POLARIZED LADDER

BY MICHAEL LAVOIE (9778004)

RUNNING THE APPLICATION/GAME

This game was programmed using Java Version 1.7.0_17-b02.

For Eclipse:

1. Create a new Java Project.
2. Right click the ‘src’ folder in the Package Explorer and select ‘import’.
3. Select ‘General’ and then ‘Archive File’ and click ‘Next’.
4. Browse the file system to find ‘a2_9778004_Jar.jar’.
5. Click ‘Finish’.
6. Run ‘Game.java’ in the ‘com.pl.app’ package.
7. Play the game from the console of Eclipse.

For CMD (Windows):

1. Run the game from the console by using the following command.

   java –jar “Path to the file on your system”\a2_9778004_Jar.jar

2. Play the game through the Command-line interface.

Note: These steps assume that Java is already installed and that it is part of the “Path” Environment Variable on your Windows OS.

DETAILS OF IMPLEMENTATION

Polarized Ladder (PL) is a game that I have implemented in Java that uses bit strings as a way to store player moves along with winning and neutralizing patterns. I won’t go into too much detail about the implementation but I will try to convey the concept as clearly as possible.

Since the game board has 49 slots, I used a *long* to store the bit strings (*long* is 64 bits).

For example, an empty board could be represented as the following in java:

```
long empty_board = 0B0_000_00000_0000000_000000000_00000000000_0000000000000L;
```

Note: Each underscore denotes a new line on the game board. It is used for readability and understanding. The first zero to the right of the 'B' is equivalent to the position G7 on the game board.

By using this method of storing game-states we save on computation costs. If we wanted to check if a certain pattern exists on the game board, we would simply create a bit mask and perform the '&' bitwise operation.

As an example, let's say that we have the following variable declaration:

```
long player_ones_moves = 0B1_00000000011_0000000000011L;
```

This is equivalent to having the following game board:

```
7                   .
6                 .   .   .
5               .   .   .   .   .
4             .   .   .   .   .   .   .
3           x   .   .   .   .   .   .   .   .
2       x   x   .   .   .   .   .   .   .   .   .
1   x   x   .   .   .   .   .   .   .   .   .   .   .

    A   B   C   D   E   F   G   H   I   J   K   L   M
```

Notice that the pattern above is a winning pattern. How would we know that it is a winning pattern if we were given the player_ones_moves variable?

Answer: Create a bit mask (in this case it would be exactly the same bit string as player_ones_moves) and perform the '&' bitwise operation. If the bit mask matched the player_ones_moves variable, we would then have a bit string with a cardinality of 5. If the '&' operation resulted in a bit string with a cardinality of anything other than 5, we would know that the pattern did not match.

The following relates the game board indices with the bit string indices (which is how it was implemented):

Game Board:

```
                            48
                        45  46  47
                    40  41  42  43  44
                33  34  35  36  37  38  39
            24  25  26  27  28  29  30  31  32
        13  14  15  16  17  18  19  20  21  22  23
    0   1   2   3   4   5   6   7   8   9   10  11  12
```

Bit String:

```
0  B | 48 | … | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | L
```

If we wanted to retrieve a specific bit to find out if it has been set (if a player has a token there), we would use the right bit shift operation along with another bit mask.

To demonstrate the power of the bit shift operator when used to check for a winning solution, I will give another example. Given the following game board state:

```
7                   .
6                 .   .   .
5               .   .   .   .   .
4             .   .   .   .   .   .
3           .   .   .   .   x   .   .   .   .
2         .   .   .   .   x   x   .   .   .   .   .
1       .   .   .   .   x   x   .   .   .   .   .   .   .

        A   B   C   D   E   F   G   H   I   J   K   L   M
```

By analyzing this pattern, you can see that it is just the player_ones_moves variable (from above) left shifted 4 times. If we had an initial bit mask starting at position zero, we would just have to loop through, one shift at a time and checking to see if the cardinality is equal to 5 when the bit mask is ANDED with player_ones_moves.

DESCRIPTION OF THE ALGORITHM AND THE HEURISTIC

To find the best move, my implementation of the solution uses the MiniMax algorithm along with Alpha-Beta pruning to reduce the amount of sub trees that are traversed. No specific data structure was used for the tree. Since recursion is often used to implement a tree data structure, I used recursive calls to traverse an imaginary tree (the tree is automatically generated through recursion).

The heuristic function returns one of four types of possible values:

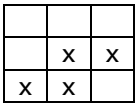
1. If the current board state is full, it returns 0 (which means the score is neutral).
2. If player one has won the game in its current state, it returns (100 * (level + 1)).
    a. Player one is the MAX player, so the score is positive if he/she has won.
    b. The value 100 is a value determined by trial and error.
    c. I multiply (level + 1) with 100 because I want higher-level wins to have precedence over lower-level wins (I use the level variable in my program to denote the number of levels left to traverse). A higher level means that it is a shorter path to a win.
    d. I add 1 to level simply because level becomes 0 at one point and multiplying 0 with 100 will produce 0 (which means neutral, and that should not be the case).
3. If player two has won the game in its current state, it returns -(100 * (level + 1)).

a. Player two is the MIN player, so the score is negative if he/she has won.
b. The rest of the reasoning is the same as #2(b,c,d).

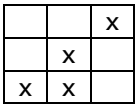
4. This value is the most complicated and will be explained below.

Explanation of 4: I have created a method that counts the number of partial ladders on the game board with a specified number of tokens for a specific player. For example:

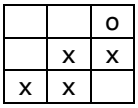
The method accepts two arguments, the player to count the ladders for and the number of tokens that we should consider.



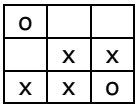
= 1 possible ladder(s) with 4 tokens



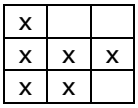
= 1 possible ladder(s) with 4 tokens, etc…



= 0 possible ladder(s) with 4 tokens *Note: This one wouldn't be counted because it isn't a possible ladder since the other player blocked it.*



= 0 possible ladder(s) with 4 tokens *Note: This one wouldn't be counted because it is neutralized (Unless of course it's grounded).*



= 2 possible ladder(s) with 4 tokens, etc…

I used small test cases to demonstrate for now, but in the game it checks the whole board and counts all of the ladders.

Here is the java code that calculates the score:

1. (board.getNumOfLadders(Player.PLAYER_ONE, 4) * 2)
2. (board.getNumOfLadders(Player.PLAYER_ONE, 3))
3. (board.getNumOfLadders(Player.PLAYER_TWO, 4) * 2)
4. (board.getNumOfLadders(Player.PLAYER_TWO, 3)

1. Gets the number of partial ladders in the game board with 4 tokens (for player one) and multiplies the result by 2 because the number of partial ladders with 4

tokens is worth more than the number of partial ladders with 3 tokens (1 ladder with 4 tokens is better than 1 ladder with 3 tokens).

2. Gets the number of partial ladders in the game board with 3 tokens (for player one). Line number 1 added with line number 2 gives the total score for player one.
3. The same as number 1 but with player two instead of player one.
4. The same as number 2 but with player two instead of player one. Line number 3 added with line number 4 gives the total score for player two.
   By subtracting player two's total score from player one's total score, we get the final calculated score for the game board (If it's negative, the current board state is in favor of player two because it would mean that player two has more possible ladders).