Imperial College London

COURSEWORK 2

ADVERSARIAL SEARCH ON *(m,n,k)* GAMES

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

# Introduction to Symbolic AI

*Authors:*

1. Alex Gaskell (*01813313*, aeg19@imperial.ac.uk)
2. Harry Coppock (*01800452*, Hgc19@ic.ac.uk)

Date: January 6, 2020

# 1 ReadMe

To play a (m,n,k) game against our minimax/alpha beta algorithm simply run **main.py** in any Python3 environment. Set the m,n,k values in line 499 and set ab_flag = True for Alpha-Beta and False for minimax in line 500.

# 2 Minimax Implementation

Minimax was implemented in the class **Game()** in the recursive method **minimax()** (see lines 257-316). The general methodology was to first check whether the game was over or not using the functions **has_won()** (see lines 353-412) and **draw()** (see lines 414-416). If the game had terminated then the node was set to a leaf, returning the outcome of the game to be passed up the tree. Otherwise, a new node was created for every possible move, recursively calling the **minimax()** function for each possible move.

To record the predicted score for each move a dictionary, move_scores, was created. After iterating through all the possible moves and recursively searching down them, the optimal move was calculated by passing the move_scores dictionary to the method **choose_best_move()** see lines 334-347. To improve performance, the **has_won()** method only checked whether taking the last move resulted in a win, rather than scanning through the whole board.

# 3 Alpha Beta Implementation

Our implementation of Alpha-Beta pruning closely follows that described in the lecture notes and can be seen in method **minimax_ab()** (see lines 193-254). This is identical to our minimax method but with the addition of lines 232-235 and lines 248-251. Lines 232-235 say that is the value of a state is greater than the maximum value imposed by Min (to Max) (i.e. $evaluation > \beta$) then it can be pruned as Min will never choose values yielding greater than $\beta$ for Max. Likewise, lines 232-235 say that a branch can be pruned if it yeild less reward for Max than the previous minimum reward for Max as Max would never choose this path.

# 4 Analysis

An interesting observation is that as there is no penalisation for the number of moves taken to achieve a win/draw. Once the computer has guaranteed a win/draw it will not necessarily take the move which wins/draws in the smallest number of steps. This is due to the absence of any discounting of the returns in the implementation. This also has the affect that once the computer has predicted a guaranteed loss, based on assuming the player takes the optimal actions, it's chosen move will not be the one which prolongs defeat for the longest period of time, allowing the player to easily win. This is a flaw in the algorithm as the player could make a non optimal action by mistake, an opportunity that currently the computer would not always take advantage of.

# 5 Results

This section addresses questions 3 and 4 from the specification. When using the standard tic-tac-toe configuration (m,n,k = $3, 3, 3$), our implementation took $13.6s$ without Alpha-Beta pruning, and $0.5s$ with. This clearly illustrates the performance benefit of using Alpha-Beta pruning.

Our results below elaborate upon the performance benefits of using Alpha-Beta pruning as illustrated in Figure 1 and Table 1. Specifically, Figure 1 plots the comparison of computation time under different scenarios of m,n (the x-axis is $m*n$ capturing all of these combinations), with k = 2 in all of these cases; this gave more data points without our implementation time becoming impractical. Additionally, the time shown is the time our implementation took to compute its first move only, with the player beginning the game by playing in cell A1. This is because the time to compute the first move accounted for at least 90% of the total computation time of a whole game, so the computation time of subsequent moves is insignificant. The y-axis is log of computation time (seconds).

Our results, particularly Figure 1, clearly illustrate two features:

1. There is a clear log-linear relationship between the number of states (m * n) and computation time. This is expected as the time complexity of minimax is $O(b^m)$ (b = branching factor, m = maximum length of any path in the state space), therefore the log of computation time should be linear.

2. The gradient of the Alpha-Beta version of the algorithm is considerably shallower than the gradient of the minimax version. This is to be expected as the best case time complexity for Alpha Beta pruning is $O(b^{\frac{m}{2}})$ and $O(b^m)$ in the worst case, so the gradient should be shallower for the Alpha-Beta version.

We could not verify the results from question 5 because our implementation took too long to compute moves when the board was $5x6$ or larger, even with Alpha-Beta pruning.

| M | N | K | Time (s) (with Alpha-Beta pruning) | Time (s) (without Alpha-Beta pruning) |
|---|---|---|---|---|
| 2 | 2 | 2 | 0.0003 | 0.0002 |
| 2 | 3 | 2 | 0.0004 | 0.0010 |
| 3 | 3 | 2 | 0.0007 | 0.0173 |
| 4 | 2 | 2 | 0.0004 | 0.0054 |
| 5 | 2 | 2 | 0.0006 | 0.0386 |
| 6 | 2 | 2 | 0.0009 | 0.3055 |
| 7 | 2 | 2 | 0.0011 | 2.5751 |
| 4 | 3 | 2 | 0.0011 | 0.3892 |
| 4 | 4 | 2 | 0.0022 | 23.7187 |
| 5 | 3 | 2 | 0.0014 | 12.1499 |
| 8 | 2 | 2 | 0.0011 | 26.6721 |

**Table 1:** Table showing the computation times as plotted in Figure 1.

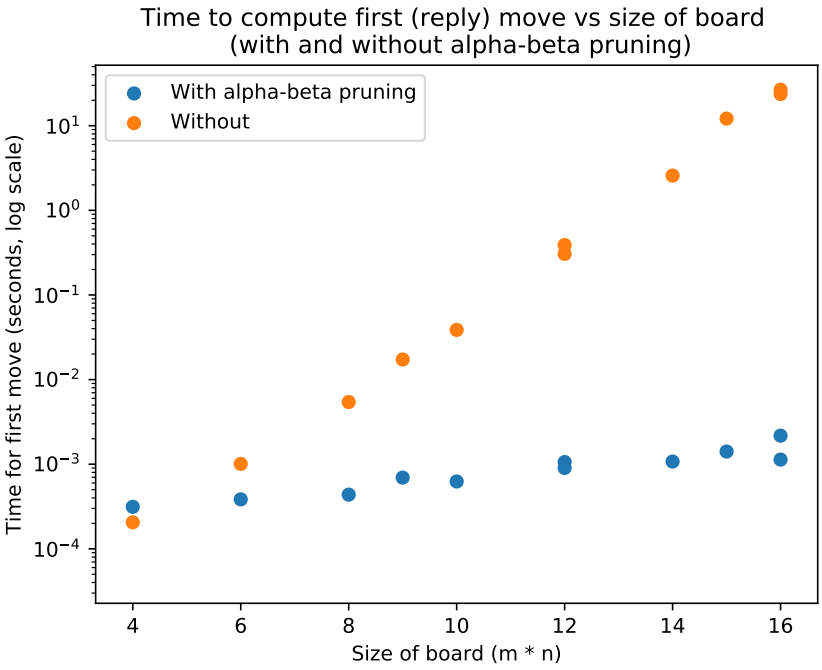


**Figure 1:** Comparison of computation time for our minimiax implementation under different combinations of $m$ and $n$ (with $k = 2$); with and without Alpha-Beta pruning.