Xingwen Wei
Thuc Nhi Le
CS 420
Professor Tao
October 28, 2018

PROJECT 4 REPORT

1 Pseudo-code for MINIMAX and ALPHA-BETA PRUNING algorithms

We introduced several Class variables:

evaluation_ct ← The number of times a static evaluation was done.
node_ct ← The number of nodes that we find the legal actions for.
branching_ct ← The number of branches for each state, the average branching factor is calculated by dividing branching_ct by node_ct
cutoff_ct ← The number of cut offs that took place.

Since there are similarities in the algorithms for MINIMAX and ALPHA-BETA PRUNING, we combined two algorithms into one. We used *alphabeta* boolean to notify whether we were running alpha-beta pruning or not.

```
function MINIMAX(state, level, black, alphabeta) returns (status of the game, action)
    inputs: state, a board representing the current state
            level, depth of search
            black, a boolean representing whether this is the black move
            alphabeta, a boolean whether this is alpha-beta pruning algorithm

    if this is black move then return maxVal(state,-infinity,+infinity,level,alphabeta)
    else return minVal(state,-infinity,+infinity,level,alphabeta)

function maxVal(state, alpha, beta, level, alphabeta) returns (evaluation score, action)
    inputs: state, a board representing the current state
            alpha, the value of the best choice found so far at any choice point along the
path for Black move, initially -infinity, only used for alpha-beta pruning algorithm
            beta, the value of the best choice found so far at any choice point along the path
for White move, initially +infinity, only used for alpha-beta pruning algorithm
            alphabeta, a boolean whether this is alpha-beta pruning algorithm

    eval ← evaluation(state, True)
    evaluation_ct ++
    if game is over or level is 0 then
        evaluation_ct --
        return (evaluation score, NULL)
    node_ct++
    v← -infinity
    current_best_move ← NULL
    legal_actions ← the list of legal actions for current state for the current player
    branching_ct ← branching_ct + len(legal_actions)
    **for each** action **in** legal_actions **do**
        child ← the board resulted from the action
        **if** child is not NULL
                v ← MAX(v, evaluation score of minVal(child, alpha, beta, level-1,
    alphabeta))
                **if** not alphabeta **then**
                        current_best_move←action with the highest evaluation score
                **if** alphabeta **then**
                        **if** v >= beta **then**
                                cutoff_ct++
                                **return** (v, action)
                        v ← MAX(v, alpha)
                        current_best_move←action
    **return** (v, current_best_move)

**function** minVal(state, alpha, beta, level, alphabeta) **returns** (evaluation score, action)
    **inputs**: *state*, a board representing the current state
            *alpha*, the value of the best choice found so far at any choice point along the
path for Black move, initially -infinity, only used for alpha-beta pruning algorithm
            *beta*, the value of the best choice found so far at any choice point along the path
for White move, initially +infinity, only used for alpha-beta pruning algorithm
            *alphabeta*, a boolean whether this is alpha-beta pruning algorithm

    eval ← evaluation(state, False)
    evaluation_ct ++
    **if** game is over **or** level is 0 **then**
        evaluation_ct --
        **return** (evaluation score, NULL)
    node_ct++
    v← +infinity
    current_best_move ← NULL
    legal_actions ← the list of legal actions for current state for the current player
    branching_ct ← branching_ct + len(legal_actions)
    **for each** action **in** legal_actions **do**
        child ← the board resulted from the action
        **if** child is not NULL
                v ← MIN(v, evaluation score of maxVal(child, alpha, beta, level-1,
    alphabeta))
```

```
                    if not alphabeta then
                        current_best_move←action with the lowest evaluation score
                    if alphabeta then
                        if v <= alpha then
                            cutoff_ct++
                            return (v, action)
                        v ← MIN(v, alpha)
                        current_best_move←action
    return (v, current_best_move)
```

2 Static Evaluation Function

2.1 Pseudo-code

```
function EVALUATION(state, black) returns (status of the game, evaluation score)
    inputs: state, a board representing the current state
            black, a boolean representing whether this is the black move
    black_actions ← list of potential legal actions for black player that can be performed sequentially
    white_actions ← list of potential legal actions for white player that can be performed sequentially
    if not black and white_actions is 0 then return (True, black_actions - white_actions)
    if black and black_actions is 0 then return (True, black_actions - white_actions)
    return (False, black_actions - white_actions)
```

2.2 Explanation

Our goal is to maximize our potential legal moves and minimize our opponent's legal moves. Therefore, we took the difference between black legal moves and white legal moves. Black player will try to maximize this difference while White player will try to minimize this difference. This is the reason why we associated black player with maxVal function and white player with minVal function.

3 Data comparison

MiniMax			
Level	Static Evaluation Count	Average Branching Factor	Cut offs Count
1	365	6.47	0

2	4106	9.41	0
3	27421	8.08	0
4	206407	8.73	0
5	1402324	8.44	0
6	N/A	N/A	N/A

Alpha Beta Pruning			
Level	Static Evaluation Count	Average Branching Factor	Cut offs Count
1	365	6.47	0
2	1845	9.41	297
3	6166	8.08	810
4	24934	8.28	5278
5	58782	7.56	10186
6	464525	9.33	98734

It is obvious that except for level 1, the alpha-beta pruning had to explore much fewer nodes to make a logical decision than minimax and we could not get the result for level 6 for minimax. Both algorithms grow exponentially in respect to the number of levels but alpha-beta pruning can reduce the number of nodes needed to be explored with the polynomial factor. Although alpha-beta still grows exponentially, pruning can help with relatively low level.

4 Optimization Choice

When we expand a node, we explore the children in a best-to-worst order. That is we explore the node that is potentially the best choice first according to our evaluation function so that there is a higher chance of pruning than just explore the children in a random order. Based on the data we collected from actual game play (provided below), we explore about only half of the nodes after we change the order of visiting children from random to sorted.

We implemented forward pruning in our algorithm. In theory, forward pruning is supposed to prune out bad nodes without looking into their children. This is good because it saves us time of looking into the bad nodes children and their children's children so that we can use that saved time on going deeper with good nodes. However it is inherently risky since it is very possible that we prune out a node being deemed as bad by our estimation function but turns out to produce good children better than any other nodes. Moreover, it is not specified how many bad nodes to prune out. Considering the factors mentioned above, we choose to put the children of a node in order of their estimation score and prune out the worst 25%. This decision is made because the estimation score of children of a node is rather close based on our experiment and this yields the best result. We try to find a balance point between saving more time by pruning out more bad children and being more careful not to prune out potential good children.

Given the same amount of time, the alpha-beta pruning algorithm implemented with forward pruning can beat the standard alpha-beta pruning algorithm with various levels.

We chose the depth of level 4 for the reasonable amount of time to process the steps and still give a good enough result.

Data

Alphabeta
Without ordering

Level	Evaluation Count	Branching Factor	Cutoff Count	Result	Length of game
1	378	6.89	0	White	0.081617s
2	1797	10.04	306	White	0.390444s
3	7333	8.42	898	White	1.546320s
4	21442	9.37	4709	White	4.771501s
5	135943	9.42	23614	White	30.706351s
6	280953	9.96	64259	Black	63.329302s

With ordering

Level	Evaluation Count	Branching Factor	Cutoff Count	Result	Length of game
1	442	7.36	0	White	0.158220s
2	1164	8.84	297	White	0.963077s
3	4074	8.51	643	White	2.866236s
4	10089	9.06	3028	White	9.610137s
5	24526	7.26	5314	White	17.388248s
6	147856	11.25	44017	White	167.538933s

	1	2	3	4	5	6
Time for 4th-8th steps of Ordered alpha beta pruning (Trying to Optimize version)	Black: 00.002796 White: 00.002522 Black: 00.002272 White: 00.002550	Black: 00.011687 White: 00.020842 Black: 00.022826 White: 00.021047	Black: 00.056042 White: 00.036824 Black: 00.039992 White: 00.030933	Black: 00.268434 White: 00.222071 Black: 00.184400 White: 00.280974	Black: 00.668490 White: 00.537166 Black: 00.895720 White: 00.976599	Black: 01.900566 White: 02.607293 Black: 04.447977 White: 06.979998

Time for 4th-8th steps of non-order alpha beta pruning (Standard)	Black: 00.000566 White: 00.001899 Black: 00.000581 White: 00.001984	Black: 00.006288 White: 00.009475 Black: 00.009900 White: 00.010765	Black: 00.024222 White: 00.019591 Black: 00.032931 White: 00.036434	Black: 00.088776 White: 00.131091 Black: 00.213377 White: 00.168128	Black: 00.141288 White: 00.294514 Black: 00.255834 White: 00.333723	Black: 02.339191 White: 02.429965 Black: 02.738355 White: 04.621385

Minimax

Level	Evaluation Count	Branching Factor	Cutoff Count	Result	Length of game
1	388	6.63	0	Black	0.087319s
2	3532	8.85	0	Black	0.752847s
3	27421	8.71	0	White	6.039057s
4	206407	8.73	0	Black	46.37055s
5	3234335	10.57	0	White	767.097409s
6	N/A	N/A	N/A	N/A	