Programming project

Artificial Intelligence

The project has 4 parts. In each part you are asked to write two programs. The programming language can be C, C++, JAVA, Python, or anything else approved by the instructor.

Part I: MINIMAX (45%)

Write two programs that get three command line arguments: two file names as the input and output board positions, and the depth of the tree that needs to be searched. The programs print a board position after White plays its best move, as determined by a MINIMAX search tree with the input depth and the static estimation function given in pages 4-12 of this document. That board position should also be written into the output file. In addition, the programs print the number of positions evaluated by the static estimation function (for time complexity analysis) and the MINIMAX estimate for that move. The board position is given by a list of 23 letters. See pages 4-12 of this document for additional information.

1. First program: MiniMaxOpening

The first program plays a move in the opening phase of the game. We request that you name it exactly as **MiniMaxOpening**. This program will accept 3 parameters: the path of the file containing the input board position, the path of the file containing the output board position, and the depth of the search.

For example, the input can be:

(you type:)

```
MiniMaxOpening board1.txt  board2.txt  2
```

(the program replies in this format:)

```
Input position: xxxxxxxxxWxxxxxxBxxxxxx
Output position: xxxxxxxxxWxxWxxxBxxxxxx
Positions evaluated by static estimation: 9.
MINIMAX estimate: 9987.
```

Here it is assumed that the file board1.txt exists and its content is:

```
xxxxxxxxxWxxxxxxBxxxxxx
```

The file board2.txt is created by the program, and its content is:

```
xxxxxxxxxWxxWxxxBxxxxxx
```

(The position and the numbers above may not be correct. They are given just to illustrate the format.)

Please use the move generator and the static estimation function for the opening phase. You are not asked to verify that the position is, indeed, an opening position. You may also assume that this game never goes into the midgame phase in this part.

2. Second program: MiniMaxGame

The second program plays in the midgame/endgame phase. We request that you name it exactly as **MiniMaxGame**. This program will accept 3 parameters: the path of the file containing the input board position, the path of the file containing the output board position, and the depth of the search.

For example, the input can be:

(you type:)

MiniMaxGame board3.txt board4.txt 3

(the program replies in this format:)

Input position: xxxxxxxxxxWWxWWxBBBxxxx
Output position: xxxxxxxxWWWxWWxBBBBxxxx
Positions evaluated by static estimation: 125.
MINIMAX estimate: 9987.

Here it is assumed that the file board3.txt exists and its content is:

xxxxxxxxxxWWxWWxBBBxxxx

The file board4.txt is created by the program, and its content is:

xxxxxxxxWWWxWWxBBBBxxxx

(The position and the numbers above may not be correct. They are given just to illustrate the format.)

Part II: ALPHA-BETA Pruning (35%)

In this part you are asked to write two program that behave exactly the same as the programs in Part I, but implement the ALPHA-BETA pruning algorithm instead of the MINIMAX. Notice that these programs should return the exact same estimate values as the programs of Part I; the main difference is in the number of nodes that were evaluated. We request that you name these programs as **ABOpening** and **ABGame**.

Part III: PLAY A GAME FOR BLACK (10%)

Write the same programs as in Part I, but the computed move should be Black's move instead of White's move. We request that you name these programs as **MiniMaxOpeningBlack** and **MiniMaxGameBlack**.

Part IV: STATIC ESTIMATION (10%)

Write an **improved** static estimation function. The new function should be better than the one which was suggested in the handout. Rewrite the programs of Part I with your improved static estimation function. We request that you name these programs as **MiniMaxOpeningImproved** and **MiniMaxGameImproved**.

Submission

Please prepare your source code and executable binaries compressed in a single .zip file. Your files should be organized as follows: **create a parent project folder "FirstNameLastName_Project2", and inside it create two folders "src" and "bin"**, and put your **source code in "src" folder**, and your **binaries in "bin" folder**. Compress the parent folder containing the entire project into a single .zip file to submit on Blackboard.

Along with your zip file containing your project, please also submit a PDF report on Blackboard as a separate attachment. This report should include at least the following points:

- Please put your full name in the header and filename of the document
- A section describing how to compile and run you code (in case your binaries can't be executed)
- Some test runs of your programs, showing the outputs (with any input board positions)
- Write a short (one or two paragraphs) explanation about why you believe your evaluation function to be an improvement over the static functions given in the handout.

Nine Men's Morris Game, Variant-D

Morris game is a classic and ancient one, and versions have been found way back to 1400BC.

There are many variations. A simple version is Three Men's Morris, which is believed to designed for children and is very similar as *Tic Tac Toe* or *Noughts and Crosses*. The most popular version with adults is Nine Men's Morris. The board is illustrated in *Figure 1*. Each player can have 9 stones (or 9 pieces of any small objects). There are 3^{24} or approximately 2.8×10^{11} states (exceed the max size of a list/vector/array in Java and C++) for this game [1]. We need to figure out a good representation for the board. Obviously, it's not a good idea to perform a blind search.

This game is well studies. The paper from Ralph Gasser in 1996 concludes: when played well, it always comes to a draw [1]. So this is a fair game, no matter which player plays first.

We will use a variant of Nine Men's Morris, Variant-D. The board layout is showed in *Figure* 2.

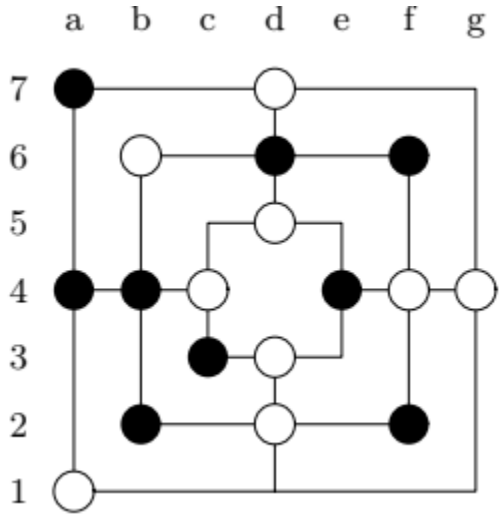

Figure 1. Classic Nine Men's Morris game.

1. Game Rules

The Morris Game, Variant-D, is a board game between two players: White and Black. Each player has 9 pieces, and the game board is as shown below. Pieces can be placed on intersections of lines. There are 23 locations for pieces, and they can be indexed as *Table* 1. The goal is to capture (remove) opponents' pieces by getting three pieces on a single line (a mill). **The winner is the first player to reduce the opponent to only 2 pieces, or block the opponent from any further moves.** If one player has **repeated moves** (repeated moves without removing the opponent's piece), this player will be judged as lost. The game has three distinct phases: opening, midgame, and endgame.

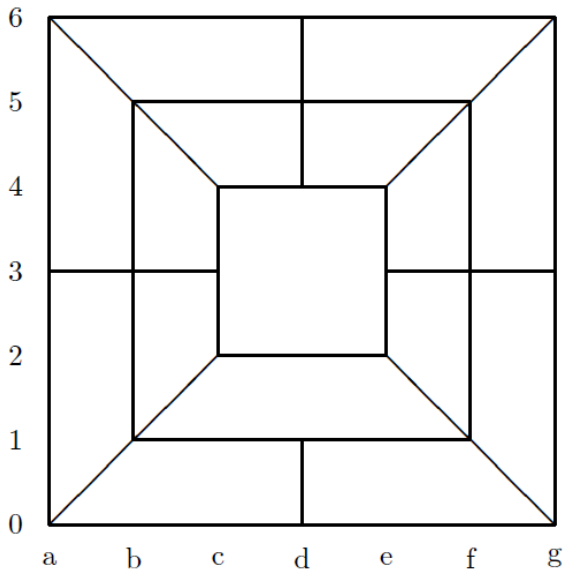

Figure 2. Board for Nine Men's Morris Variant-D.

Opening: Players take turns placing their 9 pieces - one at a time - on any vacant board intersection spot.

Midgame: Players take turns moving one piece along a board line to any adjacent vacant intersection spot.

Endgame: A player down to only three pieces may move (hopping) a piece to any vacant intersection spot, not just an adjacent one.

Mills: At any stage if a player gets three of their pieces on the same straight board line (a mill), then one of the opponent's isolated pieces is removed from the board, unless there is no isolated piece: when the player forms a mill and there is no isolated piece from the opponent, this player can remove any piece on the board. An isolated piece is a piece that is not part of a mill.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a0	d0	g0	b1	d1	f1	c2	e2	a3	b3	c3	e3	f3	g3	c4	d4	e4	b5	d5	f5	a6	d6	g6

Table 1. Indices for board positions.

2. A Computer Program That Plays Variant-D

The basic components of a computer program that plays Variant-D are a procedure that generates moves, and functions for assigning static estimation value for a given position.

2.1 Representing a board position

One way of representing a board position is by an array of length 23, containing the pieces as the letters *W, B, or x*. (The letter *x* stands for a "non-piece"/empty.) The array specifies the pieces starting from bottom-left and continuing left-right for the rows that are explored in a bottom-up fashion. Here are two examples from *Figure* 3:

xxxxxBxWWWWWBBBBxxxxxxx

WxWWxWWWWWBBBBBBBBxxxxx

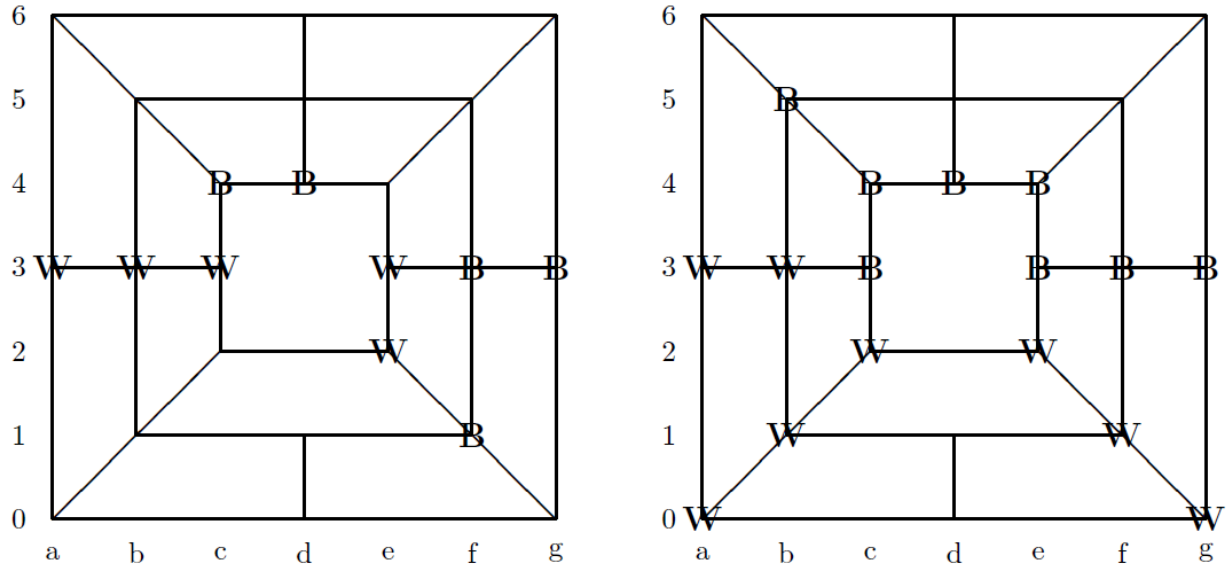

Figure 3. Examples of board representation.

This is the format for input and output. But the way how the board information is maintained in your program could be different from the examples above. For illustration purpose, we will still use the letters *W, B, x* in the following sections.

2.2 Move generator

A move generator gets as input a board position and returns as output a list of board positions that can be reached from the input position. In the following section we will describe the pseudo-code that can be used as a move generator for White. This is the basic implementation, and you need to make your own improvements.

A move generator for Black can be obtained by the following steps. The basic idea is both players will use the same strategy, so we can change the colors and make a move, and then change the colors back.

Input: a board position **b**.

Output: a list **L** of all positions reachable by a black move.

1. Compute the board **tempb** by swapping the colors in **b**: replace each *W* by a *B*, and each *B* by a *W*.

2. Generate **L** containing all positions reachable from **tempb** by a white move (following the description in the next section).

3. Swap colors in all board positions in **L**, replacing *W* with *B* and *B* with *W*.

2.2.1 A move generator for White

A pseudo-code is given for the following move generators: **GenerateAdd**, generates moves created by adding a white piece (to be used in the opening phase). **GenerateMove**, generates moves created by moving a white piece to an adjacent location (to be used in the midgame). **GenerateHopping**, generates moves created by white pieces hopping (to be used in the endgame). These routines get a board as an input and generate a list **L** containing the generated positions as the output. They require a method of generating moves created by removing a black piece from the board when the White form a mill. We name it **GenerateRemove**.

GenerateAdd
Input: a board position
Output: a list **L** of board positions

```
L = empty list;
for each location in the board:
  if board[location] == empty {
    b = a copy of board;
    b[location] = W;
    if closeMill(location, b) generateRemove(b, L);
    else add b to L;
  }
return L;
```

GenerateMove
Input: a board position
Output: a list **L** of board positions

```
L = empty list;
for each location in board:
  if board[location] == W {
```

```
    n = a list of neighbor positions of the current location;
    for each j in n: // j is the index of the position
      if board[j] == empty {
        b = a copy of board;
        b[location] = empty;
        b[j] = W;
        if closeMill(j, b) GenerateRemove(b, L);
        else add b to L;
      }
  }
return L;
```

GenerateMovesOpening

Input: a board position
Output: a list **L** of board positions

```
return the list produced by GenerateAdd applied to the board.
```

GenerateMovesMidgameEndgame
Input: a board position
Output: a list L of board positions

```
if the board has 3 white pieces
  return the list produced by GenerateHopping applied to the
  board.
else
   return the list produced by GenerateMove applied to the
   board.
```

GenerateHopping
Input: a board position
Output: a list L of board positions

```
L = empty list;
for each location α in board:     // α is the index of the
                                  // location
  if board[α] == W {
    for each location β in board:  // β is the index of the
                                  // location
      if board[β] == empty {
        b = a copy of board;
```

```
        b[α] = empty;
        b[β] = W;
        if closeMill(β, b) generateRemove(b, L);
        else add b to L;
      }
  }
return L
```

GenerateRemove
Input: a board position and a list **L**
Output: positions are added to L by removing black pieces

```
for each location in board:
  if board[location] == B {
    if not closeMill(location, board) {
      b = a copy of board;
      b[location] = empty;
      add b to L;
    }
  }
if no positions were added (all black pieces are in mills)
  for each location in board:
    if board[location] == B {
      b = a copy of board;
      b[location] = empty;
      add b to L;
    }
```

2.3 neighbors and closeMill

The proposed coding of the methods neighbors and closeMill is by "brute force". The idea is as follows:

neighbors
Input: a location j in the array representing the board
Output: a list of locations in the array corresponding to j's neighbors

```
switch(j) {
  case j==0 (a0): return [1,3,8].   // These are d0, b1, a3
  case j==1 (d0): return [0,4,2].   // These are a0, d1, g0
  etc.
}
```

closeMill
Input: a location j in the array representing the board and the board **b**
Output: true if the move to j closes a mill

```
C = b[j];    // C must be either W or B, and cannot be x
switch(j) {
  case j==0:    // this is a0
    if (b[1] == C and b[2] == C)
        or (b[3] == C and  b[6] == C)
        or (b[8] == C and  b[20] == C)
      return true;
    else
      return false;
  case j==1:   // this is d0
    if (b[0] == C and b[2] == C)
      return true;
    else
      return false;
  etc.
}
```

2.4 Static estimation

The following static estimation functions are proposed. Given a board position b compute:
numWhitePieces = the number of white pieces in **b**.
numBlackPieces = the number of black pieces in **b**.
L = the MidgameEndgame positions generated from **b** by a black move.
numBlackMoves = the number of board positions in ***L***.

A static estimation for MidgameEndgame:

```
if (numBlackPieces ≤ 2) return 10000;
else if (numWhitePieces ≤ 2) return -10000;
else if (numBlackMoves==0) return 10000;
else return  1000 * (numWhitePieces – numBlackPieces) –
    numBlackMoves;
```

A static estimation for Opening:
return **numWhitePieces – numBlackPieces;**

3. Basic Strategies

You can make improvements on the static estimations in your game. And this is left for your design. We can get some hints from the way how we play this game.

Here are some suggested strategies.

3.1 Maintaining mobility

Intersections have the most mobility, then sides and corners have less – it is unlikely to be trapped in the second ring. This is illustrated in *Figure* 4.

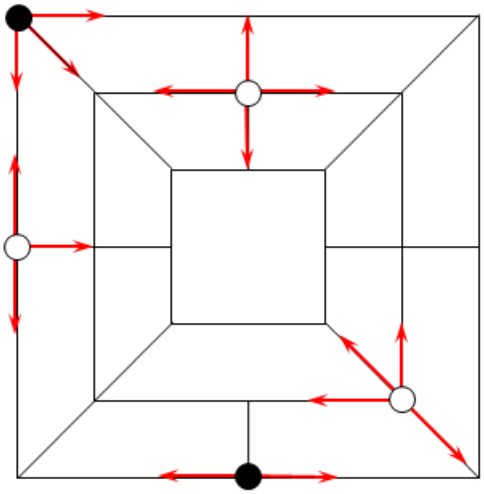

Figure 4. Mobility of different positions.

3.2 Double mills

A double mill can result in the loss of the opponent with every move. This is illustrated in *Figure* 5. The black player can move the piece on d5 to b5, back and forth.

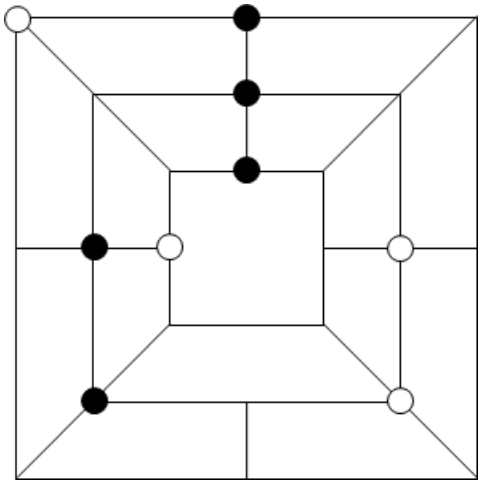

Figure 5. An example of double mills.

3.3 End game

Be careful of removing pieces when it is close to the end game: the transition from 4 to 3 can be critical. In *Figure 6*, now the black side just formed a mill, if this player chooses to remove the up-left white piece a0, the black side will lose: white piece on c3 can be moved to f5 (hopping in end game), and form a mill, and then remove a black piece.

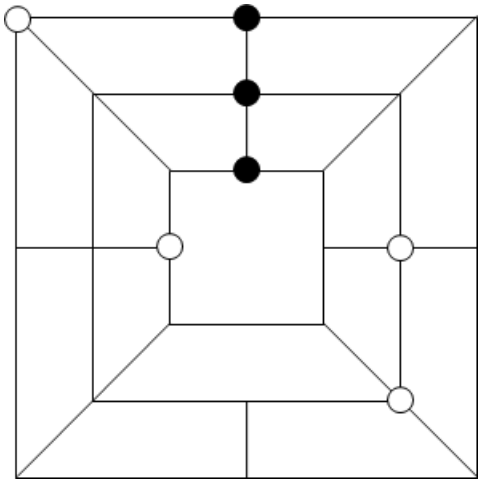

Figure 6. An example when it is close to the end game.

Reference

[1]. Gasser, Ralph. "Solving Nine Men's Morris." Games of No Chance, Vol. 29, 1999, pp. 101-113.

Some material courtesy of Michelle Sharp, Walter Voit, Kedar Naidu, Haim Schweitzer, James Latham and others