Polarized Ladder

Samer Ayoub 26750265 Hussain Qabbani 25609879

Final Project Report

- **Our game is fast, intelligent and unbeatable**

1. **Specifications:**
   Java\jdk1.8.0_51-b16

2. **Launching the application:**
   - **CMD(Windows):**
     {filepath}\java -jar PolarLadderGame.jar
   - **In Eclipse, import the zip file:**
     1) **Go to File -> Import. An import dialog panel will appear.**
     2) **Select Existing Projects into Workspace.**
     3) **Click the radio button next to Select archive file and click the Browse button.**
     4) **Find the archive file "PolarLadderGame.Zip" on your hard disk.**
        Click Open to select it.
     5) **The project name will appear in the box, already checked.**
        Click Finish to perform the import.

3. **How to start:**
   see Fig.1

```
Enter zz to quit the game at any time, else to start a new game
s
------------------------ <<<< Welcome to Polarized Ladder >>>> ------------------------
Choose a play mode
1       --> human VS A.I.
2       --> A.I. VS human
else    --> human VS human
1
Enter human player's name :
Samer
Level   |       average thinking time in sec\move
  1     |               0.032 (Default)
  2     |               0.080
  3     |               2
  4+    |               more than 15
_____________________________________
Enter A.I. playing strength level: 3
Samer (Human) starts the game with symbol X Against  (A.I.) with symbol O

-----------------------------------------------------------------------
Enter a letter [a-m] (lower or upper case) + number [1-7] like g5 or B2
```

1

4. Implementation details

```
Enter human player's name :
Samer
Enter human player's name :
Hussain
Samer (Human) starts the game with symbol X Against Hussain (Human) with symbol O

-----------------------------------------------------------------
Enter a letter [a-m] (lower or upper case) + number [1-7] like g5 or B2


7                           .
6                       .   .   .
5                   .   .   .   .   .
4               .   .   .   .   .   .   .
3           .   .   .   .   .   .   .   .   .
2       .   .   .   .   .   .   .   .   .   .   .
1   .   .   .   .   .   .   .   .   .   .   .   .   .
        A   B   C   D   E   F   G   H   I   J   K   L   M
_____________________________________________________________

Samer (Human)   Enter a valid move:     g3
```

2

a) Game Board : see Fig.2

Each move is mapped to a number on the Game board from 1-49, to be used as an offset to construct a player bit board

The mapping from a move to a position is not hardcoded, however it is determined using a function that takes the row and column positions as arguments
Main reason for enumerating from right to left is to be compatible with the least significant bit of a 49-bit Long

For example:
H2 is corresponding to shifting a 1 (0B1L) to the left 17 times.
H2 corresponds to column = 5, row = 2

$$pos(H2) = (row - 1) \times (15 - row) + col - row + 1$$
$$= (2 - 1) \times (15 - 2) + 5 - 2 + 1 = 17$$

Board Positions enumeration

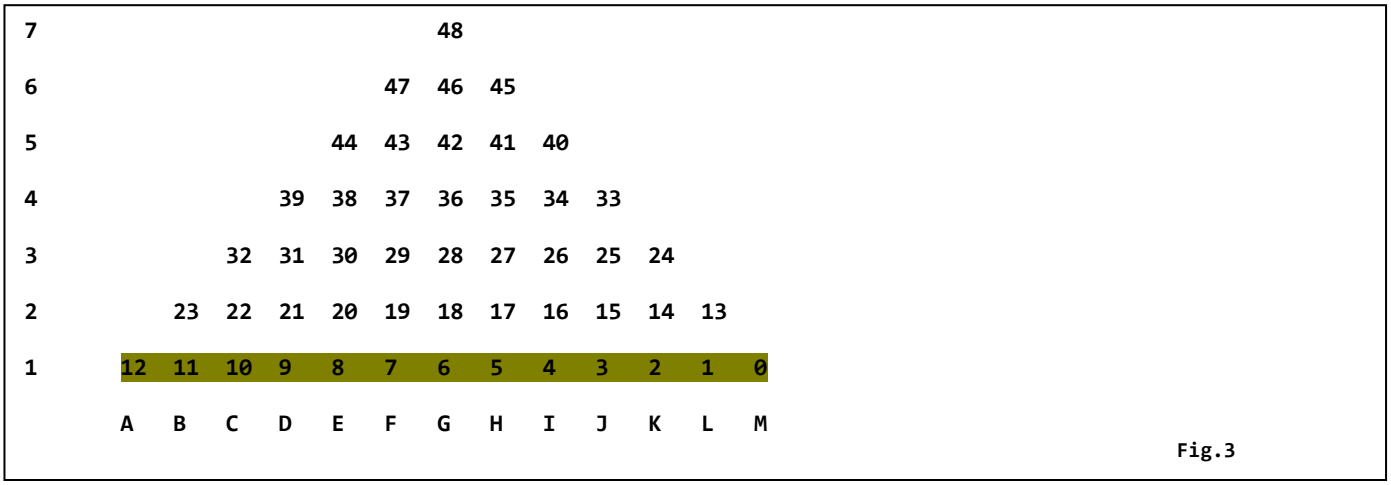


Fig.3

b) State (board) Representation:

By far, the hardest part that consumed most of the design phase. After a lot of reading and negotiating we decided to represent the state using bit board technique.

"A bit board is a data structure designed for efficiently encoding game boards as sets of bits, first used for computer chess in the 1950s (Frey, 1977). Rather than allocating an integer for each board cell to store the value of any piece there, each cell is assigned a bit indicating the presence or absence of a piece (or pattern) there, requiring only a fraction of the memory"[1].

For example, a player board is conveniently packed into a single 64-bit long integer (only 49 bits are used)

$$(13 + 11 + \ldots + 3 + 1 = 7^2 = 49)$$

13 bits for row 1
11 bits for row 2
9 bits for row 3
7 bits for row 4
5 bits for row 5
3 bits for row 6
1 bit for row 7
In fig.3
User_1 (plays with X) would be represented as:
0B0_000_00000_0001000_000110000_10000000000_1000000000000L
Whereas
User_2 (plays with O) would be represented as:
0B1_000_00000_0000001_000001000_00000001000_0000000000011L

enumerated from right to left and down up (Fig.2)

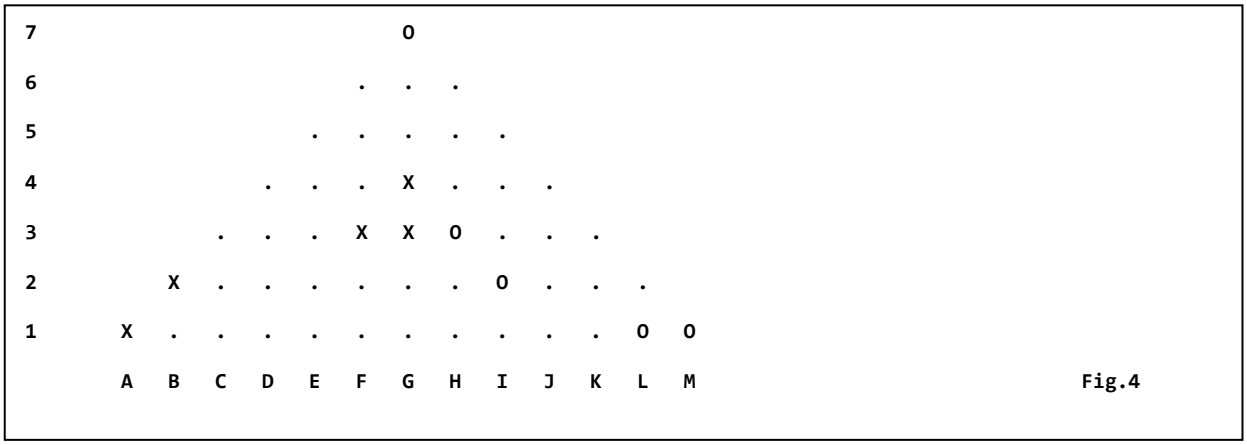


Fig.4

1 means a position is occupied by the player,
0 means empty or occupied by the opponent
Hence the whole state is a pair of (player bit board, opponent bit board)

Current game state is:
(0B0_000_00000_0001000_000110000_10000000000_1000000000000L ,
0B1_000_00000_0000001_000001000_00000001000_0000000000011L)

Note:
In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code[2].

c) Ladders (winning & neutral patterns):

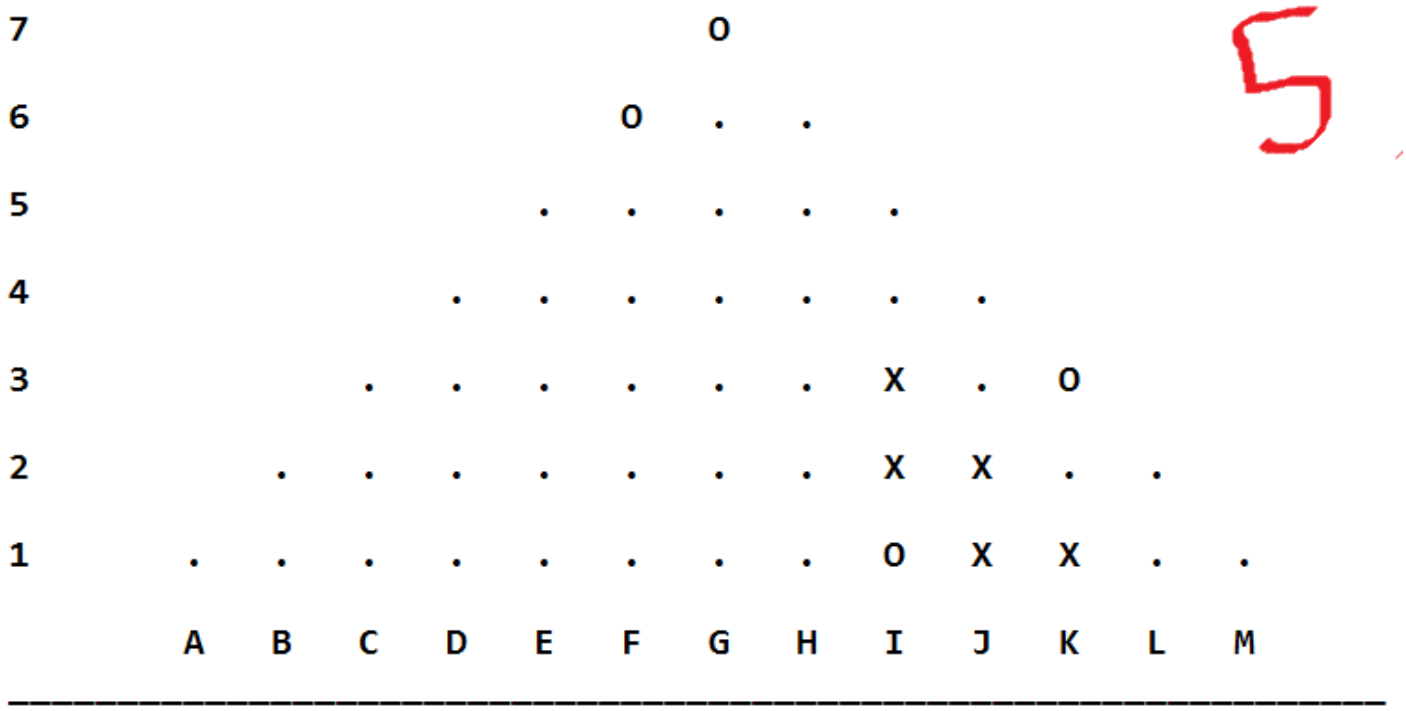
- **There exist 50 winning patterns, 32 of which can be neutralized by 2 moves from the opponent.**
- **Only 10 winning patterns are hardcoded (20%), 2 for each level (base of ladder lies on levels 1, 2, 3, 4, 5); one going left and another going right. The remaining 40 patterns are generated at the start of each game, by applying left and right shift operations on the hardcoded 10 patterns.**
- **Likewise, only 8 neutral patterns are hardcoded (25%), 2 for each of levels 1, 2, 3, 4 and the remaining are auto generated by shift operations.**
- **The function that build the remaining patterns by bit-operations from the initial patterns is so sophisticated "generateLadders()" in class player.**
- **Each player start with his own List of 50 available ladders, as the playing proceeds, the number of ladders decreases and their worth values changes.**

d) How to determine a win:

Two conditions must be satisfied:

- **check if winning pattern exists:**
  A player bit board is masked with a winning pattern, using bitwise operations and then the mask is compared to the winning pattern:
  this.bitBoard & winPattern == winPattern

- **check if this winning pattern is not neutralized:**
  The opponent bit board is masked with the corresponding neutral pattern (if exists) of the winning pattern found, thus compared to the neutral pattern
  opponentBoard & neutralPattern != neutralPattern



In Fig.5

Is it a win for Player (X) ?? No, it has been Neutralized by Opponent (O)

Current game state is:

0B0_000_00000_0000000_000000100_00000001100_1000000001100L ,
0B1_100_00000_0000001_000000001_00000000000_0000000010000L

Player bit-board is masked with all 50 winning patterns, it yields a winning pattern → TRUE, Not yet

Opponent bit-board is then masked with the neutral pattern associated with the found winning pattern, → TRUE (it is neutralized)

(TRUE, FALSE) is what required for a win

e) Scoring criteria (Heuristic):

➢ **We combine all these characteristics to create our heuristic:**

- **Position Score: each position has a score directly proportional to the number of ladders (winning patterns) that use that position, for example G3 is the most strategic position as it comprises 10 ladders. See (Fig. 6)**

7						2							
6					3	4	3						
5				4	5	6	5	4					
4			4	7	8	8	8	7	4				
3		3	7	8	9	10	9	8	7	3			
2	2	4	6	7	8	8	8	7	6	4	2		
1	1	2	2	3	4	4	4	4	3	2	2	1	
	A	B	C	D	E	F	G	H	I	J	K	L	M

Fig.6

Number of winning patterns passing through each position

- **Ladder Score: each available unconstructed (empty) ladder worth 2 points to distinguish it from unavailable or destroyed ladder, for each added move that constructs this specific ladder, the value of the ladder is doubled each time 1 move is added, and keep doubling:**
  2 points if it is available, then 4, 8, 16, 32, 64(win if not neutralized)
- **If a ladder is destroyed (the opponent used 1 of its 5 positions), the player who owns it lose all the previously assembled points for that ladder.**
- **it takes 2 steps for a ladder to be neutralized by the opponent:**
  first step from opponent cost half the ladder score,
  second step destroys it (remove it from this player Ladders list) completely along with the other half and all its score.
- **Player score:**
  Summing up all Ladders' scores in this player's List of Ladders (each ladder has a score that signifies it's worth according to how constructed and neutralized it is.
- **A move is then evaluated by the difference between the player's score and his opponent due to this move (explained below), the greater the difference, the better the move**
- **The Min Max Algorithm is then used with this heuristic to pick the best move up to a certain depth.**

- **How a move is evaluated in this Heuristic?**

  we used 2 different heuristics which use the former characteristics, so they are basically similar:

  [1] For basic game level (1 depth of thinking):

  we measure the increase in the Player(A.I.) score, by aggregating the increase of values of all the Ladders associated with (due to) this move,

  This is done by:

  i) aggregating the scores change of the winning patterns of the player (A.I.) due to this move, AND

  ii) aggregating the scores change of the neutralized and destroyed winning patterns of the opponent (Human player) due to this move.

  This method is very fast (at most 16 milliseconds/move) and very smart as well, we could never beat it and it sometimes beat us.

  [2] For upper levels (2, 3, 4..... depth of thinking)

  We use the Min-Max Algorithm with Alpha-Beta pruning

  - **At each game instance (Game node in the recursive tree of Min-Max) the player (A.I) score is calculated according to the following heuristic:**

    a = A.I. score after the move

    b = A.I. score before the move

    c = Human score after the move

    d = Human score before the move

    Node value = (a - b) + (d - c) = a - b - c + d =

    (d - b) + (a - c) = scoreBefore + scoreAfter see Fig.7

    Example:

    if A.I. was 16 before playing its move then 24 (must increase for increasing a Ladder value) after A.I. played its move i.e. a = 24, b = 16 and

    if Human was 18 before the move then 14 after the move (must decrease because opponent move will either neutralize or destruct one of his/her ladders) i.e. c = 14, d = 18

    Node Value = a - b - c + d = (24 - 16) A.I. increase + (18 - 14) Human decrease = 12 points won because of this move

  - **If during thinking ahead and evaluating a player situation, a winning pattern was found, it is calculated differently with overestimating the score according to depth (depth+10)*1000, ladder values are worthless in this case as direct winning (or loss) needs to be taken (or avoided).**
  - **Hence the win (or loss) at upper depths of the tree (imminent win) is better than the win (or loss) at lower depths (belated win), yes of course !!**

```
// copy the parent, apply a move from legal moves, hence it is a child
protected GameNode getNextChild(String newMove, int depth){
    GameNode childNode = new GameNode(game);
    childNode.type = this.type.equals(NodeType.MAX) ? NodeType.MIN:NodeType.MAX;
    childNode.setPcMove(newMove);
    // players' scores before the move
    // notice that player becomes opponent and opponent becomes player after applying the move
    int scoreBefore = childNode.player.getScore() - childNode.opponent.getScore();
    childNode.getGame().updateState(newMove);
    childNode.setNodeScore(depth, scoreBefore);
    return childNode;
}

public void setNodeScore(int depth, int scoreBefore){
    int tempScore = 0 ;
    if(game.evaluateState() == true){
        tempScore = (depth+10)*1000;
    } else if(game.getStrength() == 1){ // for basic level only
        tempScore = calcNodeScore();
    } else {            // for higher levels
        //players' scores after the move
        int ScoreAfter = player.getScore() - opponent.getScore();
        tempScore = scoreBefore + ScoreAfter;
    }
    if(type.equals(NodeType.MAX)){
        nodeScore = -tempScore;
    } else {
        nodeScore = tempScore;
    }
```

f) Min Max algorithm:

- **implemented using recursion and Alpha-Beta pruning[3].**
- **state space size is 49!**
- **starts with Alpha = -100000 (represents -infinity) and Beta = 100 000 (infinity)**
- **allows for searching the search state tree to a certain depth, to avoid exhaustive search whenever needed.**
- **Each state of the game (Game Node is saved) exploiting memory to save time in adding and removing a move. (That's how it is fast)**
- **At any point of the game, the maximum number of game nodes expanded and saved is 50**

The value of a leaf node is determined by the previous heuristic if not a direct win or loss, (10+tree depth)*1000 if winning (a positive value) , same value but negated for a loss. Hence the shallower the tree, the better, i.e. winning at depth 10 is better than winning at depth 40 (maximum depth is 49).

g) Levels of difficulty (thinking depths):

The game can be played up to 49 level of difficulty (thinking ahead for the next 49 moves) but it would take 49! simulations.

Moreover iterative deepening is used but commented, in case it is needed to play according to a given time limit.

In conclusion, we believe our solution is rich, fast and intelligent, we are proud of our product and very thankful for the opportunity given to us to learn and apply the theoretical knowledge acquired in class.

References:

[1] BITBOARD METHODS FOR GAMES, Cameron Browne1, QUT, Brisbane, Australia

[2] Oracle Java Documentation, Underscores in Numeric Literals,

https://docs.oracle.com/javase/8/docs/technotes/guides/language/underscores-literals.html

[3] Artificial Intelligence "Foundations of computational agents",

by David L.Poole and Alan K.MackWorth, procedure MinmaxAlphaBeta, page 432

"We certify that this submission is the original work of members of the group and meets the faculty's Expectations of Originality"

1. *Samer Ayoub* ***26750265***
2. *Hussain Qabbani* ***25609879***

Date April 12, 2016

(Please See the 2 Expectations of originality attached)