A*算法实验报告

1. 导言

在本次实验中，我们要使用A*算法解决八数码问题。

A算法是一种或图通用搜索算法，其启发式函数的形式为 $f(n)=g(n)+h(n)$ 。而A*算法是一种特殊的A算法，有 $h(n)<=h^*(n)$，其中 $h^{*}(n)$ 表示从 n 到 S_g 的实际最小费用的估计。

2. 实验过程

A*搜索的算法流程

设 S_0 为初始节点集，S_g 为目标节点集，G 为搜索图，$Tree$ 为搜索树。

1. $Open$表 = S_0 ， $Close$表 = {}。
2. 如果 $Open$表为空，失败退出。
3. 在 $Open$表上取出 $f(n)$ 值最小的节点 n ， n 放到 $Close$表中，即 $f(n)=g(n)+h(n), h<=h^*$ 。
4. 若 $n \in S_g$ ，则成功退出。
5. 产生 n 的一切后继节点，构成集合 M 。
6. 对 M 中的元素 P ，分别作两类处理：
    1. 如果 $P \notin G$ ，则对 P 进行估计，加入 $Open$ 表、G 和 $Tree$ 。
    2. 如果 $P \in G$ ，且如果选择从 n 指向 P 的路径使得 P 有更小的代价，即更小的 f 值 ，则更新 $Tree$ 中的 P 的前驱结点。

       如果 $P \in Close$表，则对 P 的后继结点同样需要重新评估、更新。P 的后继结点的后继结点同样如此，以此类推。
7. 转第2步。

代码框架

- package `astar` ： 定义A*搜索的基本框架
    - interface `SimpleState` ： A*搜索的基本节点的状态
    - abstract class `SimpleNode<State extends SimpleState>` ： A*搜索的基本节点，包括状态、索引、父节点索引、f/g/h评估值
    - abstract class `Astar<Node extends SimpleNode>` ： A*搜索的基本实现，子类要重载 `h` 函数
- package `eightpuzzle` ： 八数码问题的具体实现
    - class `EPState implements SimpleState` ： 八数码问题的节点状态，即数码摆放情况
    - class `EPNode extends SimpleNode<EPState>` ： 八数码问题的节点
    - package `Search1`

        - class `EP1Search extends Astar<EPNode>` ：使用h1函数的八数码A*搜索，重写 `h` 函数为返回放错位置的码个数
    - package `Search2`
        - class `EP2Search extends Astar<EPNode>` ：使用h2函数的八数码A*搜索，重写 `h` 函数为返回所有放错位置的码到正确位置的曼哈顿距离之和
    - class `EPController` ：UI控制类
    - `EPScene.fxml` ：UI布局文件

基本结点

关键代码如下：

```
package astar;

public abstract class SimpleNode<State extends SimpleState> implements Comparable<SimpleNode>{
    private State state;
    private int prev;
    private int index;
    private double fvalue;
    private double gvalue;
    private double hvalue;

    public String print() {
        return "index=" + index + " f=" + fvalue + " g=" + gvalue + " h=" + hvalue + "\n" +
state.print();
    }

    public abstract ArrayList<State> next();    // 获取后继的状态，留待子类重写

    @Override
    public boolean equals(Object obj) {     // 保证可以在List<SimpleNode>里找到与当前结点状态相同的结点
        if(!(obj instanceof SimpleNode)) return false;
        SimpleNode sn = (SimpleNode<State>) obj;
        return state == sn.state;
    }

    @Override
    public int compareTo(SimpleNode o) {    // 保证可以排序List<SimpleNode>、比较两个结点的f值大小
        return fvalue == o.fvalue ? 0 : (fvalue > o.fvalue ? 1 : -1);
    }
}
```

A*搜索

A*搜索的基本实现由抽象类 `abstract class Astar<Node extends SimpleNode>` 完成，其关键部分的代码如下：

```
package astar;

public abstract class Astar<Node extends SimpleNode> {
    private ArrayList<Node> open;   // open表，包括已评估、尚未生成后继结点的结点
```

```
    private ArrayList<Node> close;  // close表，包括已评估、已生成后继结点的结点
    private HashMap<Integer, ArrayList<Integer>> tree = new HashMap<>();    // 记录每个结点的索引和它的后继结点们的索引
    private ArrayList<Node> begin;  // 初始结点集
    private ArrayList<Node> end;    // 终止结点集
    private ArrayList<Node> graph;  // 搜索图
    private Node endNode;           // 搜索到的终止结点
    private boolean finish = false; // 搜索是否结束

    public int f(Node node) {
        return g(node) + h(node);
    }

    public int g(Node node) {
        ArrayList<Node> list = getGraph();
        int gvalue = 0;
        Node ptr = node;
        while(ptr.getIndex() != 0) {
            ptr = list.get(ptr.getPrev());
            gvalue++;
        }
        return gvalue;
    }

    public abstract int h(Node node);

    public String search() throws Exception{
        if(open.size() == 0) {
            throw new Exception("fail");
        }
        String str = "";
        str += "open表的结点数=" + open.size() + " 图的结点数=" + graph.size() + "\n\n";
        str += "选择有最小的f值的结点：\n";
        Node node = open.remove(0);
        close.add(node);
        str += node.print();

        // 已经搜索到一个终止结点
        if(end.indexOf(node) != -1) {
            endNode = node;
            finish = true;

            return str;
        }

        // 当前结点的后继状态
        ArrayList<SimpleState> states = node.next();
        for(int i = 0; i < states.size(); i++) {
            if(states.get(i) == node.getState()) continue;

            // 生成可能的后继结点
            Node newNode = getNewNode(states.get(i), node.getIndex(), graph.size());
            newNode.setValue(g(newNode), h(newNode));
```

```
            int index1 = graph.indexOf(newNode);
            if(index1 == -1) {  // 候选的后继结点不在graph里，则作为一个新结点添加到图和open表
                graph.add(newNode);
                insertOpen(newNode);
                str += "添加新的后继结点：\n";
                if(end.indexOf(newNode) != -1) {
                    endNode = newNode;
                    finish = true;
                    str += newNode.print();
                    break;
                }
            }
            else {// 候选的后继结点在graph里，则更新图
                str += "生成的后继结点是旧结点，\n需要更新图：\n";
                updateNode(newNode);
                newNode = graph.get(index1);
            }
            str += newNode.print();
        }
        return str;
    }

    private void insertOpen(Node node) {    // 向图中加入新的结点
        if(open.isEmpty() || open.get(open.size() - 1).compareTo(node) <= 0) open.add(node);
        else for(int j = 0; j < open.size(); j++) {
            if(open.get(j).compareTo(node) > 0) {
                open.add(j, node);
                break;
            }
        }

        tree.put(node.getIndex(), new ArrayList<>());
        if(!tree.containsKey(node.getPrev())) {
            tree.put(node.getPrev(), new ArrayList<>());
        }
        tree.get(node.getPrev()).add(node.getIndex());
    }

    private void updateNode(Node node) {    // 生成的后继结点node已在图中，更新对应结点
        int openIndex = open.indexOf(node);
        Node origin;
        if(openIndex != -1) {                   // node在open表里，尚未生成后继
            origin = open.get(openIndex);       // origin代表open表里，node对应的原本
        } else {                                // node在close表里，已生成后继
            int closeIndex = close.indexOf(node);
            origin = close.get(closeIndex);     // origin代表close表里，node对应的原本
        }

        if(node.compareTo(origin) < 0) {    // 若生成的新结点有比原本更小的f值，更新原本结点
            if(origin.getPrev() >= 0) {
                tree.get(origin.getPrev()).remove(new Integer(origin.getIndex()));
                origin.setPrev(node.getPrev());
```

```
                tree.get(origin.getPrev()).add(origin.getIndex());
            }
            origin.setValue(node.getGvalue(), node.getHvalue());
        } else {
            return;
        }
        if(openIndex == -1) {   // node在close表里，已生成后继，后继结点也需要检查并更新
            ArrayList<Integer> list = tree.get(origin.getIndex());
            for(int i = 0; i < list.size(); i++) {
                updateNode(graph.get(i));
            }
        }
    }
}
```

h评估函数留待子类重写。

八数码问题的具体解决

状态

八数码问题的状态的定义如下（只截取部分关键代码），包括了一个长度为9的数组：

```
package eightpuzzle;

public class EPState implements SimpleState {
    private int[] puzzles;

    @Override
    public boolean equals(Object obj) {
        if(!(obj instanceof EPState)) return false;
        EPState eps = (EPState) obj;
        return Arrays.equals(puzzles, eps.puzzles);
    }

    public ArrayList<EPState> next() {  // 获取后继状态
        ArrayList<EPState> nextStates = new ArrayList<>();
        int space = 0;
        for(int i = 0; i < puzzles.length; i++) {
            if(puzzles[i] == 0) {
                space = i;
                break;
            }
        }

        if(space % 3 == 0 || space % 3 == 1) {  // 右移
            EPState state = new EPState(this);
            state.puzzles[space] = state.puzzles[space + 1];
            state.puzzles[space + 1] = 0;
            nextStates.add(state);
        }
        if(space % 3 == 1 || space % 3 == 2) {   // 左移
```

```
            EPState state = new EPState(this);
            state.puzzles[space] = state.puzzles[space - 1];
            state.puzzles[space - 1] = 0;
            nextStates.add(state);
        }
        if(space > 2) {  // 上移
            EPState state = new EPState(this);
            state.puzzles[space] = state.puzzles[space - 3];
            state.puzzles[space - 3] = 0;
            nextStates.add(state);
        }
        if(space < 6) {    // 下移
            EPState state = new EPState(this);
            state.puzzles[space] = state.puzzles[space + 3];
            state.puzzles[space + 3] = 0;
            nextStates.add(state);
        }

        return nextStates;
    }
}
```

结点

八数码问题的搜索结点的定义如下（只截取部分关键代码），包括了状态：

```
package eightpuzzle;

public class EPNode extends SimpleNode<EPState> {

    @Override
    public boolean equals(Object obj) {     // 状态相同的结点视为相同结点
        if(!(obj instanceof EPNode)) return false;
        EPNode eps = (EPNode) obj;
        return this.getState().equals(eps.getState());
    }

    @Override
    public ArrayList<EPState> next() {  // 获取后继状态
        return getState().next();
    }
}
```

h1搜索

关键代码如下，重写了Astar的h评估函数：

```
public class EP1Search extends Astar<EPNode> {

    @Override
    public int h(EPNode node) {
        int[] puzzles = node.getState().getPuzzles();
        int[] origin = new int[]{1, 2, 3, 8, 0, 4, 7, 6, 5};
        int hvalue = 0;
        for(int i = 0; i < 9; i++) {
            if(origin[i] != puzzles[i] && puzzles[i] != 0) hvalue++;
        }
        return hvalue;
    }
}
```

h2搜索

关键代码如下，重写了Astar的h评估函数：

```
public class EP2Search extends Astar<EPNode> {

    @Override
    public int h(EPNode node) {
        int[] puzzles = node.getState().getPuzzles();
        ArrayList<Integer> list = new ArrayList<>();
        for(int i = 0; i < 9; i++) {
            list.add(puzzles[i]);
        }
        int[] origin = new int[]{1, 2, 3, 8, 0, 4, 7, 6, 5};
        int hvalue = 0;
        for(int i = 0; i < 9; i++) {
            if(origin[i] == 0) continue;
            int index = list.indexOf(origin[i]);
            int drow = index / 3 - i / 3;
            int dcol = index % 3 - i % 3;
            if(origin[i] != puzzles[i]) hvalue += Math.abs(drow) + Math.abs(dcol);
        }
        return hvalue;
    }
}
```

3. 结果分析

运行

```
java -jar Astar.jar
```

点击“h1/h2搜索过程”，可以开始h1/h2搜索；

点击“h1/h2”最优路径，可以演示最优路径；

点击“随机初始化”，可以重新初始化数码。


Astar
h1搜索过程
h1最优路径
随机初始化
h2搜索过程
h2最优路径
open表的结点数=1 图的结点数=1
选择有最小的f值的结点:
index=0 f=14.0 g=0.0 h=14.0
6 7 5
1 2 3
0 8 4
添加新的后继结点:
index=1 f=9.0 g=1.0 h=8.0
6 7 5
1 2 3
8 0 4
添加新的后继结点:
index=2 f=9.0 g=1.0 h=8.0
6 7 5
0 2 3
1 8 4
==========
open表的结点数=2 图的结点数=3
选择有最小的f值的结点:
index=1 f=9.0 g=1.0 h=8.0
6 7 5
1 2 3
8 0 4
添加新的后继结点:
index=3 f=10.0 g=2.0 h=8.0
6 7 5
1 2 3
8 4 0
生成的后继结点是旧结点,
需要更新图:
index=0 f=10.0 g=2.0 h=8.0
6 7 5
最优路径的长度为: 22
index=2 f=9.0 g=1.0 h=8.0
6 7 5
0 2 3
1 8 4
index=5 f=10.0 g=2.0 h=8.0
6 7 5
2 0 3
1 8 4
index=12 f=11.0 g=3.0 h=8.0
6 0 5
2 7 3
1 8 4
index=26 f=12.0 g=4.0 h=8.0
0 6 5
2 7 3
1 8 4
index=47 f=13.0 g=5.0 h=8.0
2 6 5
0 7 3
1 8 4
index=91 f=14.0 g=6.0 h=8.0
2 6 5
1 7 3
0 8 4
index=156 f=15.0 g=7.0 h=8.0
2 6 5
1 7 3
8 0 4
index=293 f=16.0 g=8.0 h=8.0
2 6 5
1 0 3
1 2 3
8 4
7 6 5
h1: 放错位置的码数
open表的结点数: 4487
图的结点数: 12270
1 2 3
8 4
7 6 5
h2: 曼哈顿距离之和
open表的结点数: 420
图的结点数: 1096
open表的结点数=1 图的结点数=1
选择有最小的f值的结点:
index=0 f=10.0 g=2.0 h=8.0
6 7 5
1 2 3
0 8 4
添加新的后继结点:
index=1 f=14.0 g=1.0 h=13.0
6 7 5
1 2 3
8 0 4
添加新的后继结点:
index=2 f=16.0 g=1.0 h=15.0
6 7 5
0 2 3
1 8 4
==========
open表的结点数=2 图的结点数=3
选择有最小的f值的结点:
index=1 f=14.0 g=1.0 h=13.0
6 7 5
1 2 3
8 0 4
添加新的后继结点:
index=3 f=16.0 g=2.0 h=14.0
6 7 5
1 2 3
8 4 0
生成的后继结点是旧结点,
需要更新图:
index=0 f=10.0 g=2.0 h=8.0
6 7 5
最优路径的长度为: 22
index=2 f=16.0 g=1.0 h=15.0
6 7 5
0 2 3
1 8 4
index=5 f=18.0 g=2.0 h=16.0
6 7 5
2 0 3
1 8 4
index=42 f=18.0 g=3.0 h=15.0
6 0 5
2 7 3
1 8 4
index=74 f=18.0 g=4.0 h=14.0
0 6 5
2 7 3
1 8 4
index=103 f=18.0 g=5.0 h=13.0
2 6 5
0 7 3
1 8 4
index=114 f=18.0 g=6.0 h=12.0
2 6 5
1 7 3
0 8 4
index=117 f=18.0 g=7.0 h=11.0
2 6 5
1 7 3
8 0 4
index=121 f=18.0 g=8.0 h=10.0
2 6 5
1 0 3

问题回答

比较搜索效率


Astar
h1搜索过程
h1最优路径
随机初始化
h2搜索过程
h2最优路径
open表的结点数=1 图的结点数=1
选择有最小的f值的结点:
index=0 f=14.0 g=0.0 h=14.0
6 7 5
1 2 3
0 8 4
添加新的后继结点:
index=1 f=9.0 g=1.0 h=8.0
6 7 5
1 2 3
8 0 4
添加新的后继结点:
index=2 f=9.0 g=1.0 h=8.0
6 7 5
0 2 3
1 8 4
==========
open表的结点数=2 图的结点数=3
选择有最小的f值的结点:
index=1 f=9.0 g=1.0 h=8.0
6 7 5
1 2 3
8 0 4
添加新的后继结点:
index=3 f=10.0 g=2.0 h=8.0
6 7 5
1 2 3
8 4 0
生成的后继结点是旧结点,
需要更新图:
index=0 f=10.0 g=2.0 h=8.0
6 7 5
最优路径的长度为: 22
index=2 f=9.0 g=1.0 h=8.0
6 7 5
0 2 3
1 8 4
index=5 f=10.0 g=2.0 h=8.0
6 7 5
2 0 3
1 8 4
index=12 f=11.0 g=3.0 h=8.0
6 0 5
2 7 3
1 8 4
index=26 f=12.0 g=4.0 h=8.0
0 6 5
2 7 3
1 8 4
index=47 f=13.0 g=5.0 h=8.0
2 6 5
0 7 3
1 8 4
index=91 f=14.0 g=6.0 h=8.0
2 6 5
1 7 3
0 8 4
index=156 f=15.0 g=7.0 h=8.0
2 6 5
1 7 3
8 0 4
index=293 f=16.0 g=8.0 h=8.0
2 6 5
1 0 3
1 2 3
8 4
7 6 5
h1: 放错位置的码数
open表的结点数: 4487
图的结点数: 12270
1 2 3
8 4
7 6 5
h2: 曼哈顿距离之和
open表的结点数: 420
图的结点数: 1096
open表的结点数=1 图的结点数=1
选择有最小的f值的结点:
index=0 f=10.0 g=2.0 h=8.0
6 7 5
1 2 3
0 8 4
添加新的后继结点:
index=1 f=14.0 g=1.0 h=13.0
6 7 5
1 2 3
8 0 4
添加新的后继结点:
index=2 f=16.0 g=1.0 h=15.0
6 7 5
0 2 3
1 8 4
==========
open表的结点数=2 图的结点数=3
选择有最小的f值的结点:
index=1 f=14.0 g=1.0 h=13.0
6 7 5
1 2 3
8 0 4
添加新的后继结点:
index=3 f=16.0 g=2.0 h=14.0
6 7 5
1 2 3
8 4 0
生成的后继结点是旧结点,
需要更新图:
index=0 f=10.0 g=2.0 h=8.0
6 7 5
最优路径的长度为: 22
index=2 f=16.0 g=1.0 h=15.0
6 7 5
0 2 3
1 8 4
index=5 f=18.0 g=2.0 h=16.0
6 7 5
2 0 3
1 8 4
index=42 f=18.0 g=3.0 h=15.0
6 0 5
2 7 3
1 8 4
index=74 f=18.0 g=4.0 h=14.0
0 6 5
2 7 3
1 8 4
index=103 f=18.0 g=5.0 h=13.0
2 6 5
0 7 3
1 8 4
index=114 f=18.0 g=6.0 h=12.0
2 6 5
1 7 3
0 8 4
index=117 f=18.0 g=7.0 h=11.0
2 6 5
1 7 3
8 0 4
index=121 f=18.0 g=8.0 h=10.0
2 6 5
1 0 3

可见，h2搜索的图的结点数、open表的结点数总是小于h1搜索，并且在运行时普遍是h2更快，由此可知h2搜索效率更高。

验证凡A*算法挑选出来求后继的点 n ，必定满足：f(n) ≤ f*(S0)。

f*(S0)即为最佳路径的长度。

```
    private void showBestPath(Astar<EPNode> search, TextArea bestPath, Label[] puzzles, int
index) {
        // ...
                for(EPNode node : bestlist) {
                    int[] result = node.getState().getPuzzles();
                    if(node.getFvalue() > bestlist.size()) {
                        throw new Exception("f(n) > f*(s0)");
                    }
                    // ...
                }
        // ...
    }
```

每次运行都没有该异常抛出，再观察搜索过程，可见，挑选出来求后继的点 n 的 f(n) 都满足 f(n) ≤ f*(S0) 。

验证 h1(n) 的单调性，显示凡A*算法挑选出来求后继的点 ni 扩展的一个子结点 nj ，检查是否满足: h(ni) ≤ 1+h(nj)。

由每次初始化后的h1搜索过程可见，凡是挑选出来求后继的点 ni 扩展的一个子结点 nj， 都满足 h(ni) ≤ 1+h(nj) 。

如果将空格看作0，即九数码问题，利用相似的启发函数h1(n)和h2(n)，求解相同的问题的搜索图是否相同?

这要看九数码的最终状态的定义。如果最终状态是:

1 2 3

8 0 4

7 6 5

那么搜索图是相同的，否则搜索图必然有差异。

写出能否达到目标状态的判断方法。

open表为空而未求得解，搜索失败，即达不到目标状态。

4. 结论

评估函数会大大影响A*算法的效率，如本次实验，在最优路径很长的时候，h1搜索明显慢于h2搜索，其搜索图远大于h2，h1搜索图的大小甚至可以达到h2的搜索图的大小的几十倍。

其实八数码问题也可以用广度优先搜索来解决，当 $h(n) \equiv 0$ 时，A*搜索算法就相当于广度优先搜索。但是这样的搜索是很盲目的，效率很低。

所以，精心选择一个好的评估函数十分有益。

主要参考文献

[1]《人工智能基础教程》（第二版）