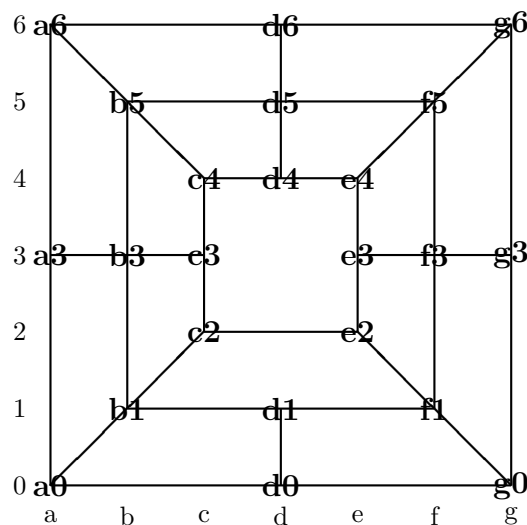
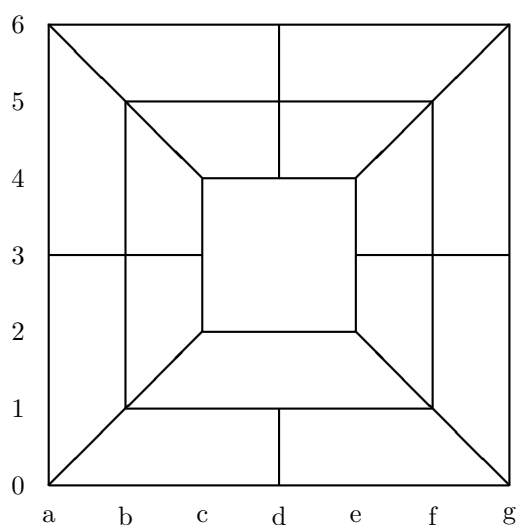


Morris Game, Variant-D

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



Game rules

The Morris Game, Variant-D , is a variant of Nine Men's Morris game. It is a board game between two players: White and Black. Each player has 9 pieces, and the game board is as shown above. Pieces can be placed on intersections of lines. (There are a total of 23 locations for pieces.) The goal is to capture 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. The game has three distinct phases: opening, midgame, and endgame.

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 spot.

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

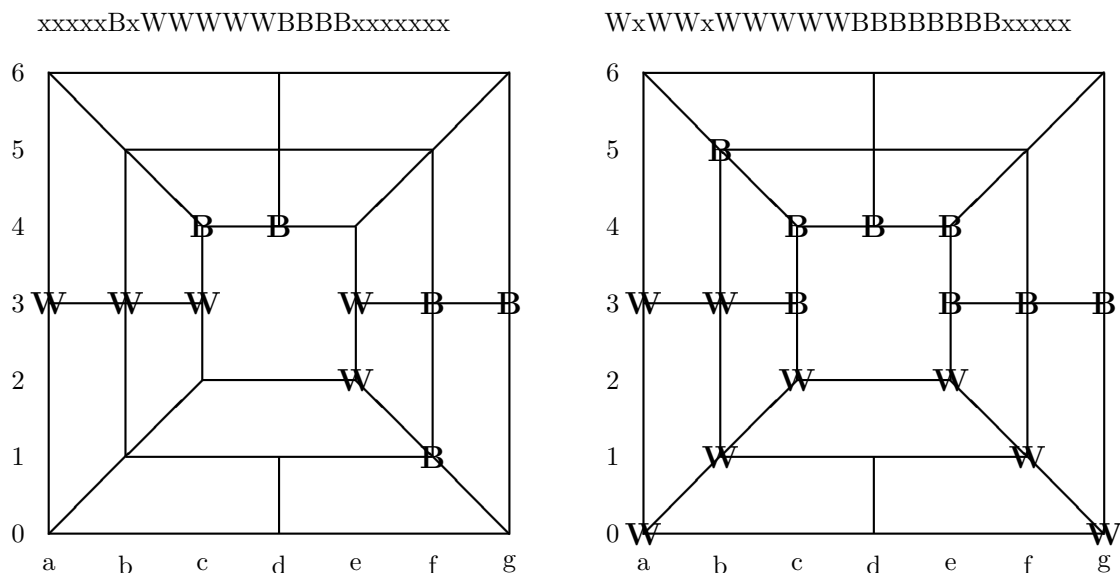
Mills: At any stage if a player gets three of their pieces on the same straight board line (a mill), then one of the opponent's isolated pieces is removed from the board. An isolated piece is a piece that is not part of a mill.

A computer program that plays Variant-D

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

Representing board positions

One way of representing a board position is by an array of length 23, containing the pieces as the letters W, B, x . (The letter x stands for a “non-piece”.) The array specifies the pieces starting from bottom-left and continuing left-right bottom up. Here are two examples:



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 next section we describe a pseudo-code that can be used as a move generator for White. A move generator for Black can be obtained by the following steps.

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

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). **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 as an input a board and generate as output a list L containing the generated positions. They require a method of generating moves created by removing a black piece from the board. We name it **GenerateRemove**.

GenerateMovesOpening**Input:** a board position**Output:** a list L of board positionsReturn the list produced by **GenerateAdd** applied to the board.**GenerateMovesMidgameEndgame****Input:** a board position**Output:** a list L of board positionsif the board has 3 white pieces Return the list produced by **GenerateHopping** applied to the board. Otherwise return the list produced by **GenerateMove** applied to the board.**GenerateAdd****Input:** a board position**Output:** a list L of board positions

L = empty list

for each location in board:

if board[location] == empty {

b = copy of board; b[location] = W

if closeMill(location, b) generateRemove(b, L)

else add b to L

}

return L

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

L = empty list

for each location α in boardif board[α] == W { for each location β in board if board[β] == empty { b = copy of board; b[α] = empty; b[β] = W if closeMill(β , 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 = list of neighbors of location
    for each j in n
      if board[j] == empty {
        b = copy of board; b[location] = empty; b[j]=W
        if closeMill(j, 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 = copy of board; b[location] = empty
      add b to L
    } }

```

If no positions were added (all black pieces are in mills) add b to L.

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. Cannot be x.

```
switch(j) {  
    case j==0 (a0) : return true if  
        (b[1]==C and b[2]==C)  
        or (b[3]==C and b[6]==C)  
        or (b[8]==C and b[20]==C)  
    else return false  
    case j==1 (d3) : return true if  
        (b[0]==C and b[2]==C)  
    else return false  
    etc.  
}
```

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)
```