Hexagonal Chess with Handcrafted AI Alorithms

Luu H.Son

Abstract - This report represents the statistic of AI algorithms used to play hexagonal chess game. This problem contains a lot of challenges, such as no available product/ open-source public on the internet, no suitable chess egine, different set of rules,.. Based on the complexity of search algorithms, we would implement basic algorithms such as minimax, environment, client app.

I. Introduction and Task Settings

THE most important part of an AI algorithms is how to make decision smart. And one of the key factors is the evaluation function. In this paper, we will implement 3 evaluation functions to combine with 2 search algorithms to compare their speed and performance. To do statistic and scaling up, python 3 is chosen.

Because of there are no predecessor of hexagonal chess, the environment of chess is a problem. We decide to build an environment of hexagonal chess on python 3.

1. Game introduction.

Hexagonal chess refers to a group of chess variants played on boards composed of hexagon cells. The best known is Gliński's variant, played on a symmetric 70-cell hexagonal board.

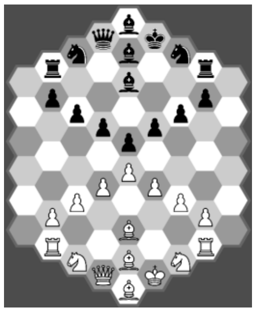
Since each hexagonal cell not on a board edge has six neighbor cells, there is increased mobility for pieces compared to a standard orthogonal chessboard. Three colours are typically used so that no two neighboring cells have the same colour, and a colour-restricted game piece such as the orthodox chess bishop usually comes in sets of three per player to maintain the game's balance.

Many different shapes and sizes of hexagon-based boards are used by variants. The nature of the game is also affected by the 30° orientation of the board's cells; the board can be horizontally (Wellisch's, de Vasa's, Brusky's) or vertically (Gliński's, Shafran's, McCooey's) oriented. (E.g., when the sides of hexagonal cells face the players, pawns typically have one straightforward move direction. If a variant's gameboard has cell vertices facing the players, pawns typically have two oblique-forward move directions.) The six-sidedness of the symmetric hexagon gameboard has also resulted in a number of three-player variants.

The first applications of chess on hexagonal boards probably occurred mid-19th century, but two early examples did not include a checkmate as the winning objective. More chess-like games for hexagon-based boards started appearing regularly at the beginning of the 20th century. Hexagon-celled gameboards have grown in use for strategy games generally; for example, they are popularly used in modern wargaming.

In this report, we using a minimalist version of Glinski chess which has a symmetric 70-cell hexagonal chessboard. In terms of board pieces, we using 7 pawns, 2 knights, rooks, 3 bishops, 1 queen and 1 king for each team.

Initial chess board:

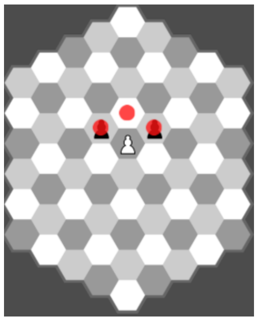


Different rules:

No castling moves
No promotion for pawns

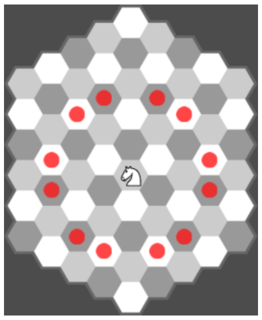
Move of chess pieces:

PAWN:

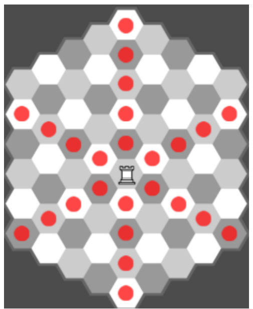


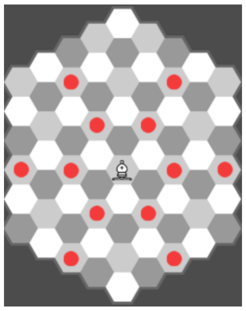
Reversed for black pawn

KNIGHT:



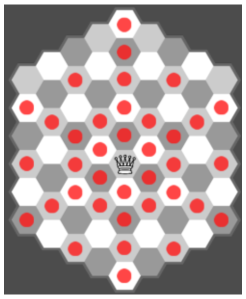
BISHOP:
ROOK:

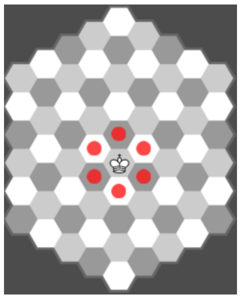




QUEEN:

KING:





2.Environment and Client UI/UX

To start with the environment, we created classes that refer to kinds of chess, such as Pawn, Bishop, Knight... Each class contains a valid position on chessboard, team, value, and it can generate moves itself. Moreover, a chessboard contains valid position of chess pieces. Through chessboard, we can generate all possible legal moves of each team. To support look-ahead chessboard, we allow creating new chessboard by list of legal positions of each chess pieces.

By using chessboard environment, users can generate next legal moves, create new chessboard by using list of pieces' positions.

Godot Engine is an appropriate tool for creating the client app, which allows users to play with our AI. We used basic chess sprites which are available on the internet, including chessboard sprite. Client has to follow basic logic of a chess game. Because of splitting, we must have a communication between client and our algorithms. Flask is an appropriate choice because it is based on REST paradigm.

3. Algorithms

We decide to implement the algorithms in python using our environment. The reasons are it is easier to debug, test, scale.. than implementing algorithms on client.

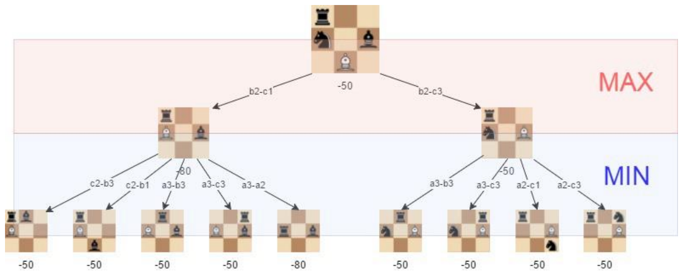
MINIMAX

Chess is a zero-sum game, so the minimax algorithm is a good choice to make a decision. This algorithm bases on the max and min value of the state board. Each chessboard will have a value that is generated by the heuristic evaluation function. If a value is greater than 0, the advantage belongs to White. If a value is lower than 0, the advantage belongs to Black team. If you are playing as Black team, you need to make a move generating advantage - the lower value the higher advantage.

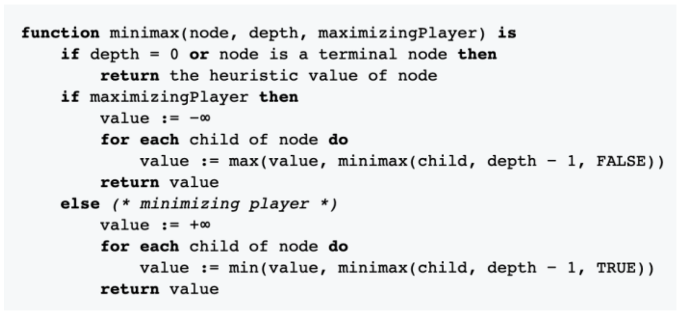
Algorithms description:

The minimax function returns a heuristic value for leaf nodes - also know as chessboard (terminal nodes and nodes at the maximum search depth). Non-leaf nodes inherit their value from a descendant leaf node. The heuristic value is a score measuring the favorability of the node - chess board - for the maximizing player. Hence nodes resulting in a favorable outcome, for the maximizing player have higher scores than nodes more favorable for the minimizing player. For non-terminal leaf nodes at the maximum search depth, an evaluation function estimates a heuristic value for the chessboard. The quality of this estimate and the search depth determine the quality and accuracy of the final minimax result. So evaluation function is one of the most important parts of this algorithms. The better the evaluation function is, the more accurate decision is made.

Example:



Pseudo code:



```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

maximizing player : WHITE
minimizing player : BLACK

Initial invoke for BLACK - AI player:

minimax(board,3,false)

NEGAMAX

Algorithms desciption:

Negamax algorithm is a variant of minimax which bases on the fact that:

$$max(a, b) = -min(-a, -b)$$

to simplify the implementation of the minimax algorithm. More precisely, the value of a position to maximizing player (WHITE) in a game is the negation of the value to minimizing player. Thus, the player on move looks for a move that maximizes the negation of the value resulting from the move: this successor position must by definition have been valued by the opponent. The reasoning of the previous sentence works regardless of whether WHITE or BLACK is on move. This means that a single procedure can be used to value both positions. This is a coding simplification over minimax, which requires that maximizing selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor.

Pseudo code:

```
function negamax(node, depth, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    value := −∞
    for each child of node do
        value := max(value, −negamax(child, depth − 1, −color))
    return value
```

Initial invoke for BLACK - AI player:

negamax(board,3,-1)

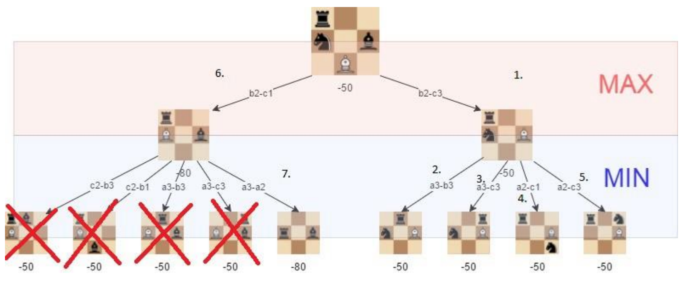
MINIMAX WITH ALPHA-BETA PRUNING

Algorithms description:

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player (beta aka BLACK) is assured of becomes less than the minimum score that the maximizing player (alpha aka WHITE) is assured of (beta <= alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

Example:



by using alpha-beta pruning, we can reduce alot of computations.

Pseudo code:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if β ≤ α then
                break (* α cut-off *)
        return value
```

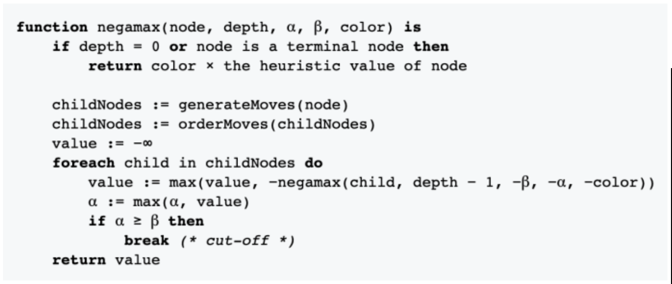
Initial invoke for BLACK - AI player:

minimaxAlphaBeta(board,3,10000000000,1000000000, BLACK)

NEGAMAX WITH ALPHA-BETA PRUNING

Negamax is an more simplified version of minimax. We can using alpha-beta pruning for negamax algorithm.

Pseudo code:



```
function negamax(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node

    childNodes := generateMoves(node)
    childNodes := orderMoves(childNodes)
    value := −∞
    foreach child in childNodes do
        value := max(value, −negamax(child, depth − 1, −β, −α, −color))
        α := max(α, value)
        if α ≥ β then
            break (* cut-off *)
    return value
```

Initial invoke for BLACK - AI player:

negamaxAlphaBeta(board,3,-10000000000,1000000000,-1)

4. Evaluation Function

The important part of minimax, negamax algorithms is evaluation function - which is also the heart of AI. As long as AI players can appropriate evaluation state, it still can make good decisions. But how to evaluate an evaluation function is good enough? We use 2 different evaluation function.

Material Evaluation Function

Consider each piece of pieces on board have different value as following table:

	WHITE	BLACK
PAWN	10	-10
KNIGHT	30	-30
BISHOP	60	-60
ROOK	100	-100
QUEEN	250	-250
KING	900	-900

We compute sum of all pieces' value on the board:

$$\text{Evaluation} = \sum x + \sum y$$

x: piece's value of WHITE
y: piece's value of BLACK

It's is a naive material position because of comparison of number of piece for each team whatever it's position. It's not good as when a piece stands at appropriate position, it may have more impact on adversary. For example, a knight has more impact when it is at the center position of board, not angle. Thus, we improve material position evaluation function by adding coefficient of position of each board.

Material Evaluation with Coefficient of Position

White pawn's coefficient position table:

	A	B	C	D	E	F	G	H	I
1	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	1
3	0.5	0	0	0	0	0	0	0	0.5
4	0.75	0.5	0	0	0	0	0	0.5	0.75
5	-1	0.75	0.5	0	0	0	0.5	0.75	-1
6	-2	-1	0.75	0.5	0	0.5	0.75	-1	-2
7	x	-2	-1	0.75	0.5	0.75	-1	-2	x
8	x	x	-2	-1	0.75	-1	-2	x	x
9	x	x	x	-2	-1	-2	x	x	x
10	x	x	x	x	-2	x	x	x	x

Black pawn's coefficient position table:

	A	B	C	D	E	F	G	H	I
1	-2	-2	-2	-2	-2	-2	-2	-2	-2
2	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75
4	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
5	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	0	0
7	x	0	0	0	0	0	0	0	x
8	x	x	0	0	0	0	0	x	x
9	x	x	x	0	0	0	x	x	x
10	x	x	x	x	0	x	x	x	x

Knight's coefficient position table:

	A	B	C	D	E	F	G	H	I
1	-5	-4	-3	-2	-1	-2	-3	-4	-5
2	-4	-3	0	0	0.5	0	0	-3	-4
3	-3	-2	1	1	1.5	1	1	-2	-3
4	-3	-1	1.5	1.5	2	1.5	1.5	-1	-3
5	-4	-2	1.5	2	2	2	1.5	-2	-4
6	-5	-3	1	1.5	2	1.5	1	-3	-5
7	x	-4	0	1	2	1	0	-4	x
8	x	x	-3	0	1.5	0	-3	x	x
9	x	x	x	-2	0.5	-2	x	x	x
10	x	x	x	x	-1	x	x	x	x

Bishop's coefficient position table:

	A	B	C	D	E	F	G	H	I
1	-2	-1	-0.5	-1	0	-1	-0.5	-1	-2
2	-1	0.5	0	-0.5	0	-0.5	0	0.5	-1
3	-0.5	1	1	1	0	1	1	1	-0.5
4	-0.5	0	0	2	0	2	0	0	-0.5
5	-1	1	2	2	0	2	2	1	-1
6	-2	0.5	1	2	0	2	1	0.5	-2
7	x	-1	0	1	0	1	0	-1	x
8	x	x	-0.5	-0.5	0	-0.5	-0.5	x	x
9	x	x	x	-1	0	-1	x	x	x
10	x	x	x	x	0	x	x	x	x

Rook's coefficient position table:

	A	B	C	D	E	F	G	H	I
1	-1	0.5	0.5	0.5	-1	0.5	0.5	0.5	-1
2	1	1	1	1	1.5	1	1	1	1
3	1	1	1.5	1.5	2	1.5	1.5	1	1
4	1	1	1.5	2	2	2	1.5	1	1
5	1	1	1.5	2	2	2	1.5	1	1
6	-1	1	1.5	2	2	2	1.5	1	-1
7	x	0.5	1	1.5	2	1.5	1	0.5	x
8	x	x	0.5	1	2	1	0.5	x	x
9	x	x	x	0.5	1.5	0.5	x	x	x
10	x	x	x	x	-1	x	x	x	x

Queen's coefficient position table:

	A	B	C	D	E	F	G	H	I
1	-2	-1	-0.5	-1	-2	-1	-0.5	-1	-2
2	-1	0	1	0	-1	0	1	0	-1
3	1	0	1.5	0.5	0	0.5	1.5	0	1
4	1	0.5	1.5	0.5	0.5	0.5	1.5	0.5	1
5	-1	0	1.5	0.5	1	0.5	1.5	0	-1
6	-2	0	1.5	0.5	1	0.5	1.5	0	-2
7	x	-1	1	0.5	1	0.5	1	-1	x
8	x	x	0.5	0	0.5	0	0.5	x	x
9	x	x	x	-1	-1	-1	x	x	x
10	x	x	x	x	-2	x	x	x	x

King's coefficient position table:

	A	B	C	D	E	F	G	H	I
1	-2	-1	0.5	-1	-2	-1	0.5	-1	-2
2	-1	0.5	1	1	1	1	1	0.5	-1
3	1	1	1	1	1	1	1	1	1
4	1	1	1	2	2	2	1	1	1
5	-1	1	1	2	2	2	1	1	-1
6	-2	0.5	1	2	2	2	1	0.5	-2
7	x	-1	1	1	2	1	1	-1	x
8	x	x	0.5	1	1	1	0.5	x	x
9	x	x	x	-1	1	-1	x	x	x
10	x	x	x	x	-2	x	x	x	x

x: position is not valid so it doesn't have coefficient

We compute sum of all pieces' value on the board:

Evaluation = $\sum x * c_x + \sum y * c_y$

x: piece's value of WHITE
y: piece's value of BLACK
c_x: position coefficient of x - WHITE
c_y: position coefficient of y - BLACK

II. Statistic

When No next state is 42 using naive material evaluation function:

	depth = 2	depth=3
Minimax	0.7836	33.8448
Minimax with alpha-beta pruning	0.1699	1.9678
Negamax	0.7503	33.1839
Negamax with alpha-beta pruning	0.6329	13.2482

When No next state is 42 using coefficient position material evaluation function :

	depth = 2	depth=3
Minimax	0.7779	33.9653
Minimax with alpha-beta pruning	0.1703	1.9557
Negamax	0.7479	33.0685
Negamax with alpha-beta pruning	0.6203	12.6925

When No next state is 76 using material evaluation function :

	depth = 2	depth=3
Minimax	2.2416	174.7319
Minimax with alpha-beta pruning	1.2120	33.7114
Negamax	2.1064	162.2979
Negamax with alpha-beta pruning	1.8852	54.7841

When No next state is 76 using coefficient position material evaluation function:

	depth = 2	depth=3
Minimax	2.2353	174.3262
Minimax with alpha-beta pruning	1.1929	34.9731
Negamax	2.2674	177.1767
Negamax with alpha-beta pruning	2.0694	61.5858

Assumption that each state have average 50 next state (50 legal moves of AI player):

	depth = 2	depth=3
Minimax	~1.5s ± 0.75s	~100 ± 70s
Minimax with alpha-beta pruning	~0.75s ± 0.5s	~30s ± 25s
Negamax	~2s ± 0.5s	~100 ± 70s
Negamax with alpha-beta pruning	~1.75s ± 0.25s	~30s ± 25s

More than that, we implemented that bot can play with themselves.

III. Conclusion

Glinski's chess is a variant of chess game, which has different rules, boards. This report is an overview of implement algorithms as AI players and statistics. Through statistics, we can evaluate that negamax with alpha-beta pruning has the highest performance and easy to implement. In addition, using alpha-beta reduce a lot of computations. To improve AI policy, we need more computational resources or implement other algorithms.

Acknowledgment

We special thanks to Le Quoc Minh, who corrects my report's grammars and vocabulary.

References

En.wikipedia.org. 2020. *Alpha–Beta Pruning*. [online] Available at: <https://en.wikipedia.org/wiki/Alpha–beta_pruning> [Accessed 17 July 2020].

En.wikipedia.org. 2020. *Hexagonal Chess*. [online] Available at: <https://en.wikipedia.org/wiki/Hexagonal_chess> [Accessed 17 July 2020].

En.wikipedia.org. 2020. *Minimax*. [online] Available at: <https://en.wikipedia.org/wiki/Minimax> [Accessed 17 July 2020].

En.wikipedia.org. 2020. *Negamax*. [online] Available at: <https://en.wikipedia.org/wiki/Negamax> [Accessed 17 July 2020].