人工智能基础exp1实验

唐晨铨 PB19000137

实验1.1

算法概述

A*算法

A*算法是每次扩展最小f(n)的结点，直到找到终点。
代码实现上，观察了书上的例子，采用了树的数据结构。

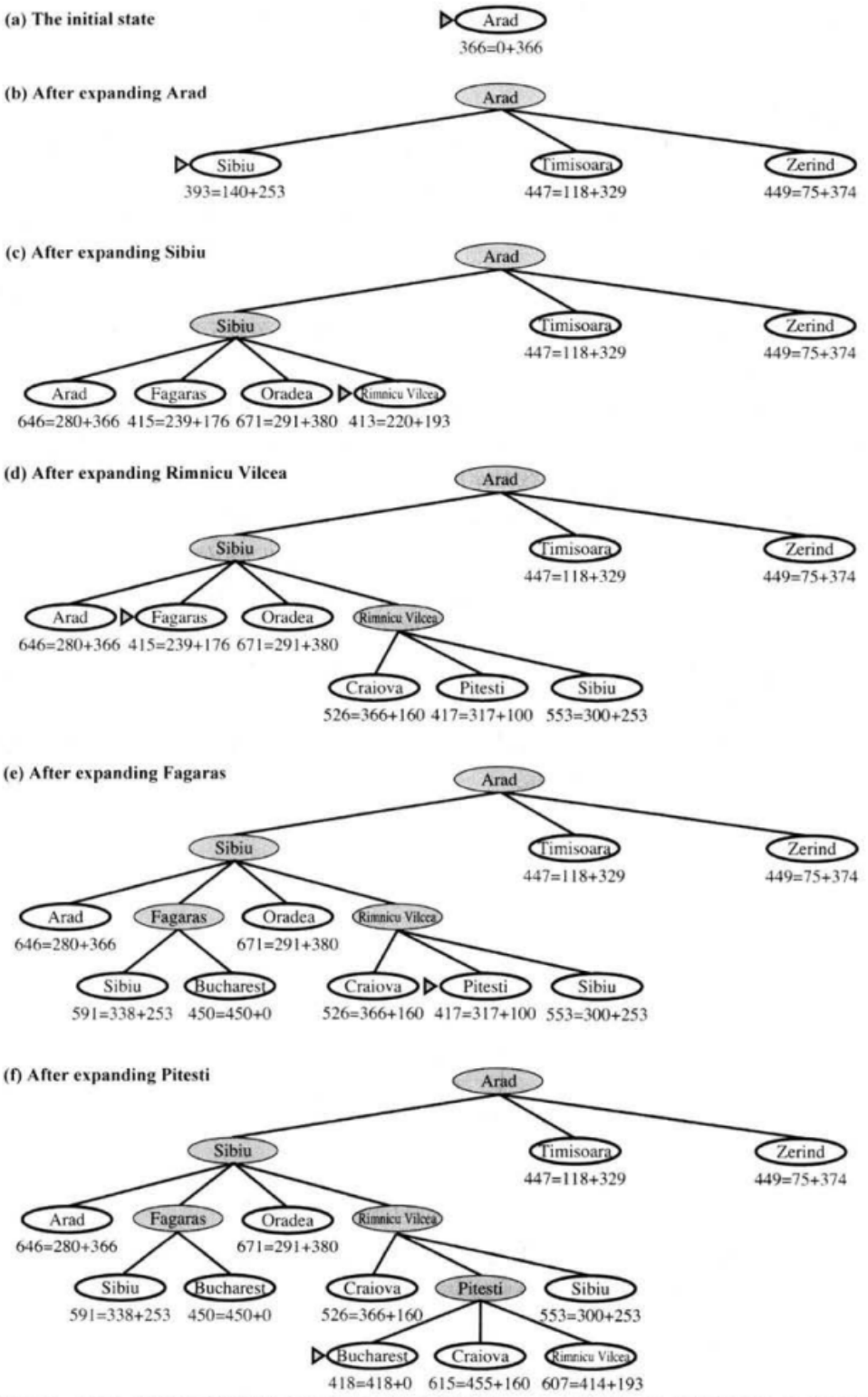

图 3.24　使用 A*搜索求解罗马尼亚问题。结点都用 $f = g + h$ 标明。h 值是图 3.22 给出的到 Bucharest 的直线距离

```
typedef struct Tree{    // A*搜索采用树结构 保存搜索路径
    int gn;             // 起点到此位置的 实际耗散
    int hn;             // 此位置到终点的 预估耗散
    int fn;             // 预估总耗散：A*中 gn+hn，IDA*中作为next_limit
    int status[5][5];   // 该节点状态
    struct Tree *child[4];// 孩子(最多4个，每个转移函数对应一个孩子)
    int n;              // 孩子数目
    struct Tree *father;// 父节点
    int action;         // 转移函数， 父节点通过 shift_func[action] 得到此节点
    struct Tree *tminfn;// 以该节点为根的子树中fn最小的节点，minfn()函数中使用与维护
    // 在IDA*中  n，tminfn，child中的3个 不会用到
}Tree;

// 初始化一个树，根节点由初始状态数组 start[5][5] 生成
Tree* initA_star(int start[5][5],int gn, int hn);
// 返回以t为根的子树中fn最小的节点
Tree* minfn(Tree* t);
// 按照转移函数，扩展节点t
void extendTree(Tree* t,int target[5][5], int h(int start[5][5], int target[5][5]));

void A_star(int start[5][5], int target[5][5],int h(int start[5][5], int target[5][5])){
    Tree* root = initA_star(start,0,h(start,target)),*tmp;  // 建立根
    while ((tmp = minfn(root))->hn)     // while (tmp is not 终点)
        extendTree(tmp,target,h);
    // tmp 是终点
}
```

在优化过程中，开始 `struct Tree` 中是没有 `struct Tree *tminfn;` 保存最小结点的， `Tree* minfn(Tree* t);` 函数的实现是每次从根遍历寻找最小fn结点，这样在跑第9个样例时，运行接近2分钟才输出结果。为此，进行了优化:**在结构体加入一个变量记录指向以此结点为根的数中最小fn的结点，minfn()函数也进行相应的调整**

Tree* minfn(Tree* t)

- 当t是NULL时，返回NULL（实际上也不曾用到）
- 当t是非叶结点时，返回 t->tminfn
- 当t是叶结点时，进行维护。（在扩展结点extendTree()中，每扩展一个叶节点，调用一次minfn()对最小fn点进行维护。）**维护方法**：扩展一个新的叶结点 **t** ，需要维护的只有 **root** 到 **t** 路径上的结点（因为 **t** 只可能是以这些结点为根的子树上的点），依次维护即可，维护的复杂度是O(h) （h是树高）

优化过后样例9仅需要0.05秒不到。

```
Tree* minfn(Tree* t){
    if(!t) return NULL; // t是NULL
    if(t->n) return t->tminfn; // t->n 不为0，说明非叶子节点，直接返回 t->tminfn
    // t是叶子节点，下面是做维护
    t->tminfn = t;
    for(Tree *tmp = t; tmp; tmp = tmp->father) // 从叶子到根结点，依次维护
        if(t->fn < tmp->tminfn->fn)
            tmp->tminfn = t;
        else if (tmp->tminfn->n){               // 此结点的tminfn不再是叶子，则从此结点孩子里的tminfn选取一个作为tminfn
            tmp->tminfn = t;
            for(int i = 0; i < tmp->n; ++i)
                if(tmp->tminfn->fn > tmp->child[i]->tminfn->fn)
                    tmp->tminfn = tmp->child[i]->tminfn;
        }
    return t->tminfn;
}
```

IDA*算法

IDA*算法就是有深度限制的dfs算法，而这个限制就是每次搜索时$gn + hn < limit$，$limit$是dfs的输入参数，当超过限制时，放松限制，进行下一轮迭代。

```
void IDA_star(int start[5][5], int target[5][5],int h(int start[5][5], int target[5][5])){
    Tree* root = initA_star(start,0,h(start,target)),*tmp;
    int bound = root->fn;
    while ((tmp = search(root,bound,target,h))->hn){ // while(tmp is not 终点)
        bound = tmp->fn;                    // 放松限制
        if(bound >= 100000){    // 限制最大值，若100000步不能找出结果，则失败
            printf("failed");
            return;
        }
    }
}
```

其他优化

除了上述寻找最小fn结点的优化外，还有

- 在状态转移时，禁止 **A->B->A** 这种转移，减少了大量的重复状态。这个算法是允许重复状态的存在，即**A->B->C->D->...->A**是允许的，但是至少需要16步转移，才能变回相同的状态，若是检查重复状态，速度必然下降！

```
if(i == 0 && t->action == 1) continue;
if(i == 1 && t->action == 0) continue;
if(i == 2 && t->action == 3) continue;
if(i == 3 && t->action == 2) continue;
```

- t->action = 0,1,2,3 分别表示了上个转移动作分别是 up,down,left,right，这里用了函数指针数组存放了4个转移函数，也便于后面书写方便。

```
int ((*shift_func[4])(int start[5][5],int status[5][5])) = {up,down,left,right};
```

- h2(n):
  h2是自定义的耗散函数，返回的是错位星球的变种曼哈顿距离之和：
  变种曼哈顿距离r=min{曼哈顿距离，4种不同的通过星际航道的曼哈顿距离}
  证明:对于飞船任何一步转移，与其转移的星球e的$r(e)$的绝对值不会变化超过1，设s是终点，则h2(s) = 0
  $h2(n) = h2(n) - h2(s) = \sum_n^{s-1} h2(k) - h2(k+1) \leq \sum_n^{s-1} |h2(k) - h2(k+1)| \leq \sum_n^{s-1} 1 =$ 步数，故h2是可取

输入要求：

将3个实参传入主函数，第一个实参表示上诉4种函数，只可能为 A_h1 A_h2 IDA_h1 IDA_h2 其中之一，第二个实参表示初始星球分布文件名，第三个实参表示目标星球分布文件名，终端执行时如 ./a.out A_h1 input01.txt target01.txt

输出要求:

输出一串动作系列（字符串形式），如 **DDLLDDR** 。
一共有4种动作，U表示向上，L表示向左，R表示向右，D表示向下。

运行结果:

```
>a.exe A_h1 ../data/input00.txt ../data/target00.txt
DDRUR
>a.exe A_h1 ../data/input01.txt ../data/target01.txt
ULLUULDD
>a.exe A_h1 ../data/input02.txt ../data/target02.txt
DDLUULLURR
>a.exe A_h1 ../data/input03.txt ../data/target03.txt
DLDRRURRRUUURR
>a.exe A_h1 ../data/input04.txt ../data/target04.txt
LUUURULLURDDRDR
>a.exe A_h1 ../data/input05.txt ../data/target05.txt
LLUURRRUURDDDDLUURDD
>a.exe A_h1 ../data/input06.txt ../data/target06.txt
DRDLLULULUUURDRURDRDRRR
>a.exe A_h1 ../data/input07.txt ../data/target07.txt
URRRRDLLLLDRRRRDLLLLDRRRR
>a.exe A_h1 ../data/input08.txt ../data/target08.txt
DDRULLLLDRUUUULDRRRRULDDDDR
>a.exe A_h1 ../data/input09.txt ../data/target09.txt
RDRURRDRUUULDLDLDDLUUUURRURR
>a.exe A_h1 ../data/input10.txt ../data/target10.txt
DDRRUUUULLULLUULLLLLUURRDDDDRR
>a.exe A_h1 ../data/input11.txt ../data/target11.txt
DRUURDRRDRUULDLULDLDRDLDRURDRURD

>a.exe A_h2 ../data/input00.txt ../data/target00.txt
DDRUR
>a.exe A_h2 ../data/input01.txt ../data/target01.txt
ULLUULDD
>a.exe A_h2 ../data/input02.txt ../data/target02.txt
DDLUULLURR
>a.exe A_h2 ../data/input03.txt ../data/target03.txt
DLDRRURRRUUURR
>a.exe A_h2 ../data/input04.txt ../data/target04.txt
LUUURULLURDDRDR
>a.exe A_h2 ../data/input05.txt ../data/target05.txt
LLUURRRUURDDDDLUURDD
>a.exe A_h2 ../data/input06.txt ../data/target06.txt
DRDLLULULUUURDRURDRDRRR
>a.exe A_h2 ../data/input07.txt ../data/target07.txt
URRRRDLLLLDRRRRDLLLLDRRRR
>a.exe A_h2 ../data/input08.txt ../data/target08.txt
DDRULLLLDRUUUULDRRRRULDDDDR
>a.exe A_h2 ../data/input09.txt ../data/target09.txt
RDRURRDRUUULDLDLDDLUUUURRURR
>a.exe A_h2 ../data/input10.txt ../data/target10.txt
DDRRUUUULLULLUULLLLLUURRDDDDRR
>a.exe A_h2 ../data/input11.txt ../data/target11.txt
DRUURDRRDRUULDLULDLDRDLDRURDRURD

>a.exe IDA_h1 ../data/input00.txt ../data/target00.txt
DDRUR
>a.exe IDA_h1 ../data/input01.txt ../data/target01.txt
ULLUULDD
>a.exe IDA_h1 ../data/input02.txt ../data/target02.txt
DDLUULLURR
>a.exe IDA_h1 ../data/input03.txt ../data/target03.txt
DLDRRURRRUUURR
>a.exe IDA_h1 ../data/input04.txt ../data/target04.txt
LUUURULLURDDRDR
>a.exe IDA_h1 ../data/input05.txt ../data/target05.txt
LLUURRRUURDDDDLUURDD
>a.exe IDA_h1 ../data/input06.txt ../data/target06.txt
DRDLLULULUUURDRURDRDRRR
```

```
>a.exe IDA_h1 ../data/input07.txt ../data/target07.txt
URRRRDLLLLDRRRRDLLLLDRRRR
>a.exe IDA_h1 ../data/input08.txt ../data/target08.txt
DDRULLLLDRUUUULDRRRRULDDDDR
>a.exe IDA_h1 ../data/input09.txt ../data/target09.txt
RDRURRDRUUULDLDLDDLUUUURRURR
>a.exe IDA_h1 ../data/input10.txt ../data/target10.txt
DDRRUUUULLULLUULLLLLUURRDDDDRR
>a.exe IDA_h1 ../data/input11.txt ../data/target11.txt
DRUURDRRDRUULDLULDLDRDLDRURDRURD

>a.exe IDA_h2 ../data/input00.txt ../data/target00.txt
DDRUR
>a.exe IDA_h2 ../data/input01.txt ../data/target01.txt
ULLUULDD
>a.exe IDA_h2 ../data/input02.txt ../data/target02.txt
DDLUULLURR
>a.exe IDA_h2 ../data/input03.txt ../data/target03.txt
DLDRRURRRUUURR
>a.exe IDA_h2 ../data/input04.txt ../data/target04.txt
LUUURULLURDDRDR
>a.exe IDA_h2 ../data/input05.txt ../data/target05.txt
LLUURRRUURDDDDLUURDD
>a.exe IDA_h2 ../data/input06.txt ../data/target06.txt
DRDLLULULUUURDRURDRDRRR
>a.exe IDA_h2 ../data/input07.txt ../data/target07.txt
URRRRDLLLLDRRRRDLLLLDRRRR
>a.exe IDA_h2 ../data/input08.txt ../data/target08.txt
DDRULLLLDRUUUULDRRRRULDDDDR
>a.exe IDA_h2 ../data/input09.txt ../data/target09.txt
RDRURRDRUUULDLDLDDLUUUURRURR
>a.exe IDA_h2 ../data/input10.txt ../data/target10.txt
DDRRUUUULLULLUULLLLLUURRDDDDRR
>a.exe IDA_h2 ../data/input11.txt ../data/target11.txt
DRUURDRRDRUULDLULDLDRDLDRURDRURD
```

4个算法得出的结果一样
列表统计

样例	A_h1时间(秒)	A_h2时间(秒)	IDA_h1时间(秒)	IDA_h2时间(秒)	结果	步数
0	7.68E-06	1.06E-05	5.91E-06	7.09E-06	DDRUR	5
1	1.06E-05	2.01E-05	5.91E-06	2.19E-05	ULLUULDD	8
2	1.06E-05	1.42E-05	7.09E-06	2.13E-05	DDLUULLURR	10
3	5.73E-05	7.86E-05	4.49E-05	1.09E-04	DLDRRURRRUUURR	14
4	1.52E-04	4.25E-05	1.73E-04	3.25E-05	LUUURULLURDDRDR	15
5	3.07E-04	4.25E-05	2.58E-04	2.60E-05	LLUURRRUURDDDDLUURDD	20
6	6.12E-04	2.90E-04	6.34E-04	5.05E-04	DRDLLULULUUURDRURDRDRRR	23
7	7.74E-05	1.09E-04	3.13E-05	5.79E-05	URRRRDLLLLDRRRRDLLLLDRRRR	25
8	2.41E-03	1.48E-04	3.20E-03	9.92E-05	DDRULLLLDRUUUULDRRRRULDDDDR	27
9	4.96E-02	8.96E-03	5.43E-02	1.45E-02	RDRURRDRUUULDLDLDDLUUUURRURR	28

样例	A_h1时间(秒)	A_h2时间(秒)	IDA_h1时间(秒)	IDA_h2时间(秒)	结果	步数
10	1.91E-03	1.68E-04	3.27E-03	2.55E-04	DDRRUUUULLULLUULLLLLUURRDDDDRR	30
11	1.50E-01	2.55E-03	1.64E-01	3.63E-03	DRUURDRRDRUULDLULDLDRDLDRURDRURD	32

实验1.2 CSP 作业调度问题

算法思想

设员工人数num，设定一个数组dispatch[num][7]，dispatch[id][day]==0就表示员工id在day天休息，等于1就是值日，那么此题可以类比着色问题，在num*7个格子着两种颜色，涂色有一定的约束如题给出。类比涂色问题采用dfs回溯即可。

变量集合：dispatch[num][7]
值域集合：{0,1}
约束集合：{check1(),check2(),check4(),check5(),check6()}

```
int dfs(int k){ // 涂第k个格子
    int id = k % num, day = k / num;
    if(day == 7) // 涂完
        return 1;

    for(int color = 0;color < 2; color++){  // 给格子上色（这里只有两种颜色，代表值日与休息）
        dispatch[id][day] = color;
        if(check1(id,day)||check2(id,day)||check4(id,day)||check5(id,day)||check6(id,day))
            continue;   // 5种约束，对于题目所说的约束
        // 无冲突，下面进行dfs迭代下一个格子
        if(dfs(k+1))
            return 1;
    }
    return 0;
}
```

算法优化

采用了MRV启发式优化，考虑到只有两种颜色，故当某个格子被约束时，直接赋值，MRV在检查约束时，顺便赋值。用了一个数组 `MRV` 保存被MRV启发式赋值的格子， `MRVn` 是上述格子数，这样当dfs迭代到这个被MRV启发式赋值的格子时候，就不考虑涂色了。

采用MRV时dfs调用次数	不采用MRV时dfs调用次数
133	173
101	131

输出格式为：

1 2 3 ...
1 2 3 ...
1 2 3 ...
1 2 3 ...
1 2 3 ...

1 2 3 ...

1 2 3 ...

其中数字代表从周一到周天的工人编号

输入输出

```
>a.exe input1.txt
2 5 6 7
2 4 6 7
1 3 5 7
2 4 6 7
2 4 5 6
1 3 5 7
2 4 5 6
>a.exe input2.txt
5 6 7 9 10
5 6 7 9 10
1 2 3 4 8
5 6 7 9 10
5 6 7 9 10
1 2 3 4 8
5 6 7 9 10
```

局部搜索

算法:

dispatch邻居定义：有且仅有一个格子赋值与邻居不同，故每个状态有$7*num$个邻居

约束函数：$check1(),check2()...check6()$同回溯算法，对应于题目中的约束条件，返回值变为冲突数

评价函数：$p=N1*check1()+N2*check2()+...+N6*check6(),$其中$N1>>N2>>...>>N6=1$

随机重启爬山法 （准确来说是 随机重启下山法）

1. 当搜索时间大于给定值时，返回失败
2. 随机生成一个dispatch[num][7]作为初始状态 S ,其中随机赋值0或1
3. 当不存在冲突时，返回 S，否则转4
4. 计算所有的邻居的 p 值，记最小 p 的邻居为 N
5. 若 $p(N)>=p(S)$ ，转1
6. $S=N$，转3