山东大学 计算机科学与技术 学院

人工智能导论 课程实验报告

学号：202300130183	姓名： 宋浩宇	邮 箱 ： 202300130183 @mail.sdu.edu.cn

实验题目：四、8 数码问题

实验过程：

（记录实验过程、遇到的问题和实验结果。可以适当配以关键代码辅助说明，但不要大段贴代码。）

为了使用 A*算法解决八数码问题，首先确定该问题为搜索问题。

该问题有有限状态空间，每个状态之间能通过有限次操作算子达到，且每次操作的 cost 是固定且可计算可预测的，故该问题可以使用搜索算法来解决，也可以使用 A*算法来得到最优解。

A*算法的实现如下：

首先定义需要的容器：

```
//状态定义
struct board
{
    int status[3][3];
};

//零点位置定义
struct zero
{
    int x;
    int y;
};

//节点定义
struct node
{
    board status;//当前数码盘状态
    int f_cost;//搜索代价
    int g_cost;//路径代价
    int h_cost;//预期代价
    vector<node*> search_list;//搜索表
    node* front;//父节点
    zero mark;
};

vector<node*> openlist;//openlist
vector<node*> closelist;//closelist
board searchtarget;//搜索终点
vector<node*>::iterator iter;//定义迭代器
vector<node> ans;//定义结果
```

然后定义需要用到的功能函数：

```
//判断两个节点是否相同
bool CheckNode(node& node1, node& node2)
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (node1.status.status[i][j]!=node2.status.status[i][j])
            {
                return false;
            }
        }
    }
    return true;
}

//将一个节点复制到另一个节点a->b
void CopyNode(node& a, node& b)
{
    b.f_cost = a.f_cost;
    b.g_cost = a.g_cost;
    b.h_cost = a.h_cost;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            b.status.status[i][j] = a.status.status[i][j];
        }
    }
    b.mark.x = a.mark.x;
    b.mark.y = a.mark.y;
}
```

```
    }
    //交换数组中两个数据的位置
void Exchange(int& a, int& b)
{
    int c = a;
    a = b;
    b = c;
}
```

然后根据容器来完成 h(x) 和 f(x) 的计算函数：

```
//h_cost计算
void HCoustCoculate(node &nodes,//当前节点
    board &taget)//目标状态
{
    int sum = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (nodes.status.status[i][j] != taget.status[i][j])
            {
                sum++;
            }
        }
    }
    nodes.h_cost = sum;
    return;
}

//f_cost计算
void FCostCoculate(node& nodes)
{
    HCoustCoculate(nodes, searchtarget);
    nodes.f_cost = nodes.h_cost + nodes.g_cost;
    return;
}
```

然后书写判定被处理节点是否存在于 openlist 和 closelist 中的函数

```
//判断新节点是否在openlist里
bool CheckOpen(node& nodes)
{
    iter = openlist.begin();
    while (iter!=openlist.end())
    {
        if (CheckNode((**iter), nodes))
        {
            return true;
        }
        iter++;
    }
    return false;
}

//刷新openlist重复节点
void RefreshOpenlist(node& nodes)
{
    iter = openlist.begin();
    while (!CheckNode((**iter),nodes))
    {
        iter++;
    }
    if ((*iter)->g_cost > nodes.g_cost)
    {
        CopyNode(nodes, **iter);
    }
}


//判断新节点是否在closelist里
bool CheckClose(node& nodes)
{
    iter = closelist.begin();
    while (iter != closelist.end())
    {
        if (CheckNode(*(*iter), nodes))
        {
            return true;
        }
        iter++;
    }
    return false;
}
```

其中“刷新 openlist 重复节点”的函数是为了处理当搜索搜到与 openlist 中存在的节点状态相同的节点的时候保留最优解。

然后创建终点判定函数：

```
//终点函数
bool Complete()
{
    int as = 0;
    iter = openlist.begin();
    if (openlist.empty())
    {
        cout << "openlist为空" << endl;
        return false;
    }
    while (iter!=openlist.end())
    {
        if ((**iter).h_cost == 0)
        {
            sas = as;
            return false;
        }
        iter++;
        as++;
    }
    return true;
}
```

以 openlist 为空（问题无解）或搜到目标节点作为终点，下方 iter 迭代器是为了保证搜索到终点节点时保留下这个节点的地址的引用。
然后实现扩展节点的函数

```
//搜索表创建
void SearchListCreate(node& nodes)
{
    sum++;
    int temmark = 1;
    //左移
    if (nodes.mark.y > 0){ ... }
    temmark = 1;
    //右移
    if (nodes.mark.y < 2){ ... }
    temmark = 1;
    //上移
    if (nodes.mark.x> 0){ ... }
    temmark = 1;
    //下移
    if (nodes.mark.x < 2){ ... }
}
```

以其中的左移为例

```
int temmark = 1;
//左移
if (nodes.mark.y > 0)
{
    node* p = new(node);
    CopyNode(nodes, *p);
    p->mark.y--;
    Exchange(p->status.status[p->mark.x][p->mark.y], p->status.status[p->mark.x][p->mark.y + 1]);
    p->g_cost++;
    p->front = &nodes;
    FCostCoculate(*p);
    if (CheckClose(*p))
    {
        delete(p);
        temmark = 0;
    }
    if (temmark == 1)
    {
        if (CheckOpen(*p))
        {
            RefreshOpenlist(*p);
        }
        else
        {
            openlist.push_back(p);
        }
        nodes.search_list.push_back(p);
    }
}
```

创建一个新的节点，先将原节点复制到新节点，再将新节点里的 0 和它左侧的数字交换位置，g(n)自增，并存下父节点的地址，然后计算新的 f(n)的值。如果这个新的节点在 closelist 中，则删除这个节点，并通过 temmark 标记跳过下方对 openlist 的检查，如果它不在 closelist 中，则检查它是否在 openlist 中，如果在则刷新 openlist 留下最优解，如果它不在 openlist 中，则把它加入 openlist 作为待搜索节点。

```
//搜索函数
void Search()
{
    while (Complete())
    {
        sas++;
        sort(openlist.begin(), openlist.end(),compare);
        // ...

        // ...
        SearchListCreate(*openlist[0]);
        closelist.push_back(openlist[0]);
        openlist.erase(openlist.begin());
    }

}
void debug(node& nodes)
```

搜索过程如上，即先对 openlist 进行排序，选出 f(n)最小的节点在对其进行拓展，并将该节点移入 closelist 中，然后重复上述过程直到搜索到终点状态或者 openlist 为空（无解）

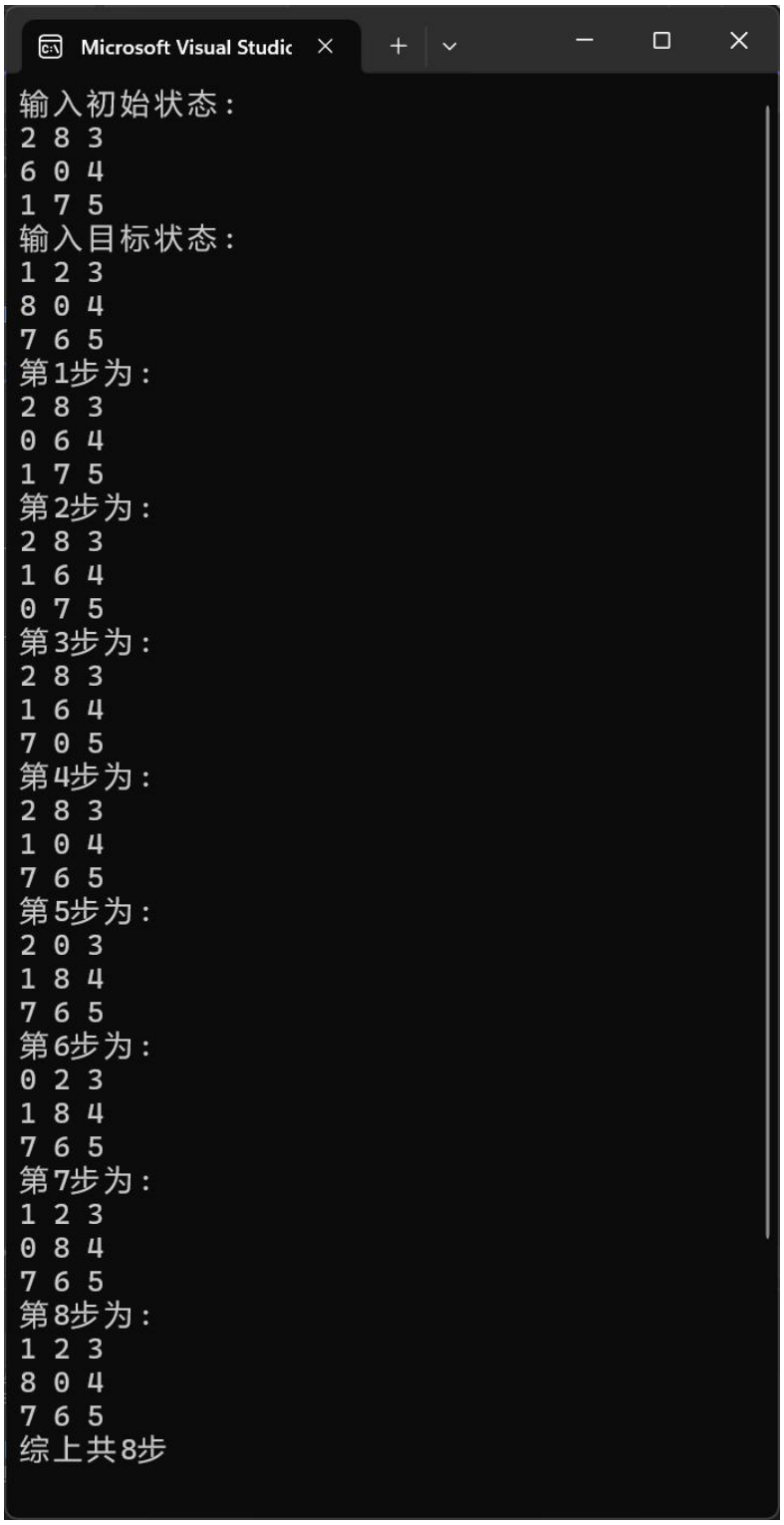
需要补充的是，由于这里的 sort（）需要进行对结构体指针的比较，故定义 compare 函数来作为 sort（）的第三个参数来实现排序过程：

```
//传入sort函数的参数
bool compare(node*& node1, node*& node2)
{
    //cout << node1->f_cost << endl;
    //cout << node2->f_cost << endl;
    return (node1->f_cost) < (node2->f_cost);
}
```

最后完善主函数

```
int main()
{
    //输入初始状态
    cout << "输入初始状态:" << endl;
    node* temp = new(node);
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cin >> temp->status.status[i][j];
            if (temp->status.status[i][j] == 0)
            {
                temp->mark.x = i;
                temp->mark.y = j;
            }
        }
    }
    temp->g_cost = 0;
    temp->front = NULL;
    cout << "输入目标状态:" << endl;
    //输入搜索终点
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cin >> searchtarget.status[i][j];
        }
    }
    //初始化openlist和closelist
    openlist.push_back(temp);
    FCostCoculate(*temp);
    Search();
    //cout << sas << endl;
    temp = *iter;
    while (temp->front!=NULL)
    {
        ans.push_back(*temp);
        temp = temp->front;
    }
    int tem = 1;
    for (int i = ans.size() - 1; i >= 0; i--)
    {
        cout << "第" << tem++ << "步为:" << endl;
        debug(ans[i]);
    }
    cout << "综上共" << tem << "步" << endl;
    return 0;
}
```

测试课本所给的数据，运行，成功



成功搜索到答案。

但由于算法本身没有进行优化，所以在测试更复杂的数据时产生了下述问题。

遇到的问题：

1. 答案不是最优解
2. 搜索时间过长

上述两个问题，第一个问题通过改进 h(n)估价函数来解决，通过将计算曼哈顿距离的估价函数改为计算不在对应位置上的数字的个数/2 向上取整的函数成功

做到 h(n)<=h*(n)搜到最优解；第二个问题通过将 A*改为双向 A*解决，效率提升大概为将原本需要的搜索时间缩短到 1/20，但目前看来速度依然有限，考虑到计算过程需要反复遍历 openlist 和 closelist，预计将用 vector 储存改为用 map 等哈希类数据结构来储存将节省大量时间，但因为需要重构代码故未执行。

结果分析与体会：

A*算法最重要的就是 h(n)这个函数，这个函数的性能将决定 A*算法的搜索速度和是否能搜索到最优解，在实际的算法实现中，这也是需要考虑和实验最多的部分。另外对于大部分的搜索算法，如果已知搜索终点，都可以使用双向搜索的方式来大大提高搜索效率，且如果使用的算法需要储存大量节点，并且因为需要进行查找操作而要多次遍历容器，那么使用哈希容器来储存是效率最高的。