黑白棋游戏

陈璐 181250012 chenlu@smail.nju.edu.cn

2019 年 10 月 17 日

Contents

1 MiniMax 搜索实现

1.1 基本思想

MiniMax 算法的主要思想为：假定对手会选择所有可选择行动中最有利于自己的选择，在此基础上，选择对自己最有利的行为，具体而言，在每个状态中，agent 都选择最坏情况下能够带来最优收益的行动，而每个行动的收益为执行此行动后所进入的次态的 value，每个状态的 value 通过下式计算：

$$MiniMax(s) = \begin{cases} UTILITY(s) & if\ TERMINAL-TEST(S) \\ max_{a \in ACTION(s)} MINIMAX(RESULT(s,a)) & if\ PLAYER(s) = MAX \\ min_{a \in ACTION(s)} MINIMAX(RESULT(s,a)) & if\ PLAYER(s) = MIN \end{cases} \tag{1}$$

而由于此游戏规模较大，搜索到最终状态所需的时间过长，因此，可对 (1) 式进行修改，使其在游戏结束时或搜索到一定深度后就通过一启发式函数返回对状态的估计值。

综上所述，基于上述式子和贪心的思想，即可编写程序对应实现。

1.2 主要实现过程

对于此程序，可以将其内容与上一小节所描述的过程进行对应，主要分为计算 value 值和贪心选取最优行动两个部分。

1、计算状态对应的 value

```
1  public float miniMaxRecursor(State state, int depth, boolean maximize) {
2      if (computedStates.containsKey(state))
3          return computedStates.get(state);
4      if (state.getStatus() != Status.Ongoing)
5          return finalize(state, state.heuristic());
6      if (depth == this.depth)
7          return state.heuristic();
8      float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
9      int flag = maximize ? 1 : -1;
10     List<Action> test = state.getActions();
11     for (Action action : test) {
12         try {
13             State childState = action.applyTo(state);
14             float newValue = this.miniMaxRecursor(childState, depth + 1, !
                   maximize);
15             if (flag * newValue > flag * value)
16             value = newValue;
17         } catch (InvalidActionException e) {
18             throw new RuntimeException("Invalid action!");
19         }
20     }
21     return finalize(state, value);
22 }
```

Listing 1: 计算每个状态对应的 value

此函数用于计算每个状态的 value 值，主要实现了对 (1) 式中 3 种不同情况进行讨论：

1、结束搜索 (line 4-7)：当游戏结束或者到达设置深度时，即返回此状态对应的 value。

2、递归计算子状态，以对当前状态进行评估 (line 8-21)：判断当前执行行动的为 max 还是 min(line 8-9)，并计算次态的 value(line 14)，最后选择可以选择的最优 value 值作为当前结点的 value，返回此值。

贪心选择最优行动

```
1  public Action decide(State state) {
2      float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
3      List<Action> bestActions = new ArrayList<Action>();
4      int flag = maximize ? 1 : -1;
5      for (Action action : state.getActions()) {
6      try {
```

```
7           State newState = action.applyTo(state);
8           float newValue = this.miniMaxRecursor(newState, 1, !this.maximize);
9           if (flag * newValue > flag * value) {
10            value = newValue;
11            bestActions.clear();
12          }
13          if (flag * newValue >= flag * value) bestActions.add(action);
14      } catch (InvalidActionException e) {
15          throw new RuntimeException("Invalid action!");
16          }
17      }
18      // If there are more than one best actions, pick one of the best randomly
19      Collections.shuffle(bestActions);
20      return bestActions.get(0);
21  }
```

Listing 2: 贪心选取最优行动

此过程和计算 value 时的过程类似，不同的是，在获得所有次态的 value 后，miniMaxRecursor 过程对当前状态 value 进行更新，而 decide 过程只选择最优次态所对应的行动，并返回此行动。

1.3 实现细节

1、使用动态规划思想，避免对状态进行重复计算

算法将已经计算过的状态和所对应的 value 值保存起来，在每次计算状态的 value 时，首先判断此状态是否已被计算 (listing1 line2-3)，如果已经被计算，则直接返回此状态的 value，如果未被计算，则在计算结束后将此状态和对应 value 储存起来 (line 21)，以供后续使用，从而避免了多次对同一状态进行计算，以一定的空间代价换取搜索时间的缩减。

在此处，已被计算的状态和其对应的 value 主要通过 map 数据结构进行存储，经过查询得知，JAVA 中 map 的实现原理为 hash，而其被理论上证明，在基于一定假设的情况下，插入元素和查找元素两个操作的均只需要常数时间代价，因此整体而言效率较高。

但令人困惑的是，作者注释掉了存储操作，并标注上"THIS IS BROKEN DO NOT USE" 来告诉使用者不要使用此功能，但经过测试，去掉注释后程序依旧运行正常，因此，不明白作者的用意何在。

如果我们假设插入和查找功能被正确实现，经过测试，不使用此功能时，单次运行时间平均约为 1.52ms，使用此功能时，为 1.47ms，二者差别不大，分析其原因，可能是因为设置深度较小，因此子问题的重叠部分并也比较少，从而此功能用处不大。

在后续的所有实现过程中，我们仍按照作者的意思将其注释掉，不去使用它，分析时也不考虑它所带来的影响。

```
1  private float finalize(State state, float value) {
2      // THIS IS BROKEN DO NOT USE
3      //computedStates.put(state, value);
4      return value;
5  }
```

Listing 3: 存储 state-value pair

2、通过一个简单的指示位避免了对 max 和 min 状态的判断

按照 (1) 式的过程，在为 min 回合时，选择最小值，在 max 回合时，选择最大值，而实际上，所有元素的最小值对应的为所有元素取负后的最大值，因此设置标志位 flag(line 4)，在 max 回合时其为 1，比较时不取负，按照原值比较，选取其中最大值，在 min 回合时其为 -1，比较时取负 (乘以 flag)，按照取负后的值比较，选取最大值，即原最小值。

2 AlphaBeta 剪枝

2.1 实现过程

修改 miniMaxRecursor 过程，在参数列标中加入 preval，用于表示当前父节点的最优值，也即是过程内原本就存在的 value 变量中的值，在每次循环过程中进行判断，由于当前循环中 value 中的值在不断向不利于父节点的方向发展（如父节点为 max，子结点的 value 就不断减小，父节点为 min，子节点的 value 就不断增大），因此，只要子节点目前的值劣于父节点已经可以获得的最优值，那么就无需对此子节点的后续结点进行搜索，而是直接进行剪枝，返回当前获得的值，由于此值劣于最优解，因此对最终结果不造成影响，具体实现过程如下所示。

```
1  for (Action action : test) {
2      try {
3          State childState = action.applyTo(state);
4          float newValue = this.miniMaxRecursor(childState, depth + 1, !maximize,
               value);
5          if (flag * newValue > flag * value)
6              value = newValue;
7      } catch (InvalidActionException e) {
8          throw new RuntimeException("Invalid action!");
9      }
10     if((0-flag) * value <= (0-flag) * preval) return value; //2
11 }
```

Listing 4: alphabeta 剪枝

如上所示，剪枝过程在 listing4 的第 10 行，其中 0-flag 表示其父节点的 flag 值，此判断语句的意思是：由于 value 值在不断向不利于父节点的方向变化，因此，如果当前值不优于父节点值，那么在整个循环中，value 的值都不会优于父节点的值，即其不会对父节点进行更新，属于无用结点，因此停止探索，直接返回。

2.2 性能分析