School of Data and Computer Science, Sun Yat-sen University

Lab3 Othello Game ($\alpha - \beta$ pruning)

17341015 Hongzheng Chen

September 18, 2019

Contents

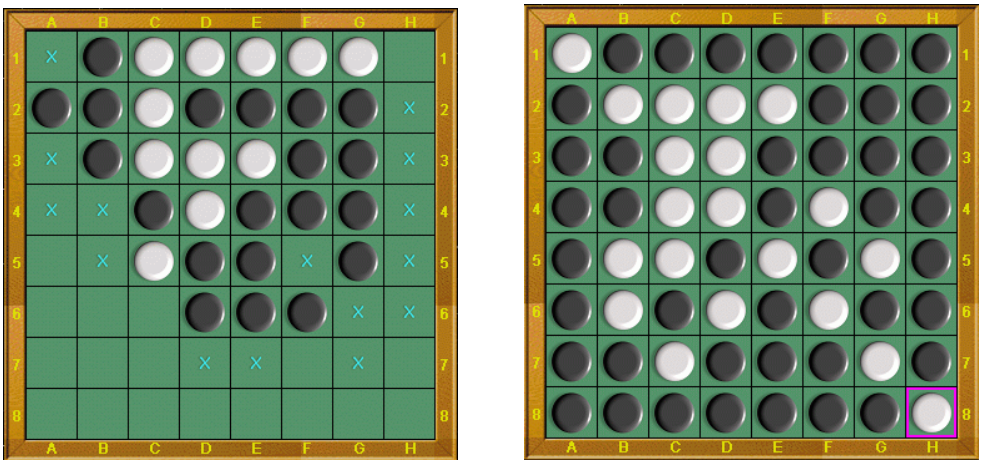

Figure 1: Othello Game

1 Othello

Othello (or Reversi) is a strategy board game for two players, played on an 8×8 uncheckered board. There are sixty-four identical game pieces called disks (often spelled "discs"), which are light on one side and dark on the other. Please see figure 1.

Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color.

The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

You can refer to http://www.tothello.com/html/guideline_of_reversed_othello.html for more information of guideline, meanwhile, you can download the software to have a try from http://www.tothello.com/html/download.html. The game installer `tothello_trial_setup.exe` can also be found in the current folder.

2 Tasks

1. In order to reduce the complexity of the game, we think the board is 6×6.
2. There are several evaluation functions that involve many aspects, you can turn to http://blog.sina.com.cn/s/blog_53ebdba00100cpy2.html for help. In order to reduce the difficulty of the task, I have given you some hints of evaluation function in the file `Heuristic Function for Reversi (Othello).cpp`.
3. Please choose an appropriate evaluation function and use min-max and $\alpha - \beta$ prunning to implement the Othello game. The framework file you can refer to is `Othello.cpp`. Of course, I wish your program can beat the computer.
4. Write the related codes and take a screenshot of the running results in the file named E03_YourNumber.pdf, and send it to ai_201901@foxmail.com.

3 Codes

The codes below only show the search part and the judge part. For other parts, please refer to the attached `Othello.cpp`.

Since this LaTeXtemplate cannot insert Chinese, so I only screenshot the minimax search algorithm here.

```
170     for (k = 0; k < num; k++) /* 尝试在之前得到的num个可落子位置进行落子 */
171     {
172         // 即枚举每一个MAX结点
173         Othello tempBoard;
174         Do thisChoice, nextChoice;
175         Do *pNextChoice = &nextChoice;
176         // 当前局面对方已经落完(MIN node)
177         // thisChoice是己方的落子选择（下一步MAX node），nextChoice是对方的落子选择（下下步）
178         thisChoice = allChoices[k];
179         board->Copy(&tempBoard, board);
            为了不影响当前棋盘，需要复制一份作为虚拟棋盘
180         board->Action(&tempBoard, &thisChoice, player);
            在虚拟棋盘上落子
181         // 递归调用α-β剪枝，得到对手的落子评分
182         // 每次step-1，到0为止，防止陷入无穷递归
183         pNextChoice = Find(&tempBoard, (enum Option) - player, step - 1, -max, -min, pNextChoice,
            who_judge);
184         // 将对方收益取反得到己方收益
185         thisChoice.score = -pNextChoice->score;
186
187         // minimax α-β剪枝
188         // 这里是考虑双方都站在自己的角度观察，希望最大化自己的收益（即都是MAX结点）
189         // 双方都想让极小值极大（从对手给的最坏局面中挑选出最好的）
190         if (min < thisChoice.score && thisChoice.score < max) /* 可以预计的更优值 */
191         {
192             // 当前局面我最坏也不会坏过min（给出下界）
193             min = thisChoice.score;
194             choice->score = thisChoice.score;
195             choice->pos.first = thisChoice.pos.first;
196             choice->pos.second = thisChoice.pos.second;
197         }
198         else if (thisChoice.score >= max) /* 好的超乎预计 */
199         {
200             // 这是由于枚举深度受限所导致的
201             choice->score = thisChoice.score;
202             choice->pos.first = thisChoice.pos.first;
203             choice->pos.second = thisChoice.pos.second;
204             break;
205         } // thisChoice.score <= min
206         /* 不如已知最优值，直接忽略 */
207     }
```

I modify the provided heuristics to 6×6 game, and the code is shown below. There is lots of aprior knowledge and filled with magic numbers!

```
double Othello::MyJudge(Othello *board, enum Option player)
{
  int my_tiles = 0, opp_tiles = 0, i, j, k, my_front_tiles = 0, opp_front_tiles = 0, x, y
      ↪ ;
  double p = 0, c = 0, l = 0, m = 0, f = 0, d = 0;
  enum Option my_color = player;
  enum Option opp_color = (enum Option) - player;

  // eight directions
  int X1[] = {-1, -1, 0, 1, 1, 1, 0, -1};
  int Y1[] = {0, 1, 1, 1, 0, -1, -1, -1};
  // aprior weights for each movement
```

```
int V[6][6] =
  {{20, -3, 11, 11, -3, 20},
   {-3, -7, -4, -4, -7, -3},
   {11, -4, 2, 2, -4, 11},
   {11, -4, 2, 2, -4, 11},
   {-3, -7, -4, -4, -7, -3},
   {20, -3, 11, 11, -3, 20}};

// Piece difference
for (i = 0; i < 6; i++)
  for (j = 0; j < 6; j++)
  {
    // count how many tiles are occupied
    if (board->cell[i][j].color == my_color)
    {
      d += V[i][j];
      my_tiles++;
    }
    else if (board->cell[i][j].color == opp_color)
    {
      d -= V[i][j];
      opp_tiles++;
    }

    // find the difference in eight directions
    if (board->cell[i][j].color != SPACE)
    {
      for (k = 0; k < 8; k++)
      {
        x = i + X1[k];
        y = j + Y1[k];
        if (x >= 0 && x < 6 && y >= 0 && y < 6 && board->cell[x][y].color == SPACE)
        {
          if (board->cell[i][j].color == my_color)
            my_front_tiles++;
          else
            opp_front_tiles++;
          break;
        }
      }
    }
  }

// calculate the proportions
if (my_tiles > opp_tiles)
  p = (100.0 * my_tiles) / (my_tiles + opp_tiles);
else if (my_tiles < opp_tiles)
  p = -(100.0 * opp_tiles) / (my_tiles + opp_tiles);
else
  p = 0;

if (my_front_tiles > opp_front_tiles)
  f = -(100.0 * my_front_tiles) / (my_front_tiles + opp_front_tiles);
else if (my_front_tiles < opp_front_tiles)
  f = (100.0 * opp_front_tiles) / (my_front_tiles + opp_front_tiles);
else
```

```
    f = 0;

  // Corner occupancy
  my_tiles = opp_tiles = 0;
  if (board->cell[0][0].color == my_color)
    my_tiles++;
  else if (board->cell[0][0].color == opp_color)
    opp_tiles++;
  if (board->cell[0][5].color == my_color)
    my_tiles++;
  else if (board->cell[0][5].color == opp_color)
    opp_tiles++;
  if (board->cell[5][0].color == my_color)
    my_tiles++;
  else if (board->cell[5][0].color == opp_color)
    opp_tiles++;
  if (board->cell[5][5].color == my_color)
    my_tiles++;
  else if (board->cell[5][5].color == opp_color)
    opp_tiles++;
  c = 25 * (my_tiles - opp_tiles);

  // Mobility
  // The more tiles can be moved on, the better
  my_tiles = Rule(board, my_color);
  opp_tiles = Rule(board, opp_color);
  if (my_tiles > opp_tiles)
    m = (100.0 * my_tiles) / (my_tiles + opp_tiles);
  else if (my_tiles < opp_tiles)
    m = -(100.0 * opp_tiles) / (my_tiles + opp_tiles);
  else
    m = 0;

  // final weighted score (magic numbers!)
  double score = (10 * p) + (801.724 * c) + (78.922 * m) + (74.396 * f) + (10 * d);
  return score;
}
```

4 Results

I let the two AIs play with each other, and the results are shown below.

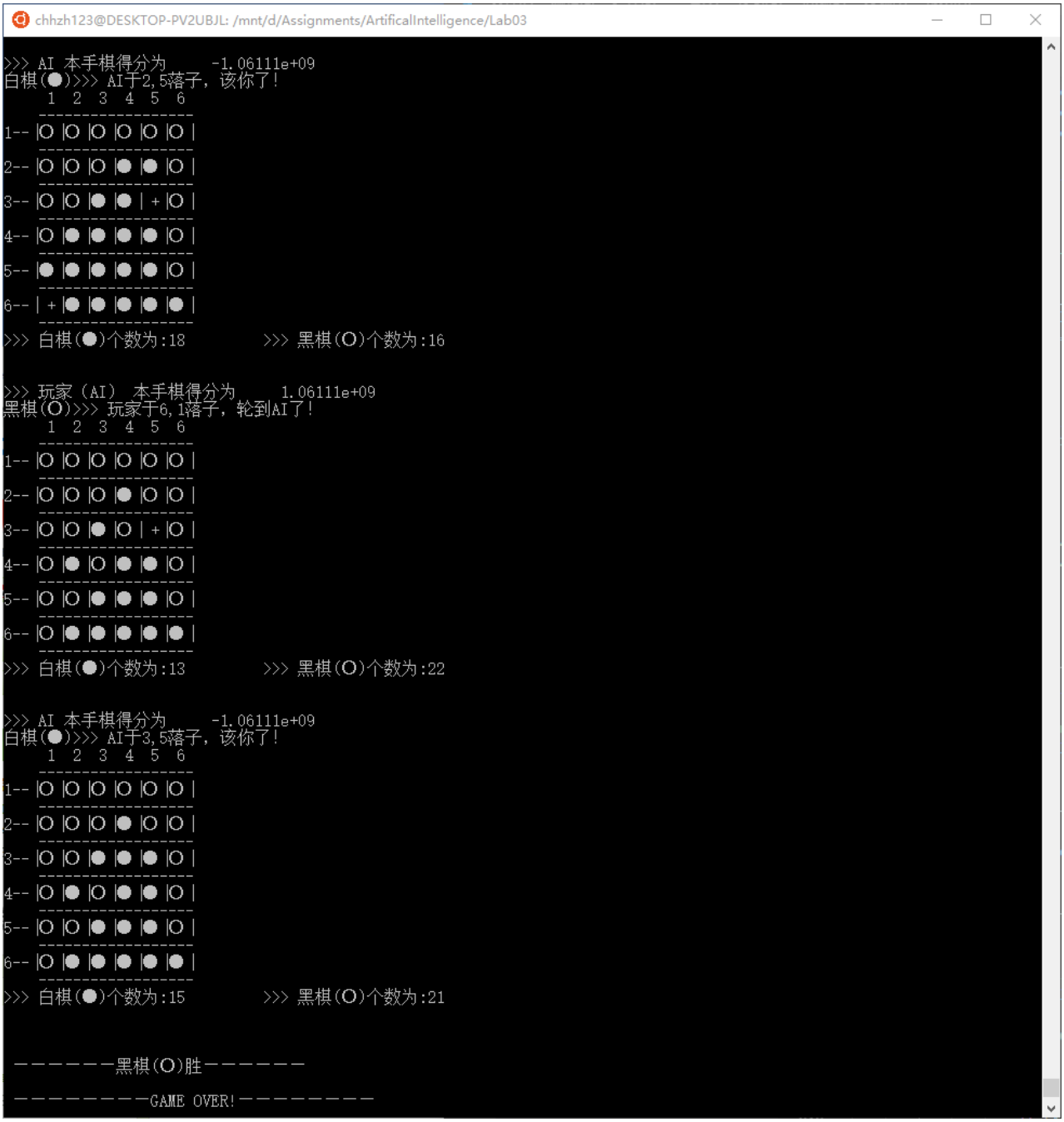

Figure 2: My AI uses **black** disk and beats the computer

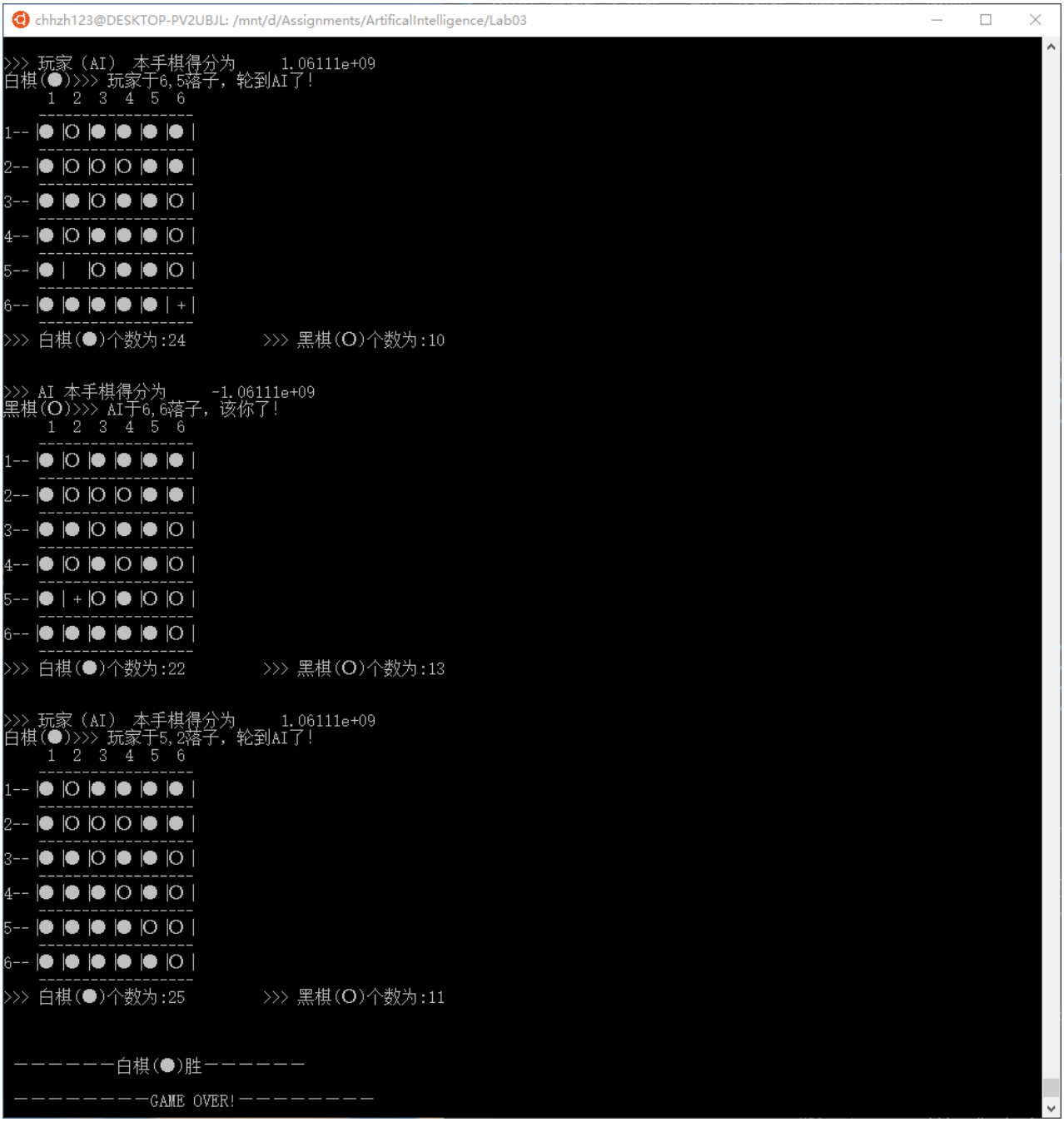

Figure 3: My AI uses **white** disk and beats the computer