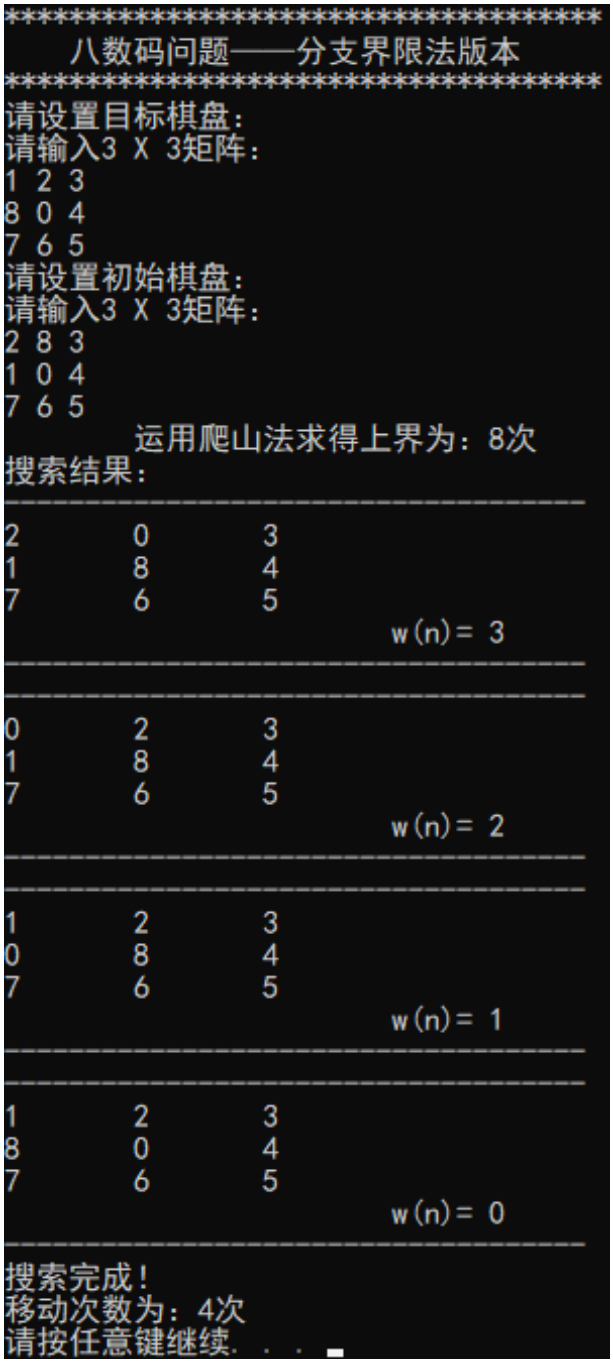
类名	成员类别	类型	成员名	描述
Travelling	属性	int	size	邻接矩阵大小
		vector<vector<int>>	matrix	邻接矩阵
		vector<int>	path	最小代价路径
		int	upperBound	上界
	方法	void	findPath()	寻找最小路径
		int	restrain(vector<vector<int>> &)	输入邻接矩阵，返回规约代价值
		void	setMatrix()	生成邻接矩阵
		void	printWage()	输出最小代价值

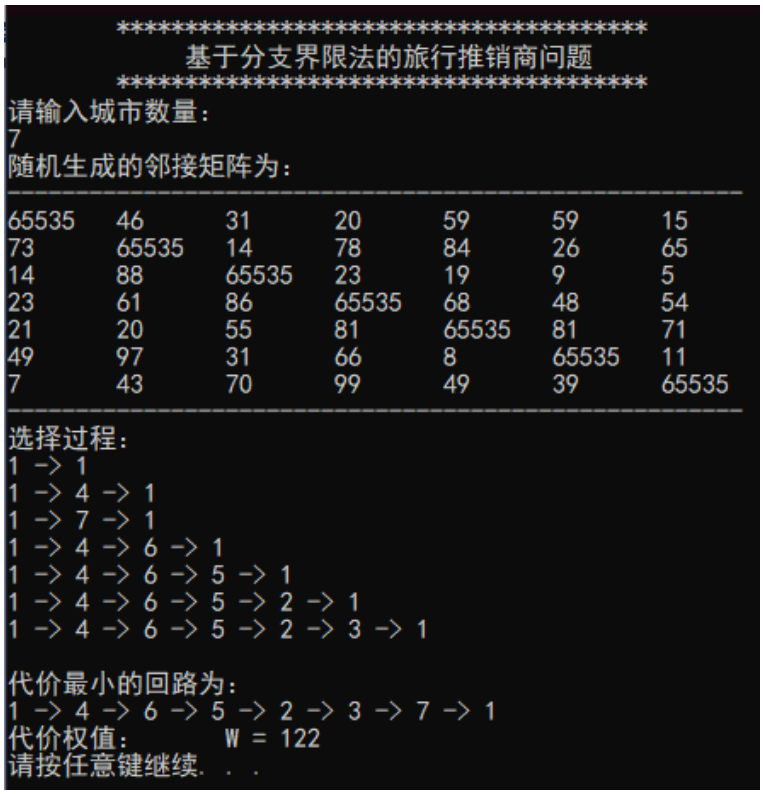
（三）调试及测试

针对八数码问题和 TSP 都采取了大量实例进行调试分析，最终程序所得结果与预期一致。

（四） 运行实例：


****
八数码问题——爬山法版本
****
请设置目标棋盘:
请输入3 X 3矩阵:
1 2 3
8 0 4
7 6 5
请设置初始棋盘:
请输入3 X 3矩阵:
2 8 3
1 0 4
7 6 5
搜索结果:
2 0 3
1 8 4
7 6 5
w(n)= 3
0 2 3
1 8 4
7 6 5
w(n)= 2
1 2 3
0 8 4
7 6 5
w(n)= 1
1 2 3
8 0 4
7 6 5
w(n)= 0
搜索完成!
移动次数为: 4次
请按任意键继续. . .


八数码问题——分支界限法版本
请设置目标棋盘：
请输入3 X 3矩阵：
1 2 3
8 0 4
7 6 5
请设置初始棋盘：
请输入3 X 3矩阵：
2 8 3
1 0 4
7 6 5
运用爬山法求得上界为：8次
搜索结果：
2 0 3
1 8 4
7 6 5
w(n)= 3
0 2 3
1 8 4
7 6 5
w(n)= 2
1 2 3
0 8 4
7 6 5
w(n)= 1
1 2 3
8 0 4
7 6 5
w(n)= 0
搜索完成！
移动次数为：4次
请按任意键继续. . .



（五） 进一步改进

① 八数码问题中，由于采用倒链表的搜索树结构，简化了数据结构，但有部分被抛弃节点的内存没有很好的处理，所以会造成内存泄漏；

② 解决八数码空格移动问题采用了屏蔽方向，有效防止往回搜索（节点的回推），但没能有效防止循环搜索，所以不能应用于复杂度较大的八数码问题；

③ 分支界限法解决 TSP 问题关于上界的计算方法还可以进一步改进。

（六） 心得体会

虽然在大二上学期中已经学习了《数据结构》课程，但是所学的知识也只有在完成作业的时候才会用到，平时缺乏锻炼的机会，本次课程设计是自己第一次通过独立构思，并且不断查阅资料来设计一项程序。这次设计，不仅巩固了对 BFS 和 DFS 算法的理解，还掌握了 C++STL 库中关于 vector 和 priority_queue 的使用技巧，深知了算法的重要性。

（七） 附录——源程序

1. 八数码问题（爬山法）

Chess.h

```
#include<iostream>
using namespace std;
const int N = 3;
//最大搜索次数
const int MAX_STEP = 50;
//空格移动方向
enum Direction{None, Up, Down, Left, Right};


class Chess
{
    friend Chess * hillClimbingSearch(Chess* begin, Chess* target,
int &depth);
    friend struct cmp;
public:
    Chess();
    ~Chess();
private:
    int cell[N][N];
    //评估值
    int value;
    //屏蔽方向
    Direction blockedDirection;
public:
    Chess * parent;
public:
    int evaluate(Chess* target);
    void setChess();
    void printChess();
    Chess * moveChess(Direction direction);
};
```

Chess.cpp

```
#include "stdafx.h"
#include "Chess.h"


Chess::Chess()
```

```
{
    value = 0;
    parent = NULL;
    blockedDirection = None;
}


Chess::~Chess()
{
}


void Chess::printChess()
{
    cout << "-----------------------------------" << endl;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout << cell[i][j] << "\t";
        cout << endl;
    }
    cout << "\t\t\tw(n)= " << value << endl;
    cout << "-----------------------------------" << endl;
}


Chess* Chess::moveChess(Direction direction)
{
    Chess *newChess;
    int i, j, t_i, t_j;
    bool hasGetBlankCell = false;
    bool ableMove = true;

    //获取空格位置
    for (i = 0; i < N; i++)
    {
        for(j = 0; j < N; j++)
            if (cell[i][j] == 0)
```

```
            {
                hasGetBlankCell = true;
                break;
            }
        if (hasGetBlankCell)
            break;
    }
    t_i = i;
    t_j = j;

    //判断空格能否移动
    switch (direction)
    {
    case Up:
        t_i--;
        if (t_i < 0)
            ableMove = false;
        break;
    case Down:
        t_i++;
        if (t_i > N)
            ableMove = false;
        break;
    case Left:
        t_j--;
        if (t_j < 0)
            ableMove = false;
        break;
    case Right:
        t_j++;
        if (t_j > N)
            ableMove = false;
        break;
    };

    //不可移动则返回原节点
    if (!ableMove)
        return this;
```

```
    newChess = new Chess();
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            newChess->cell[x][y] = cell[x][y];
    newChess->cell[i][j] = newChess->cell[t_i][t_j];
    newChess->cell[t_i][t_j] = 0;
    return newChess;
}


//评价函数，返回错误的位置个数
int Chess::evaluate(Chess* target)
{
    int error = 0;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (cell[i][j] != target->cell[i][j] && cell[i][j] != 0)
                error++;
    value = error;
    return error;
}


void Chess::setChess()
{
    cout << "请输入" << N << " X " << N << "矩阵：" << endl;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            cin >> cell[i][j];
}
```

EightDigitalCodeDmeo1.cpp

```
#include "stdafx.h"
#include "Chess.h"
#include<queue>
#include<stack>
```

```
//迭代器
struct cmp
{
    bool operator()(Chess *a, Chess *b)
    {
        return a->value > b->value;
    }
};

//爬山法搜索函数
Chess * hillClimbingSearch(Chess* begin, Chess* target, int &depth)
{
    Chess *p1, *p2, *p=NULL;
    //搜索深度
    depth = 0;
    priority_queue<Chess *, vector<Chess *>,cmp>queue;
    queue.push(begin);
    while (p == NULL || queue.empty())
    {
        //利用优先队列选取权值最小节点展开
        p1 = queue.top();
        while(!queue.empty())
            queue.pop();
        for (int i = 1; i <= 4; i++)
        {
            Direction direction = (Direction)i;

            //跳过屏蔽方向
            if (direction == p1->blockedDirection)
                continue;

            p2 = p1->moveChess(direction);
            if (p2 != p1)
            {
                p2->evaluate(target);
                if (p2->value <= p1->value)
                {
                    p2->parent = p1;
```

```
            switch (direction)
            {
            case Up:
                p2->blockedDirection = Down;
                break;
            case Down:
                p2->blockedDirection = Up;
                break;
            case Left:
                p2->blockedDirection = Right;
                break;
            case Right:
                p2->blockedDirection = Left;
                break;
            }
            queue.push(p2);

            //搜索完成，结束循环
            if (p2->value == 0)
            {
                p = p2;
                i = 5;
            }

        }
        else
        {
            delete p2;
            p2 = NULL;
        }
    }
}
depth++;

//搜索次数超过最大搜索次数
if (depth > MAX_STEP)
    return NULL;
```

```
    }
    return p;
}

int main()
{
    int depth;
    Chess *begin, *target, *t;
    begin = new Chess();
    target = new Chess();
    cout << "*******************************" << endl;
    cout << "    八数码问题——爬山法版本" << endl;
    cout << "*******************************" << endl;
    cout << "请设置目标棋盘：" << endl;
    target->setChess();
    cout << "请设置初始棋盘：" << endl;
    begin->setChess();
    begin->evaluate(target);
    t = hillClimbingSearch(begin, target,depth);
    if (t)
    {
        Chess *p = t;
        stack<Chess *>stack;
        while (p->parent != NULL)
        {
            stack.push(p);
            p = p->parent;
        }
        cout << "搜索结果：" << endl;
        while (!stack.empty())
        {
            stack.top()->printChess();
            stack.pop();
        }
        cout << "搜索完成！" << endl;
        cout << "移动次数为：" << depth << "次" << endl;
    }
    else
```

```
        cout << "搜索不到结果，搜索深度为：" << depth;
    delete begin, target;
    return 0;
}
```

2. 八数码问题（分支界限法）

Chess.h

```
#include<iostream>
using namespace std;
const int N = 3;
const int MAX_STEP = 100;
enum Direction{None, Up,Down, Left,Right};


class Chess
{
    friend Chess * hillClimbingSearch(Chess *begin, Chess *target,
int &depth);
    friend Chess * branchBoundSearch(Chess *begin, Chess *target, int
&depth);
    friend struct cmp;
public:
    Chess();
    ~Chess();
private:
    int cell[N][N];
    int value;
    Direction blockedDirection;
public :
    Chess * parent;
    int evaluate(Chess* target);
    void setChess();
    void printChess();
    Chess* moveChess(Direction direction);
};
```

Chess.cpp

```
#include "stdafx.h"
#include "Chess.h"
```

```
Chess::Chess()
{
    value = 0;
    parent = NULL;
    blockedDirection = None;
}


Chess::~Chess()
{
}


int Chess::evaluate(Chess* target)
{
    int error = 0;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (cell[i][j] != target->cell[i][j] && cell[i][j] != 0)
                error++;
    value = error;
    return error;
}


void Chess::setChess()
{
    cout << "请输入" << N << " X " << N << "矩阵：" << endl;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            cin >> cell[i][j];
}


void Chess::printChess()
{
```

```
    cout << "-----------------------------------" << endl;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout << cell[i][j] << "\t";
        cout << endl;
    }
    cout << "\t\t\tw(n)= " << value << endl;
    cout << "-----------------------------------" << endl;
}


Chess* Chess::moveChess(Direction direction)
{
    Chess *newChess;
    int i, j, t_i, t_j;
    bool hasGetBlankCell = false;
    bool ableMove = true;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
            if (cell[i][j] == 0)
            {
                hasGetBlankCell = true;
                break;
            }
        if (hasGetBlankCell)
            break;
    }
    t_i = i;
    t_j = j;
    switch(direction)
    {
    case Up:
        t_i--;
        if (t_i < 0)
            ableMove = false;
        break;
```

```
    case Down:
        t_i++;
        if (t_i > N)
            ableMove = false;
        break;
    case Left:
        t_j--;
        if (t_j < 0)
            ableMove = false;
        break;
    case Right:
        t_j++;
        if (t_j > N)
            ableMove = false;
        break;
    };
    if (!ableMove)
        return this;
    newChess = new Chess();
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            newChess->cell[x][y] = cell[x][y];
    newChess->cell[i][j] = newChess->cell[t_i][t_j];
    newChess->cell[t_i][t_j] = 0;
    return newChess;
}
```

EightDigitalCodeDemo2.cpp

```
#include "stdafx.h"
#include "Chess.h"
#include<queue>
#include<stack>

struct cmp
{
    bool operator()(Chess *a, Chess *b)
    {
        return a->value > b->value;
    }
```

```
};


//爬山法求出一个可行解
Chess * hillClimbingSearch(Chess *begin, Chess *target, int &depth)
{
    Chess *p1, *p2, *p=NULL;
    depth = 0;
    priority_queue<Chess *, vector<Chess *>, cmp>queue;
    queue.push(begin);
    while (p == NULL || queue.empty())
    {
        p1 = queue.top();
        while (!queue.empty())
            queue.pop();
        for (int i = 4; i >= 1; i--)
        {
            Direction direction = (Direction)i;
            if (direction == p1->blockedDirection)
                continue;
            p2 = p1->moveChess(direction);
            if (p2 != p1)
            {
                p2->evaluate(target);
                if (p2->value <= p1->value)
                {
                    p2->parent = p1;
                    switch (direction)
                    {
                    case Up:
                        p2->blockedDirection = Down;
                        break;
                    case Down:
                        p2->blockedDirection = Up;
                        break;
                    case Left:
                        p2->blockedDirection = Right;
                        break;
```

```
                    case Right:
                        p2->blockedDirection = Left;
                        break;
                    }
                    queue.push(p2);
                    if (p2->value == 0)
                    {
                        p = p2;
                        i = 0;
                    }
                }
                else
                {
                    delete p2;
                    p2 = NULL;
                }
            }
        }
        depth++;
        if (depth > MAX_STEP)
            return NULL;
    }
    return p;
}

//分支界限法，利用爬山法求出解的上界，对BFS进行剪枝
Chess * branchBoundSearch(Chess *begin, Chess *target, int &depth)
{
    int step, height;

    //求上界
    Chess *min = hillClimbingSearch(begin, target, depth);
    cout << "\t运用爬山法求得上界为：" << depth<<"次"<<endl;

    Chess *p1, *p2, *p=min, *q;
    //利用队列进行BFS
    queue<Chess *> queue;
    queue.push(begin);
```

```
while (min == p || queue.empty())
{
    step = 0;
    p1 = queue.front();
    queue.pop();
    for (int i = 1; i <= 4; i++)
    {
        Direction direction = (Direction)i;
        if (direction == p1->blockedDirection)
            continue;
        p2 = p1->moveChess(direction);
        if (p2 != p1)
        {
            p2->evaluate(target);
            if (p2->value <= p1->value)
            {
                p2->parent = p1;
                switch (direction)
                {
                case Up:
                    p2->blockedDirection = Down;
                    break;
                case Down:
                    p2->blockedDirection = Up;
                    break;
                case Left:
                    p2->blockedDirection = Right;
                    break;
                case Right:
                    p2->blockedDirection = Left;
                    break;
                }
                queue.push(p2);
                if (p2->value == 0)
                {
                    min = p2;
                    i = 5;
                    depth = height;
```

```
                }
            }
            else
            {
                delete p2;
                p2 = NULL;
            }
        }
    }
    q = queue.front();

    //求当前移动步数
    while (q->parent != NULL)
    {
        q = q->parent;
        step++;
    }
    height = step + 1;
    if (step >= depth)
        break;

  }
  return min;
}

int main()
{
  int depth;
  Chess *begin, *target, *t;
  begin = new Chess();
  target = new Chess();
  cout << "************************************" << endl;
  cout << "    八数码问题——分支界限法版本" << endl;
  cout << "************************************" << endl;
  cout << "请设置目标棋盘：" << endl;
  target->setChess();
  cout << "请设置初始棋盘：" << endl;
  begin->setChess();
```

```
    begin->evaluate(target);
    t = branchBoundSearch(begin, target, depth);
    if (t)
    {
        Chess *p = t;
        stack<Chess *>stack;
        while (p->parent != NULL)
        {
            stack.push(p);
            p = p->parent;
        }
        cout << "搜索结果：" << endl;
        while (!stack.empty())
        {
            stack.top()->printChess();
            stack.pop();
        }
        cout << "搜索完成！" << endl;
        cout << "移动次数为：" << depth << "次" << endl;
    }
    else
        cout << "搜索不到结果，搜索深度为：" << depth;
    delete begin, target;
    return 0;
}
```

3. 旅行推销商问题（分支界限法）

Node.h

```
#include<vector>
using namespace std;

class Node
{
    friend struct cmp;
    friend class Travelling;
public:
    Node();
    Node(vector<vector<int>>matrix, vector<int>currentPath, int wage,
```

```
int level, int currentNode)
    {
        this->matrix = matrix;
        this->currentPath = currentPath;
        this->wage = wage;
        this->level = level;
        this->currentNode = currentNode;
    }
    ~Node();
private:
    //邻接矩阵
    vector<vector<int>> matrix;
    //当前路径
    vector<int> currentPath;
    //规约代价值
    int wage;
    //当前节点所处层数
    int level;
    //当前节点编号
    int currentNode;
};


struct cmp
{
    bool operator()(Node* &first, Node* &second)
    {
        return first->wage > second->wage;
    }
};
```

Travelling.h

```
#include<iostream>
using namespace std;
#include "Node.h"
#include<queue>
#include<stack>
#include <stdlib.h>
#include<time.h>
```

```
#define NO_EDGE  65535

class Travelling
{
public:
    Travelling();
    Travelling(int size)
    {
        this->size = size;
        this->upperBound = 0;
    }
    ~Travelling();
private:
    //邻接矩阵大小
    int size;
    //邻接矩阵
    vector<vector<int>> matrix;
    //最小代价路径
    vector<int> path;
    //上界
    int upperBound;
public:
    void findPath();
private:
    //输入邻接矩阵，返回规约代价值
    int restrain(vector<vector<int>> &matrix);
    void setMatrix();
    void printPath();
    void printWage();
};
```

Travelling.cpp

```
#include "stdafx.h"
#include "Travelling.h"


Travelling::Travelling()
{
```

```
}


Travelling::~Travelling()
{
}


int Travelling::restrain(vector<vector<int>> &matrix)
{
    int wage = 0;
    int currentEdge = 0;
    bool allNoEdge;
    int matrixSize = matrix.size();

    //行规约
    for (int i = 0; i < matrixSize; i++)
    {
        allNoEdge = true;

        //从当前行找出最小边权值，进行规约
        int minEdge = matrix[i][0];
        for (int j = 1; j < matrixSize; j++)
        {
            currentEdge = matrix[i][j];
            if (currentEdge != NO_EDGE)
            {
                allNoEdge = false;
                if (minEdge > currentEdge)
                    minEdge = currentEdge;
            }
        }

        //如果当前行无边或最小边值为0则不进行计算
        if (allNoEdge == false && minEdge != 0)
        {
            for (int j = 0; j < matrixSize; j++)
            {
```

```
            currentEdge = matrix[i][j];

            //无边项不变
            if (currentEdge != NO_EDGE)
                matrix.at(i).at(j) -= minEdge;

        }
        wage += minEdge;
    }
}

//列规约
for (int i = 0; i < matrixSize; i++)
{
    allNoEdge = true;

    //从当前列找出最小边权值，进行规约
    int minEdge = matrix[0][i];
    for (int j = 1; j < matrixSize; j++)
    {
        currentEdge = matrix[j][i];
        if (currentEdge != NO_EDGE)
        {
            allNoEdge = false;
            if (minEdge > currentEdge)
                minEdge = currentEdge;
        }
    }

    //如果当前列无边或最小边值为0则不进行计算
    if (allNoEdge == false && minEdge != 0)
    {
        for (int j = 0; j < matrixSize; j++)
        {
            currentEdge = matrix[j][i];

            //无边项不变
            if (currentEdge != NO_EDGE)
```

```
                matrix.at(j).at(i) -= minEdge;

            }
            wage += minEdge;
        }
    }
    return wage;
}


void Travelling::setMatrix()
{
    int matrixSize = size;
    path.reserve(matrixSize - 1);

    //cout << "请输入对应的邻接矩阵：" << endl;
    srand((unsigned int)time(NULL));

    for (int i = 0; i < matrixSize; i++)
    {
        vector<int> col;
        for (int j = 0; j < matrixSize; j++)
        {
            int num;

            //cin >> num;
            num = rand() % 100 + 1;   //+1防止生成0

            col.push_back(num);
        }
        matrix.push_back(col);
    }
    for (int i = 0; i < matrixSize; i++)
        matrix[i][i] = NO_EDGE;

    //cout << "您输入的邻接矩阵为：" << endl;
    cout << "随机生成的邻接矩阵为：" << endl;
```

```
    cout << "-----------------------------------------------------------"
<< endl;
    for (int i = 0; i < matrixSize; i++)
    {
        for (int j = 0; j < matrixSize; j++)
        {
            cout << matrix[i][j] << "\t";
        }
        cout << endl;
    }
    cout << "-----------------------------------------------------------"
<< endl;
    for (int i = 0; i < matrixSize; i++)
    {
        if (matrix[0][i] != NO_EDGE)
            upperBound += matrix[0][i];
    }
    upperBound += matrix[matrixSize - 1][0];
}


void Travelling::findPath()
{
    setMatrix();
    vector<vector<int>> tempMatrix = matrix;
    int tempNode = restrain(tempMatrix);
    Node* firstNode = new Node(tempMatrix, path, tempNode, 0, 0);
    priority_queue<Node*, vector<Node*>, cmp>queue;
    queue.push(firstNode);
    Node* node = NULL;
    int matrixSize = matrix.size();
    cout << "选择过程：" << endl;

    //当优先队列顶端点已经到达叶子节点即找到哈密顿回路时停止搜索
    while (!queue.empty() && queue.top()->level < matrixSize - 1)
    {
        node = queue.top();
        queue.pop();
```

```
        path = node->currentPath;
        printPath();
        int j;
        for (int i = 1; i < matrixSize; i++)
        {
            //判断待搜索点是否已经在路径中
            int pathSize = node->currentPath.size();
            for (j = 0; j < pathSize; j++)
                if (i == node->currentPath[j])
                    break;
            if (j < pathSize)
                continue;

            tempMatrix = node->matrix;
            int tempWage = node->wage +
tempMatrix[node->currentNode][i];

            //当前行和被选列置为无穷
            int n = tempMatrix.size();
            for (int k = 0; k < n; k++)
            {
                tempMatrix.at(node->currentNode).at(k) = NO_EDGE;
                tempMatrix.at(k).at(i) = NO_EDGE;
            }

            //重新规约
            tempWage += restrain(tempMatrix);

            //如果下界于上界，入队
            if (tempWage <= upperBound)
            {
                vector<int> tempPath = node->currentPath;
                tempPath.push_back(i);
                Node* newNode = new Node(tempMatrix, tempPath,
tempWage, node->level + 1, i);
                queue.push(newNode);
            }
```

```
            if (!queue.empty() && queue.top()->level == matrixSize -
1)
            {
                path = queue.top()->currentPath;
                int sum = matrix[0][path.at(0)];
                int n = path.size();
                for (int i = 1; i < n; i++)
                    sum += matrix[path.at(i - 1)][path.at(i)];
                sum += matrix[path.at(n - 1)][0];
                if (sum < upperBound)
                    upperBound = sum;
            }
        }
    }
    cout << endl << "代价最小的回路为：" << endl;
    printPath();
    printWage();
    matrix.clear();
    path.clear();
}


void Travelling::printPath()
{
    cout << "1 -> ";
    for (int i = 0; i < path.size(); i++)
        cout << path[i] + 1 << " -> ";
    cout << "1" << endl;
}


void Travelling::printWage()
{
    int n = path.size();
    int sum = matrix[0][path.at(0)];
    for (int i = 1; i < n; i++)
        sum += matrix[path.at(i - 1)][path.at(i)];
    sum += matrix[path.at(n - 1)][0];
```

```
    cout << "代价权值：\tW = " << sum << endl;
}
```

TravellingSalesmanProblem.cpp

```
#include "stdafx.h"
#include "Travelling.h"


int main()
{
    int size;
    cout << "\t*************************************" << endl;
    cout << "\t     基于分支界限法的旅行推销商问题     " << endl;
    cout << "\t*************************************" << endl;
    cout << "请输入城市数量：" << endl;
    cin >> size;
    Travelling travel(size);
    travel.findPath();
    return 0;
}
```