Atropos AI Game Play

CS 440 Programming assignment 3
Name: Ziran Min
Email: minziran@bu.edu
BU ID: U59274427
Date: 04/25/2018

Problem Definition Method and Implementation

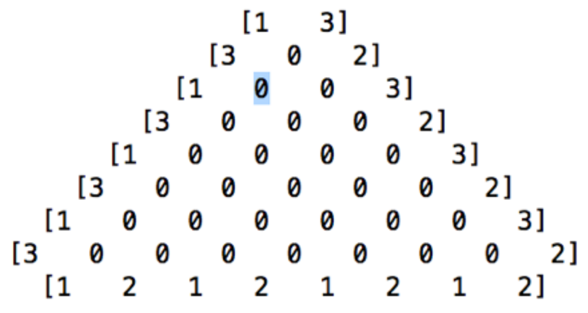
Atropos is a two-player game invented in BU Computer Science Department by Kyle Burke and Prof. Shang-Hua Teng. In this project, I will implement a strategy to play this game intelligently against random player and other people's strategies by using adversarial search algorithms.

Method and Implementation

My main strategy is minimax algorithm with alpha-beta pruning. In order to implement a minimax searching algorithm, I need to convert the game board to a readable input, write several help function, and then design an evaluator function.

Step 1: Converting game board to a 2-d list.

The game board we play game on shown in the terminal is the following:



```
                 [1   3]
               [3   0   2]
             [1   0   0   3]
           [3   0   0   0   2]
         [1   0   0   0   0   3]
       [3   0   0   0   0   0   2]
     [1   0   0   0   0   0   0   3]
   [3   0   0   0   0   0   0   0   2]
     [1   2   1   2   1   2   1   2]
```

In order to convert it into an easily readable input for my later functions, I write my first part of algorithm to change it as a vertically reversed 2-d list:

```
[ [1, 2, 1, 2, 1, 2, 1, 2],
  [3, 0, 0, 0, 0, 0, 0, 0, 2],
  [1, 0, 0, 0, 0, 0, 0, 3],
  [3, 0, 0, 0, 0, 0, 2],
  [1, 0, 0, 0, 0, 3],
  [3, 0, 0, 0, 2],
  [1, 0, 0, 3],
  [3, 0, 2],
  [1, 3] ]
```

```
     [1, 2, 1, 2, 1, 2, 1, 2]
   [3, 0, 0, 0, 0, 0, 0, 0, 2]
     [1, 0, 0, 0, 0, 0, 0, 3]
      [3, 0, 0, 0, 0, 0, 2]
        [1, 0, 0, 0, 0, 3]
          [3, 0, 0, 0, 2]
            [1, 0, 0, 3]
              [3, 0, 2]
                [1, 3]
```

In this way, by knowing the coordinates (c, x, y, z) of a circle in the original board, I can easily locate it in a 2-d list, as index (x, y). For example, saying the highlighted circle in the original board is (0, 6, 1, 2), if the 2-d list is called board, then this corresponding circle in the 2-d list is board[6][1].

Step 2: Help functions for building evaluator and minimax searching

The followings are helper functions I write to prepare for evaluator and minimax searching.

1. AllNeighbors(board, lastPlay): finding all adjacent circles next to the previous move.
2. UncoloredNeightbors(board, lastPlay): finding all uncolored adjacent circles (potential next moves for the opponent) next to the previous move
3. ColoredNeightbors(board, lastPlay): finding all colored adjacent circles next to the previous move
4. Lose(board, move): testing whether a move will cause player to lose
5. AllUncolored(board): finding all uncolored circles on the board
6. AvailableMoves(board, lastPlay): finding all available moves for the next player
7. BoundedUncolored(board, circle, visited) : for a specific circle, finding its all uncolored neighbors, those neighbors' uncolored neighbors, and so on, until we finding a biggest uncolored area that contains the specific input circle and this area is bounded by all colored circles.

Step 3: Evaluator

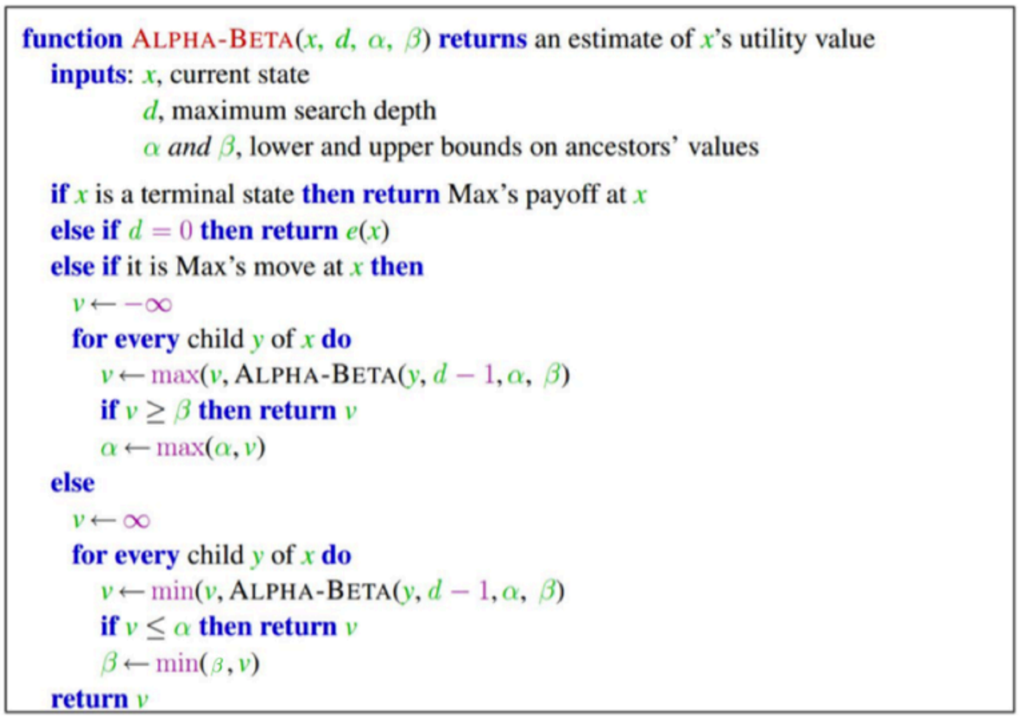
The evaluator function is to evaluate a score of a circle to determine how good or bad a move is.

Then main idea of choosing a best move is to force the opponent to fall into a trap where is the only circle he/she can color (BoundedUncolored() is a key function to find a potential trap). If a move could cause a trap later, it gains high scores because trap has six colored neighbor, the opponent has higher probability to create a fatal three color triangle.

After deciding where to put next move for creating trap, I should consider what color to choose. In order to decrease my probability to create a fatal triangle, I decide to choose the color that my move's neighbors have the most.

Step 4: Minimax with alpha-beta pruning

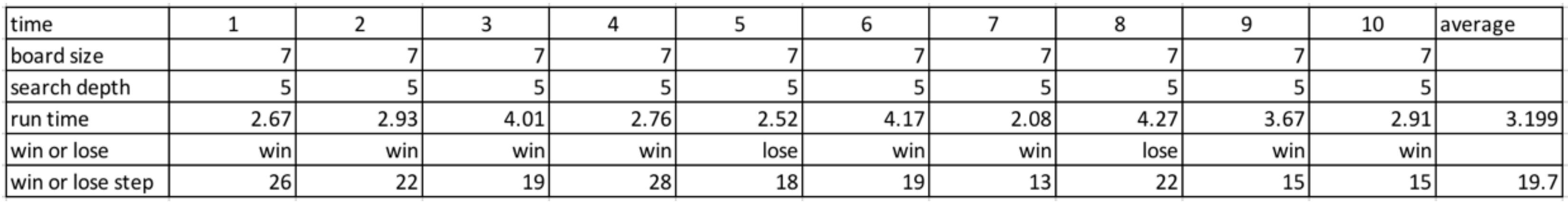
By having evaluator function, I get the value of each leaf node in a Minimax tree. After set a depth for minimax search, I can write and run a Minimax function to find the best move for myself. The following picture is the pseudo code for Minimax searching with alpha-beta pruning



```
function ALPHA-BETA($x$, $d$, $\alpha$, $\beta$) returns an estimate of $x$'s utility value
  inputs: $x$, current state
          $d$, maximum search depth
          $\alpha$ and $\beta$, lower and upper bounds on ancestors' values
  if $x$ is a terminal state then return Max's payoff at $x$
  else if $d = 0$ then return $e(x)$
  else if it is Max's move at $x$ then
    $v \leftarrow -\infty$
    for every child $y$ of $x$ do
      $v \leftarrow \max(v, \text{ALPHA-BETA}(y, d-1, \alpha, \beta)$
      if $v \geq \beta$ then return $v$
      $\alpha \leftarrow \max(\alpha, v)$
  else
    $v \leftarrow \infty$
    for every child $y$ of $x$ do
      $v \leftarrow \min(v, \text{ALPHA-BETA}(y, d-1, \alpha, \beta)$
      if $v \leq \alpha$ then return $v$
      $\beta \leftarrow \min(\beta, v)$
  return $v$
```

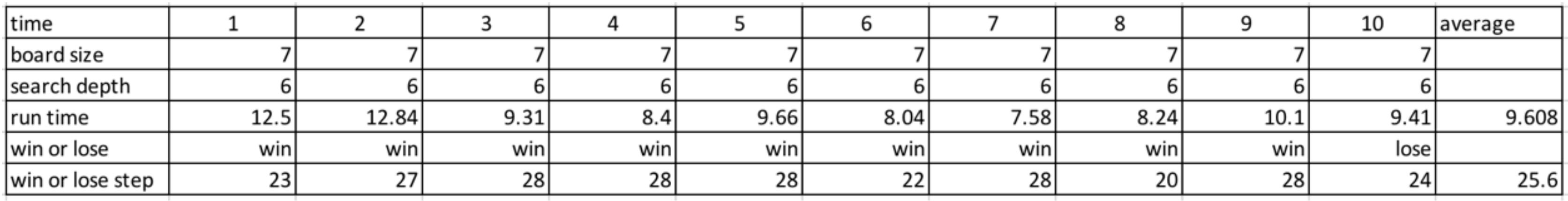
Experiment, Results, and Discussion

In order to see how intelligent my algorithm is, I run my program against a default random player. I set the depth of minimax search to be 5 or 6 and the size of board to be 6 or 7. For each combination, I run my algorithm against the default random player for ten times. The following tables are the results.



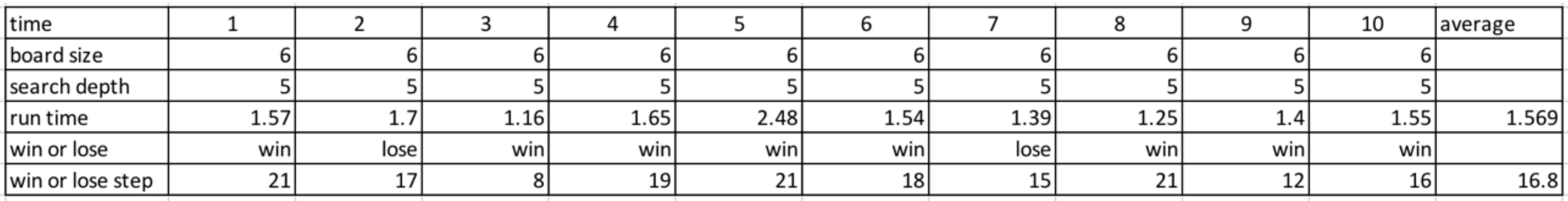
time	1	2	3	4	5	6	7	8	9	10	average
board size	7	7	7	7	7	7	7	7	7	7	
search depth	5	5	5	5	5	5	5	5	5	5	
run time	2.67	2.93	4.01	2.76	2.52	4.17	2.08	4.27	3.67	2.91	3.199
win or lose	win	win	win	win	lose	win	win	lose	win	win	
win or lose step	26	22	19	28	18	19	13	22	15	15	19.7

When search depth = 5 and board size = 7, I won 8 times out of 10.



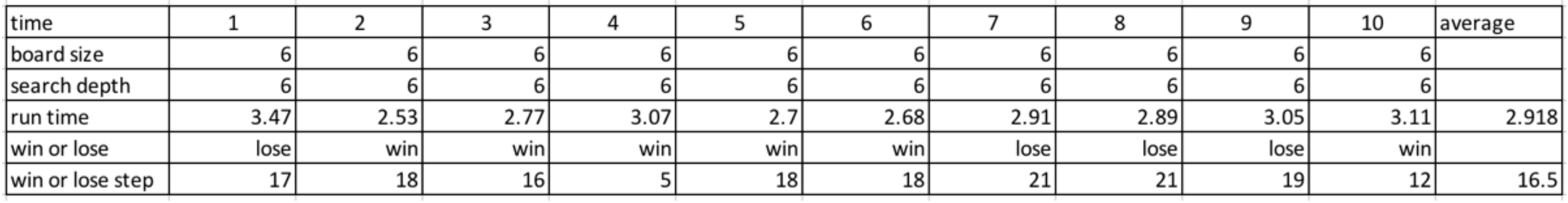
time	1	2	3	4	5	6	7	8	9	10	average
board size	7	7	7	7	7	7	7	7	7	7	
search depth	6	6	6	6	6	6	6	6	6	6	
run time	12.5	12.84	9.31	8.4	9.66	8.04	7.58	8.24	10.1	9.41	9.608
win or lose	win	win	win	win	win	win	win	win	win	lose	
win or lose step	23	27	28	28	28	22	28	20	28	24	25.6

When search depth = 6 and board size = 7, I won 9 times out of 10.



time	1	2	3	4	5	6	7	8	9	10	average
board size	6	6	6	6	6	6	6	6	6	6	
search depth	5	5	5	5	5	5	5	5	5	5	
run time	1.57	1.7	1.16	1.65	2.48	1.54	1.39	1.25	1.4	1.55	1.569
win or lose	win	lose	win	win	win	win	lose	win	win	win	
win or lose step	21	17	8	19	21	18	15	21	12	16	16.8

When search depth = 5 and board size = 6, I won 8 times out of 10.



time	1	2	3	4	5	6	7	8	9	10	average
board size	6	6	6	6	6	6	6	6	6	6	
search depth	6	6	6	6	6	6	6	6	6	6	
run time	3.47	2.53	2.77	3.07	2.7	2.68	2.91	2.89	3.05	3.11	2.918
win or lose	lose	win	win	win	win	win	lose	lose	lose	win	
win or lose step	17	18	16	5	18	18	21	21	19	12	16.5

When search depth = 6 and board size = 6, I won 6 times out of 10.

Overall, my algorithm did pretty well, especially when depth = 6 and board size = 7, my winning percentage is 90%. Moreover, I found that when board size = 7, the more search depth I set to the minimax algorithm, the longer time it took to find best move and the higher the winning percentage was I also found that the smaller the size of a board was, the less time it took to finish the game. Both above findings are reasonable, because more searching depth gives minimax algorithm to go through more levels and more leaf nodes, and the larger the board is, the more circles for player to choose.

However, when board size = 6, search depth = 6 lost more times than search depth = 5. This makes me surprise at first. Later, I guess this might because the size of board is small and default player plays randomly, this random strategy brings higher lucky chance to take a nice step to force me to a trap.

I also try to set the depth to be 7, but it takes much longer to finish a game. Also, when size = 7, depth = 6 takes more moves to win on average than depth = 5. Therefore, I conclude that for my strategy, searching depth = 5, and let board size larger than 6 is better choice.

On the other hand, I also tried to let myself play against my algorithm, setting search depth = 5 and board size = 7. Unfortunately, I only won 6 times out of 10. Compared to computer algorithm, as human, I indeed take longer time to think of a good move. So again, my algorithm is pretty smart.

Conclusion

By mainly using the idea of trap forcing and less different color around own move as evaluator and minimax searching strategy with alpha-beta pruning, I build a smart AI Atropos player that can beat not only a random player but also myself for several times.

There are also many things can be improved in this project. In experiment section, I should run my algorithm against the default random player for more than 10 times in each parameter combination, then I can get better sense of the effect of search depth and board size. Furthermore, there are other situation I could consider in my evaluator, especially like the color limitation for a node in a corner or the side of a boundary.

Credits and Bibliography

http://turing.plymouth.edu/~kgb1013/combGames/atropos.html
https://www.youtube.com/watch?v=zDskcx8FStA
https://www.youtube.com/watch?v=jhki0o54_xY&t=197s
https://github.com/nikojpapa/CS440/tree/master/P4/p4
https://github.com/chenhuiyi/atropos-game-ai-player