Lines of Action: Description of Text-Based Implementation

Albert Shau

2006.12.4

1 Purpose of this Document

This document was written to describe the C code used to implement the game Lines of Action in the Gamesman system. It will describe the data structures and major functions used, as well as the reasoning behind some of the design decisions. This is not meant to be a tutorial on C and is not meant to give playing strategies for the game. It is also not meant to describe the gamesman core. It is written so that somebody already familiar with C will be able to understand the game module code, allowing him/her to add features or make changes. More specifically, this document describes the essential functions DoMove, GenerateMoves, Primitive, and many of the helper functions they use.

2 Game Rules

A short description of game rules will be given here to give context to what the major functions are supposed to do. The game is played on a checkerboard. There are two players – black and white – who take turns moving their pieces. A player may move his/her piece in a straight line in any direction including diagonals. To find how far that piece can move, draw a straight line from one side of the board to the other in the direction you wish to move. This line is called the line of action. Count the number of pieces along that line. This is how far that piece may move. You may jump over your own pieces, but not over your opponent's pieces. You may capture your opponent's piece by landing directly on it. The object of the game is to move all your pieces so that they are all connected. Diagonals count as being connected. If a move causes both players' pieces to be all connected, the player who just moved wins. If a player only has one piece left, that player wins. More information can be found here

3 Board, Piece, Move, and Position Representation

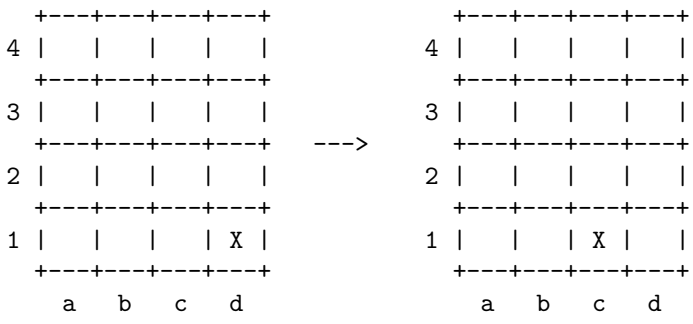
Pieces are represented by characters. Black is 'X', white is 'o', and a blank square is ' '. The board is represented by an array of characters. There is a global array called gBoard that is the size of the board + 1. The last element of the called is the terminator. Each square on the board maps to a specific index for gBoard. The squares are numbered from left to right and top to bottom. The diagram below illustrates.

```
+---+---+---+---+
| 0 | 1 | 2 | 3 |
+---+---+---+---+
| 4 | 5 | 6 | 7 |
+---+---+---+---+
| 8 | 9 |10 |11 |
+---+---+---+---+
|12 |13 |14 |15 |
+---+---+---+---+
```

A person refers to a square on the board using a letter and a number. The columns are labeled 'a', 'b', 'c', etc and the rows are labeled 1, 2, 3, etc. This is just like a chessboard. The starting board is shown as an example.

```
  +---+---+---+---+
4 |   | X | X |   |
  +---+---+---+---+
3 | o |   |   | o |
  +---+---+---+---+
2 | o |   |   | o |
  +---+---+---+---+
1 |   | X | X |   |
  +---+---+---+---+
    a   b   c   d
```

Moves are represented by a starting square and an ending square. For example, if we wanted to move a piece from the bottom right corner one square to the left, a person would refer to this as d1c1. In the game module, the letter-number representation is replaced with the board numbering discussed above, making the move 1514.



The move d1c1 (represented as 1514)

Translating from the module move representation to the people friendly letter-number representation and vice versa is done in moveHash() and moveUnhash(). The function moveHash() changes a string to an integer.

```
startSquare = input[0] - 'a' +
              (gBoardHeight - (input[1] - '0'))*gBoardLength;
```

calculates the start square from input[0], which is a letter, and input[1], which is the number. These are also ascii characters, so - 'a' and - '0' are used to change an 'a' to 0 and a '1' to 1. (gBoardHeight - (input[1] - '0'))*gBoardLength calculates the first square in the row that the piece is in. For example, if the square is a1 and we have a 4x4 board, the previous expression is $3*4 = 12$, which is the bottom left square. We then add input[0] - 'a', which is how far to the right that square is in that row. Translating a move integer into a string uses the same math except solved for input[0] and input[1].

A position is captured by the gBoard and the player turn. A hash function maps those two pieces of information to a unique number. This is done with the generic hash functions.

4 DoMove

This function is simple. It assumes the move passed to it is a valid move for the position it is passed. From the move we know the start square and the end square. All that is needed is to change gBoard[startSquare] to blank and gBoard[endSquare] to the piece at the startSquare. The player turn is then switched to the other player and these two are hashed using generic hash. This new position is then returned.

5 GenerateMoves

This function is divided into addPieceMoves(), which adds moves that a specified piece can do, and addMovesInDirection(), which adds a move if a specified

piece can move in a specified direction. Other important functions used are piecesInLineOfAction(), and goInDirection().

The algorithm for this function starts by going through the entire board, looking for pieces belonging to the player whose turn it is. For each piece found, we add all the moves that piece has.

```
for (i = 0; i < gBoardSize; i++) {
   if (gBoard[i] == playerColor)
     addPieceMoves(i, playerTurn, &moves);
}
```

When adding moves for a piece, we look in all eight directions and see if we can move that piece in that direction. If we can, that move in that direction is added to the move list.

```
for (d = UP; d <= UPLEFT; d++) {
    addMovesInDirection(d, boardSquare, playerTurn, moves);
}
```

Much of the work takes place in addMovesInDirection. The function addMovesInDirection() first uses piecesInLineOfAction() to find how far the piece can move in the given direction. It uses this to calculate the destination square. It then moves in that direction one square at a time until it hits that destination square. If a piece of the opposite color is in the way, or if it runs into an edge, that move is illegal. Also, if the destination square contains a piece of your own color, that move is illegal. If it can get to the destination square without any of those conditions occuring, the move is added to the move list.

```
void addMovesInDirection(direction, startSquare, playerTurn, movelist)
{
  moveLength = piecesInLineOfAction(direction, startSquare);
  endSquare = startSquare + moveLength*goInDirection(direction);

  i = 0;
  /* go in that direction one step at a time.  If you hit an edge or a
     piece of the opposite color, you cannot move further in this direction */
  while (i < moveLength && not on an edge))
    {
      boardSquare += goInDirection(direction);
      if (gBoard[boardSquare] != otherPlayerColor)
i++;
      else
break;
    }
  if ((boardSquare == endSquare) && (gBoard[endSquare] != playerColor))
    add this move to the movelist;
}
```

The function goInDirection() takes a board square and a direction. It returns the number that must be added to that board square to move a piece one square in the specified direction. For example, to move a piece from square 1 right one square, we must do 1 + 1 = 2. goInDirection() would therefore return 1. Note that this function can return some funny things. For example, we could pass it board square 0 and the direction left. It would return -1, since going left one square usually means subtracting 1. However, it doesn't make sense to go left if you are on the left edge. This is not a problem because goInDirection is never passed a direction that would have the piece moving off the edge of the board. This is because the onEdge() function is used in conjunction with goInDirection() in all the functions. onEdge() is used to check if a piece is on an edge or not.

As the name suggests, piecesInLineOfAction() returns the number of pieces along a line of action. It does this by moving the in one direction until it hits the edge of the board. It then starts moving the in the opposite direction one square at a time. While it is moving this time it keeps track of the number of pieces it encounters. It then returns this number.

6 Primitive

Primitive takes in a position and returns whether it is a win, loss, tie, or if the game is undecided for the black player (player1). First it makes two arrays, one for each player. These arrays are as long as the number of pieces left for their corresponding pieces. They are called chainOfBlacks and chainOfWhites. For example, if there are 3 white pieces left and 4 black pieces left, chainOfBlacks will be 4 integers and chainOfWhites will be 3 integers. These arrays hold board square numbers. They are meant to represent a chain of connected pieces. If an element does not contain a board square, it's value if -1. Each array starts off with its first element set to the board square the first piece found on the board. We then loop through each board square on the board. If it contains a piece, we check if that piece is connected to any of the pieces in its corresponding array. If it is, it is added to the array. We keep looping through the board until no pieces are added to either array for the entire loop. The pseudocode below should clarify a bit.

```
while(not doneChecking) {
  doneChecking = TRUE;
  for (i=0; i < boardSize; i++) {
    if board[i] is a black piece {
      for every non -1 element of chainOfBlacks {
if board[i] is connected to the chain {
  add board[i] to chain;
  doneChecking = FALSE;
  board[i] = BLANK;
}
      }
```

```
    }
    do the same thing for white pieces;
  }
}
```

After this process is done, if a chain is full (no -1s), that means all the pieces of that color are connected. This means we know if all the blacks are connected and if all the whites are connected. Using those facts and the player turn, we can easily determine what we should return.

7 Variants and Game Specific Options

Only one variant, misere, is implemented. Only one option is implemented. The user can change the board size. The board can also be made into a rectangle, though this gives one player a pretty big advantage. These are done in a very simple manner so they will not be discussed in this document.

8 Known Bugs

If the game is solved many times in the same session, a segmentation fault will probably occur. If you resize the board, then immediately resize it again, an error will occur. The output is Aborted (core dumped).