Intelligent Search & Game

Game Program - Andantino

Zhaolin LI
6229455
October 27, 2019

Table of Contents

1. Introduction

Andantino is a 2-player game played on a hexagonal tiled infinite plane. The player takes turns dropping stones of their color to reach the winning condition by making a five-in-a-row line of his own stones or fully enclosing one or more enemy stones. The first stone must be placed at the center of the board. The second stone played must be adjacent to the first stone. Thereafter, the new dropped stone must be touched at least two existed stones (Smith, 1995).

In this report, the game Andantino was programmed in Python 3.7 with intelligent search algorithms. Intelligent search algorithms building agent that act rationally by searching solutions leading to win. Minimax and Alpha-beta pruning algorithms were implemented and validated.

2. Method

Programming the andantino consisting of Designing board and coordinates, implementing winning conditions, and implementing intelligent search algorithms. An evaluation function was built for achieving algorithms.

2.1 Design board and coordinates

This game has a 10*10 hexagonal board. Each coordinate consist of three cube coordinates represented with q, r and s that in the range of (-9, 9). Coordinates start from the center of the board (q=0, r=0, s=0) and extend following six directions. The six directions are represented with cube coordinates (Table 1), which are represented with arrows ordered in anticlockwise direction. Coordinate ranges from *spot1* to *spot2* in the direction *d* follows *Equation 1*, where $(q1, r1, s1)$ means the coordinate of *spot1*, (dq, dr, ds) means the cube coordinates of the direction *d*, and $(q2, r2, s2)$ means the coordinate of *spot2*. As shown in *Figure 1*, the coordinates of the black stone is (0, 0, 0), the white stone has coordinate (1, -1, 0).

Table 1 Cube coordinates for six directions.

Direction	Direction 0	Direction 1	Direction 2	Direction 3	Direction 4	Direction 5
Cube coordinates	(1, -1, 0)	(1, 0, -1)	(0, 1, -1)	(-1, 1, 0)	(-1, 0, 1)	(0, -1, 1)

$$(q2, r2, s2) = (q1, r1, s1) + (dq, dr, ds) \qquad \textit{Equation 1}$$

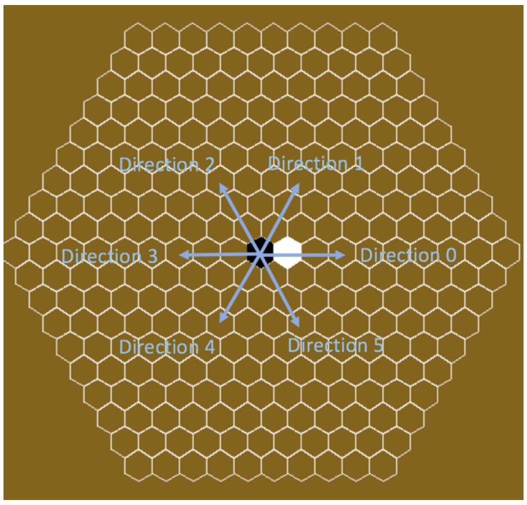

*Figure 1 A 10*10 hexagonal game board with a black stone in the center and an adjacent white stone. .*

2.2 Winning conditions

A player wins when reaching one of these conditions: forming a five-in-a-row line or fully enclosing one or more enemy stones inside 6 or more of his own stones. Two algorithms were designed for above conditions respectively.

2.2.1 Five-in-a-row

Five-in-a-row goal means the player forms an unbroken chain of five stones in one of the six directions. An algorithm named ***Check Win Five-in-a-row*** was built for judging this condition(see pseudocode in *Figure 2*). In inputs, steps_player means the coordinates of spots dropped by the player, coordinates_value contains the corresponding values of the spots.

```
Algorithm 1: Check Win Five-in-a-row

Inputs:  steps_player, coordinates_value
Output: is_player_win

Steps:   for step in steps_ player:
             For direction in directions:
                 connected_stone = 0
                 iteration = 0
                 while Iteration<5:
                     Compute next_spot with step and direction
                     if coordinates_value(next_spot) == player_value:
                         replace step with next_spot
                         iteration +=1
                         connected_stone +=1
                         if  connected_stone ==5:
                             is_player_win = True
                             break
                     else:
                         break
```

*Figure 2 The pseudocode of algorithm **Check Win Five-in-a-row** built for judging the five-in-a-row winning condition*

2.2.2 Fully-enclosed

Fully-enclosed goal means the player forms an unbroken circle that fully enclosed one or more enemy stones. Two algorithms named ***Check Win Enclose*** and ***Check Be-enclosed-flood-fill*** were built for judging this condition(see pseudocodes in *Figure 3* and *Figure 4*).

Algorithm 2 searches the neighbors of the last step from the player that has the stone from the opponent and pass the neighbors to Algorithm 3. Algorithm 3 checks whether there is a neighbor whose stone and its connected stones(dropped by the same player) are fully enclosed with stones from the player.

```
Algorithm 2: Check Win Enclose
Input: is_player1, steps_player1, steps_player2, coordinates_value
Output: is_player1_win, is_player2_win

Steps:  if is_player1 == True:
                last_step = steps_player1[-1]
        else：
                last_step = steps_player2[-1]
        for direction in directions:
                Compute next_spot with last_step and direction
                If coordinates_value(next_spot) == opponent_value
                        visited_spots = []
                        enclosed = Check Be-enclosed-flood-fill (next_spot, steps_player1,
                                        steps_player2, coordinates_value, visited_spots)
                        if enclosed==True and is_player1 == True:
                                is_player1_win = True
                        if enclosed==True and is_player1 == False:
                                is_player2_win = True
```

*Figure 3 The pseudocode of algorithm **Check Win Enclose** built for judging the fully-enclosed winning condition*

```
Algorithm 3: Check Be-enclosed-flood-fill
Input:   next_spot, steps_player1, steps_player2, coordinates_value, visited_spots
Output: enclosed

Steps:  Append next_spot to visited_spots
        next_sports = []
        for direction in directions:
                Compute next_spot with step and direction
                if next_spot not in visited_spots:
                        Append next_spot to next_spots
        If ANY (coordinates_value(next_spot)) == empty_value:
                break
        If ALL (coordinates_value(next_spot)) == player_value:
                enclosed = True
        If ANY(coordinates_value(next_spot)) == opponent_value:
                enclosed = Be-enclosed-flood-fill (next_spot, steps_player1,
                steps_player2, coordinates_value, visited_spot)
```

*Figure 4 The pseudocode of algorithm **Check Be-enclosed-flood-fill** built for judging the fully-enclosed winning condition*

2.3 Evaluation function

In this report, Minimax search and alpha-beta pruning algorithms were implemented to achieve intelligent search. For the Andantino, which has a deep search tree, searching until the terminal node is impractical. Therefore, an evaluation function is developed to give the heuristic value to nodes. (Wang, 2004) provides an overview of the evaluation function consisting ten threats in the game of Five-In-Row. Among all threats, four of them were involved in the evaluation function (*Equation 2*) of this game: five-in-a-row threat, four-in-a-row-threat, three-in-a-row threat and two-in-a-row threat. Two-in-a-row threat means the two connected chain that has the potential to be five-in-a-row threat. Five-in-a-row threat means the five connected chain.

$$Score = w1 * nr\_five\_threat + w2 * nr\_four\_threat + w3 * nr\_three\_threat + w4 * nr\_two\_threat$$

Equation 2

In above equation, $Score$ means the computed value to the node, $w1$ means the weight for threat 1, nr_five_threat means the number of the five-in-a-row threat. As the five-in-a-row threat leads to a winning condition, $w1$ equals infinity.

3. Result

3.1 Winning conditions

Andantino has two winning conditions: five-in-a-row and fully-enclosed. Successful implementing winning conditions means the program can detect the winning condition when it reached. After tests, the five-in-a-row winning condition works well, but the fully-enclosed only work when one stone is enclosed (see *Figure 5*).

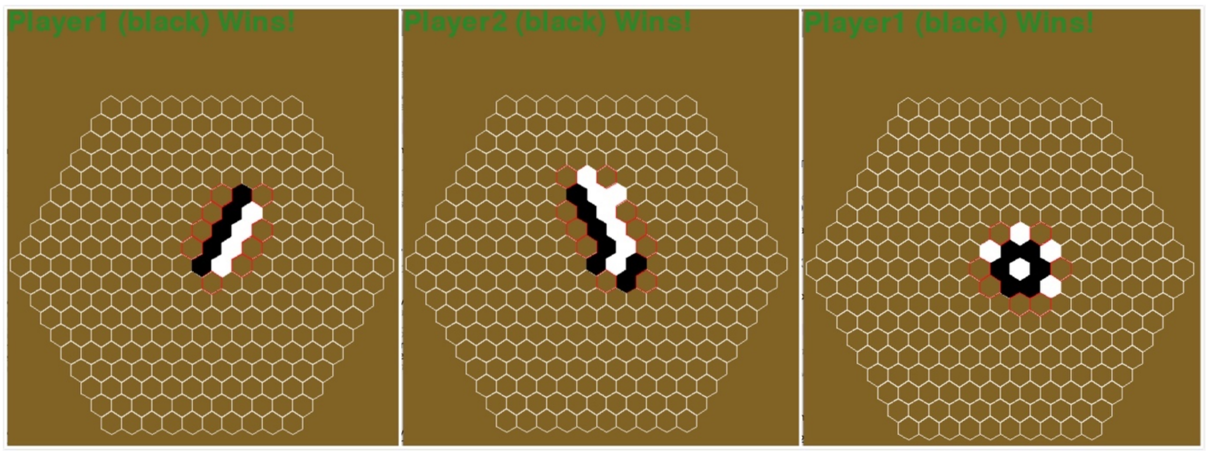

Figure 5 Examples of successful detections of winning conditions.

The program cannot detect the winning condition of fully-enclosed when more than one stone are fully enclosed. The error exists in the Algorithm 3 that checking the connected stones. When found a connected node, the Algorithm 3 get recalled, but the second IF function never get run. Therefore the program goes in infinite recursion.

3.2 Evaluation function

In this program, the player 1 (black) is set as potential winner, minimax searching and alpha-beta pruning were implemented to find the best moves. After testing, the algorithms don't work well in most case. As shown in *Figure 6*, left image shows that the algorithms don't work in simple condition. But the middle and the right images show that the algorithms works sometimes.

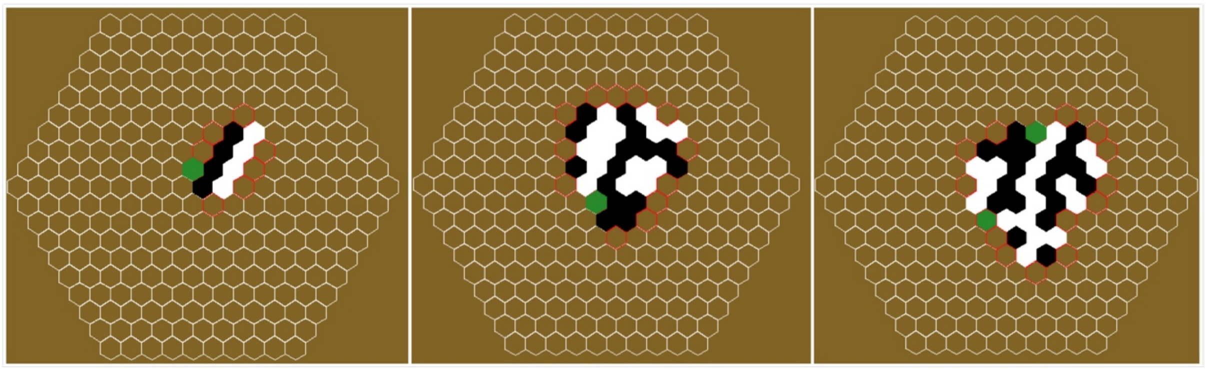

Figure 6 Examples of searching best moves with minimax searching and alpha-beta pruning. Spots with read outline means legal moves, and green spot means the searched best moves leading to the win of player 1.

After checking the code, all threats in the evaluation function are detected successfully. Possible reason for the bad performance is the weights. Current weights are infinity, 100, 10 and 1 for $w1, w2, w3$ and $w4$ respectively. Further test and calibrate should be done about the weights.

4. Conclusion

The game Andantino was programmed in Python 3.7. the five-in-a-row winning condition works well, but the fully-enclosed only work when one stone is enclosed. An evaluation function was built but doesn't work well in most case with minimax searching and alpha-beta pruning.

To successfully program the game, the bug in implementing the fully-enclosed condition need to be fixed. Current algorithm use a recursive method to find connected spots with same color, and the program goes in infinite recursion when more than one spot is enclosed. Possible solutions are re-writing the function with correct statements or changing the algorithm to an iteration method.

For applying intelligent search algorithms in Andantino, the evaluation function need updates by adding parameters that covers other threats in five-in-a-row winning condition and considers the fully-enclosed winning condition. Besides, further tests and calibrated are necessary.

5. Program Running Protocol

This is the protocol of running the program 'Andantino.py'.

1. Install pygame module.
2. Run file 'Andantino.py'.
3. Right click the center spot (drop the first black stone)
4. Right click on of the spots around the center spot. Then the spots with red outline (valid moves) and green spots (best move leading to the win of player 1 (black)) appears.

5. Right click the next step.
6. When a winning condition reached, ‘PlayerX (color) Wins!’ pops up on the left-top corner.
7. When to restart, close the window and re-run file ‘Andantino.py’.

6. References

Smith, D. (1995) *ANDANTINO*.

Wang, J.-H. C. and A. X. (2004) *Five-In-Row with Local Evaluation and Beam Search*.