Report for GOMOKU AI

Implementation of the game and AI by minimax with alpha-beta pruning

Junyan Qi

**Overview**

The files I’m responsible for are *gomoku.lisp*, *alpha-beta-for-gomoku.lisp*, *maker.lisp*, *compete.lisp* and they correspond to the following parts:

1). Implementation of Gomoku: provide game struct, evaluation functions, heuristics, apis such as who-wins?, game-over?, do-move!, undo-move!, default-policy etc.

2). Gomoku AI by minimax with alpha-beta pruning

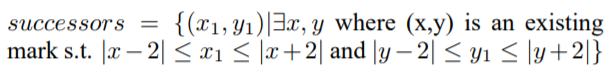
3). Compete with MCTS

**Approach**

**Part 1. Implementation of Gomoku**

The implementation of GOMOKU is pretty straightforward. It is simply placing tokens on board. And because it is more than complicated (require rather deep expert knowledge) to judge disallowed moves, I choose to not detect disallowed moves. Critical functions are listed in the head of the file *gomoku.lisp*.

One function I would like to note is legal-moves. Originally it is simply to scan through the board and return all positions that are \*blank\*. But this creates a problem of efficiency. To resolve this, I apply the following formula to reduce the searching space:



**Part 2. Gomoku AI by Minimax with Alpha-Beta Pruning**

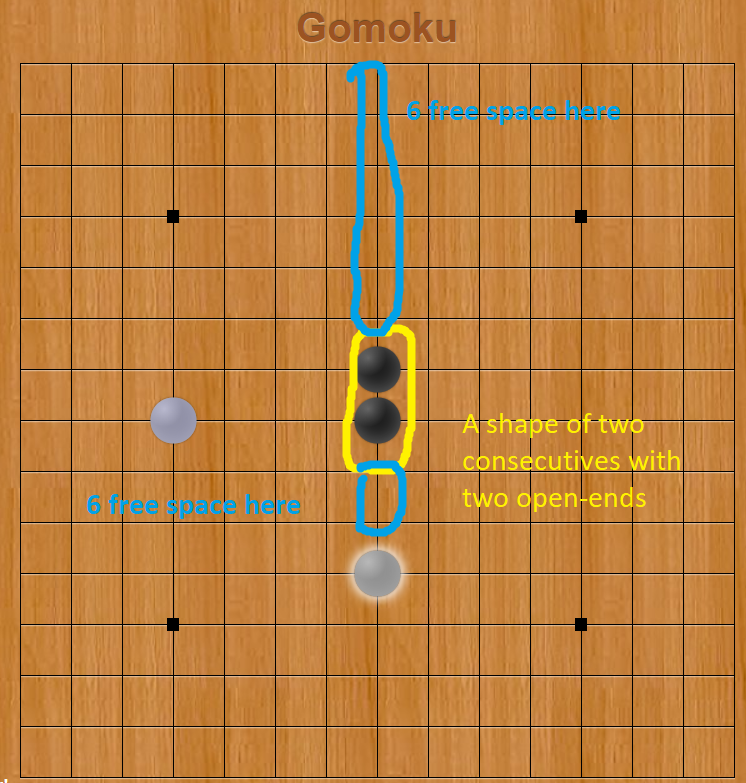
The main challenge of this part is to find a good evaluation function. My evaluation function is based on two simple intuitions:

1). How many consecutive pieces I already have?

2). How many more pieces I can add on to existing consecutive pieces

With these two intuitions in mind, the next challenge would be to recognize patterns on board. And I define it to be that there are 3 patterns: shape, singleton and break-pattern. A shape is consecutive pieces (more than 1); a singleton is an isolated piece; a break-pattern is several shapes that can be connected by one move.

For each pattern, I count the number of open-ends and the number of free space, for example:



**Shape Evaluation**

A picture containing wall

Description automatically generated Example of shape

The final eval-shape function looks like and the score will plus number of free space \* free space weight as weighing:

>= 5 consecutives – a win! (\*win-shape-score\*)

4 consecutives

2 open-ends

If it is my turn, a win! (\*win-shape-score\*)

Else, guarantee a win if there is no immediate threat! (1000)

1 open-ends

If it is my turn, a win! ((\*win-shape-score\*))

Else, an immediate threat! (300)

3 consecutives

2 open-ends

If it is my turn, guarantee a win if there is no immediate threat! (500)

Else, a potential big threat! (100)

1 open-ends

If it is my turn, not much threat but can develop into an immediate threat. (15)

Else, not a threat (10)

2 consecutives

2 open-ends

If it is my turn, a developing threat! (25)

Else, not a threat! (5)

1 open-ends

If it is my turn, not a threat! (0.5)

Else, not a threat! (0.1)

**Singleton Evaluation**

A picture containing indoor, cabinet, looking, front

Description automatically generated A picture containing sitting, indoor

Description automatically generated Examples of singleton

I use the same criterions as evaluating shape, the idea is the more the open-ends the better.

**Break-Pattern Evaluation**

A picture containing cabinet, indoor

Description automatically generatedExample of a break pattern

I evaluation such pattern by counting immediate threats a pattern will bring.

>= 5 consecutives – a win! (\*pos-inf\*)

= 4 consecutives and 2 open-ends – almost a win (10 threats)

= 4 consecutives and 1 open-end - an immediate threat (1)

= 3 consecutives and 2 open-end – an immediate threat (1)

\*pos-inf\* threats = a shape score of (\*win-value\* / 2), discourage such pattern

>= 2 threats = a shape score of 100

< 2 threats = a shape score of 0

**Further Improvements**

Order of searching is important to alpha-beta pruning and so I improve the legal-moves with this heuristic:

0. The first move is always the center.

1. If there are win moves (moves for current player to win immediately), only make them our candidates

2. Else If there are defensive moves (moves that current player has to take to prevent opponent from winning), only make them our candidates

3. Else sort legal moves based on eval-func

**Results and Tests**

**Part 1. Implementation of Gomoku**

There is not much to test for the implementation of gomoku due to its simplicity. Nonetheless, critical functions: legal-moves, who-wins?, game-over? are tested. (changes to do-move! and undo-move! which are taken from previous works are trivial)

**Part 2. Gomoku AI by Minimax with Alpha-Beta Pruning**

Study has shown that the black side (the player takes the first move) has huge advantages and will win even when both sides are experts of gomoku as well. Since disallowed moves, which are invented to restrict the player who takes the first moves, are not implemented, my goal is to show that if two minimax AIs play against each other (with same cutoff depth), the black side will win.

My testing results match the theory. Unless the board is too small (N < 9), the black side will typically win in 25 to 40 moves (or 12 to 20 exchanged hands) if both sides think 1 hand (two moves) ahead. The performance is that if AI is only thinking 1 hand (two moves) ahead, the computation is real-time. If 2 hands (4 moves) ahead, depending on the size of board can take from 3s to 20s.

Example test code:

(play-against-self 7 1 1)

(play-against-self 8 1 1)

(play-against-self 9 1 1)

(play-against-self 13 1 1)

(play-against-self 15 1 1)

**Part 3. Compete with MCTS-RAVE**

Also, we expect it (think 1 hand ahead) to beat MCTS-RAVE most of times no matter which side it takes, which, due to limitation of computational power, can’t have a reasonably large number of simulations. And the testing results match the expectation.

Example test code

(compete 8 T)

(compete 8 nil)

**Summary**

I spent most of my time on trying to come up with an accurate evaluation function and a good heuristic for gomoku. My thinking process is described in the Approach section. I think perhaps I can further improve the performance by improving the way I identify shapes and pattern. Currently I do that by walking through the board several times, which is then C \* O(N^2). Future improvement can be to reduce C, the number of times I scan the board.