Lab01 24 数码 实验报告

项子扬 PB16001768

1. 实验结果

一、Ah1

1.

8.387451us
DDLDD
5

2.

47.042658us
LULDLULLDD
10

3.

8400.943479us
LLUURRRUURDDDDLUURDD
20

4 以及之后无法得出结果。

二、Ah2

1.

12.398840us
DDLDD
5

2.

168.843028us
LULDLULLDD
10

3.

272.774481us
LLUURRRUURDDDDLUURDD
20

4.

907938.977285us
RDRDLUUUURRDRDDRUUULLDRULURR
28

5 以及之后无法得出结果。

三、IDAh1
1.

5.834748us
DDLDD
5

2.

28.444398us
LULDLULLDD
10

3.

551.383711us
LLUURRRUURDDDDLUURDD
20

4.

9001110.060856us
RDRURRDRUUULDLDLDDLUUUURRURR
28

5 以及之后无法得出结果

四、IDAh2
1.

10.575481us
DDLDD
5

2.

77.675086us
LULDLULLDD
10

3.

123.623729us
LLUURRRUURDDDDLUURDD
20

4.

52833.280821us
RDRURRDRUUULDLDLDDLUUUURRURR
28

5 以及之后无法得出结果

2.算法时空复杂度

一、Ah1
时间复杂度大于 O(n^3) 空间呈指数型增长。
二、Ah2
时间复杂度在临近计算性能极限时近似为 O(n^2)，临近极限时无法估计。
空间呈指数型增长。
三、IDAh1
比 Ah1 好，比 Ah2 差。
四、IDAh2
性能在四者中最好。

3.关键代码说明

一、Ah1、Ah2

```
double run_time;
_LARGE_INTEGER time_start;   //开始时间
_LARGE_INTEGER time_over;    //结束时间
double dqFreq;               //计时器频率
LARGE_INTEGER f;             //计时器频率
QueryPerformanceFrequency(&f);
dqFreq=(double)f.QuadPart;
```

```
run_time=1000000*(time_over.QuadPart-time_start.QuadPart)/dqFreq;
```

此部分用于程序计时。

```
typedef struct node{
    int data[5][5];//数码状态
    int zerox;//0所在的坐标
    int zeroy;
    int dir;//移动方向 1上2下3左4右
    int g;//路径耗散
    int h;//启发式函数
    struct node *parent;//父节点
    struct node *next;//下一个节点
}node,*pnode;
```

此为节点的结构体，data 数组用于存储数码状态，zerox 和 zeroy 记录 0 的位置，g 和 h 是节点的路径耗散和启发式函数，加起来就是估值 f，dir 记录上一个节点的移动方向，parent 指向上一个将其展开的节点，dir 与 parent 一起可以回溯出搜索的具体路径；next 指针用于形成 open 表或 close 表的链表结构。

```
while(open->next!=NULL){
    flag=0;
    m=1000000;
    min=open->next;
    for(q=open->next;q!=NULL;q=q->next){
        if(q->g+q->h<m){
            m=q->g+q->h;
        }
    }
    for(min=open->next,minp=open;min!=NULL;min=min->next,minp=minp->next){
        if(min->g+min->h==m) break;
    }
    minp->next=min->next;//在open表中删除最小节点
    min->next=close->next;      //插入close表头
    close->next=min;
    if(Isgoal(min)) break;
    for(i=1;i<=4;i++){//将所有可能的新节点插入open表
        if(move(min,i)){//分四种方向判断
            p=(pnode)malloc(sizeof(node));
            copy(p,min);
            if(i==1){//根据移动方向为新节点的数据赋值
                p->zerox=(min->zerox+4)%5;
                p->zeroy=min->zeroy;
                p->data[p->zerox][p->zeroy]=0;
                p->data[(p->zerox+1)%5][p->zeroy]=min->data[(min->zerox+4)%5][min->zeroy];
            }
            else if(i==2){
```

```
                p->zerox=(min->zerox+1)%5;
                p->zeroy=min->zeroy;
                p->data[p->zerox][p->zeroy]=0;
                p->data[(p->zerox+4)%5][p->zeroy]=min->data[(min->zerox+1)%5][min->zeroy];
            }
            else if(i==3){
                p->zerox=min->zerox;
                p->zeroy=(min->zeroy+4)%5;
                p->data[p->zerox][p->zeroy]=0;
                p->data[p->zerox][(p->zeroy+1)%5]=min->data[min->zerox][(min->zeroy+4)%5];
            }
            else{
                p->zerox=min->zerox;
                p->zeroy=(min->zeroy+1)%5;
                p->data[p->zerox][p->zeroy]=0;
                p->data[p->zerox][(p->zeroy+4)%5]=min->data[min->zerox][(min->zeroy+1)%5];
            }
            p->dir=i;
            p->g=min->g+1;
            p->h=geth(p);
            p->parent=min;
            if(Isgoal(p)){
                min=p;
                flag=1;
                break;
            }
            if(Isclose(close,p)){//若新节点在close表中，跳过
```

```
                free(p);
                continue;
            }
            if(q=Isopen(open,p)){//在open表中
                if(p->g+p->h<q->g+q->h){//若新节点的代价比open中的小，则替换代价
                    q->g=p->g;
                    q->h=p->h;
                    q->parent=p->parent;
                }
                free(p);
            }
            else{//不在open表中，则将新节点插入open尾部
                RearInsert(open,p);
            }
        }
    }
    if(flag) break;
}
```

此部分为循环迭代部分，其中采用了顺序遍历的方法找到 open 表中的估值最小节点，在对节点做展开时先用一个 move 函数判断移动方向是否可行，若可行，则进行如下判断：若新节点就是目标状态，直接返回；若新节点在 close 表中，则舍弃；若新节点在 open 表中，则比较两者的估值，若新节点的估值更小，则将 open 表中原来的节点相关信息改为新节点的值（如父节点），反之则丢弃；若新节点不在 open 表中，则将新节点插入 open 表的尾部。

其中用到的重要函数：

```
int geth(pnode p){
    int ret,i,j;
    ret=0;
    for(i=0;i<5;i++){
        for(j=0;j<5;j++){
            if(p->data[i][j]!=(i*5+j+1)%25) ret++;
        }
    }
    return ret;
}
```

```
int min(int a,int b,int c,int d,int e){
    int m;
    m=100000;
    if(a<m) m=a;
    if(b<m) m=b;
    if(c<m) m=c;
    if(d<m) m=d;
    if(e<m) m=e;
    return m;
}

int geth(pnode p){//变种曼哈顿
    int ret,i,j,a,b;
    ret=0;
    for(i=0;i<5;i++){
        for(j=0;j<5;j++){
            if(p->data[i][j]!=0){
                a=(p->data[i][j]-1)/5;
                b=(p->data[i][j]-1)%5;
            }
            else{
                a=4;
                b=4;
            }
            ret+=min(abs(a-i)+abs(b-j),abs(i-2)+abs(a-2)+abs(j-0)+abs(b-4),abs(i-2)+abs(a-2)+abs(j-4)+abs(b-0),
            abs(i-0)+abs(a-4)+abs(j-2)+abs(b-2),abs(i-4)+abs(a-0)+abs(j-2)+abs(b-2));
        }
    }
    return ret;
}
```

此部分为启发式函数，在 Ah1 中为错位的棋子数，Ah2 中采用了变种曼哈顿距离的计算，即在曼哈顿距离的计算方法上额外考虑四条路线，在 5 者中取最小值即为两点之间的最短曼哈顿距离。

```
int move(pnode p,int dir){//判断移动方向是否可行
    if(p->dir==0){
        switch(dir){
            case 1:{//向上移动
                if(p->zerox==0&&p->zeroy!=2) return 0;
                break;
            }
            case 2:{
                if(p->zerox==4&&p->zeroy!=2) return 0;
                break;
            }
            case 3:{
                if(p->zeroy==0&&p->zerox!=2) return 0;
                break;
            }
            case 4:{
                if(p->zeroy==4&&p->zerox!=2) return 0;
                break;
            }
        }
    }
    else{
        switch(dir){
            case 1:{//向上移动
                if((p->dir==2)||(p->zerox==0&&p->zeroy!=2)) return 0;
                break;
            }
            case 2:{
                if((p->dir==1)||(p->zerox==4&&p->zeroy!=2)) return 0;
                break;
            }
            case 3:{
                if((p->dir==4)||(p->zeroy==0&&p->zerox!=2)) return 0;
                break;
            }
            case 4:{
                if((p->dir==3)||(p->zeroy==4&&p->zerox!=2)) return 0;
                break;
            }
        }
    }
    return 1;
}
```

此函数用于判断当前节点是否能朝所选方向移动（1 上 2 下 3 左 4 右），若节点不是初始节点，则不能朝反方向移动。

二、IDAh1、IDAh2

与 A*算法不同的地方在于不在需要查重，每次迭代设置一个阈值 limit，大于 limit 的节点不予展开，并根据这些不展开的节点的估值得出下一次迭代的新阈值。每次迭代开始时，都从最初的节点开始重新展开，直到得到目标状态。

函数主体部分：

```
while(1){
    nextlimit=1000000;
    open->next=ori;
    while(open->next!=NULL){
        flag=0;
        q=open->next;
        open->next=q->next;
        if(q->g+q->h>limit){
            nextlimit=(nextlimit<(q->g+q->h))?nextlimit:(q->g+q->h);
        }
        else{
            if(Isgoal(q)){
                flag=1;
                break;
            }
            for(i=1;i<=4;i++){//将所有可能的新节点插入open表
                if(move(q,i)){//分四种方向判断
                    p=(pnode)malloc(sizeof(node));
                    copy(p,q);
                    if(i==1){//根据移动方向为新节点的数据赋值
                        p->zerox=(q->zerox+4)%5;
                        p->zeroy=q->zeroy;
                        p->data[p->zerox][p->zeroy]=0;
                        p->data[(p->zerox+1)%5][p->zeroy]=q->data[(q->zerox+4)%5][q->zeroy];
                    }
                    else if(i==2){
                        p->zerox=(q->zerox+1)%5;
                        p->zeroy=q->zeroy;
                        p->data[p->zerox][p->zeroy]=0;
                        p->data[(p->zerox+4)%5][p->zeroy]=q->data[(q->zerox+1)%5][q->zeroy];
                    }
                    else if(i==3){
                        p->zerox=q->zerox;
                        p->zeroy=(q->zeroy+4)%5;
                        p->data[p->zerox][p->zeroy]=0;
                        p->data[p->zerox][(p->zeroy+1)%5]=q->data[q->zerox][(q->zeroy+4)%5];
                    }
                    else{
                        p->zerox=q->zerox;
                        p->zeroy=(q->zeroy+1)%5;
                        p->data[p->zerox][p->zeroy]=0;
                        p->data[p->zerox][(p->zeroy+4)%5]=q->data[q->zerox][(q->zeroy+1)%5];
                    }
                    p->dir=i;
                    p->g=q->g+1;
                    p->h=geth(p);
                    p->parent=q;
                    RearInsert(open,p);
                }
            }
        }
    }
    limit=nextlimit;
    if(flag) break;
}
```

其余部分函数与 A*大致相同，不再赘述。

4.使用说明以及如何提升速度

使用说明：在源代码的同目录下添加 input.txt 文件，由于本次实验中的目标状态都相同，故已在代码中初始化，无需再添加 target.txt 文件。Input 文件中为 5*5 矩阵内容，直接编译运行源代码，会在同一目录下生成对应的 solution.txt，打开即为运行结果。

如何提升运行速度：
以某种规律对数码状态进行分类后排序，在表中查找时采用二分查找，可以提高程序运行效率。