An Analysis of Othello AI Strategies

CS7IS2 Project (2019/2020)

Alec Barber, Warren Pretorius, Uzair Qureshi, Dhruv Kumar

barberal@tcd.ie, pretoriw@tcd.ie, mqureshi@tcd.ie, kumardh@tcd.ie

Abstract. Othello is an interesting game in the domain of artificial intelligence due to a somewhat unexpected level of complexity. Simplified versions of the game on a 6x6 and 4x4 versions of the board have been solved, but no full solution has been found for the classic 8x8 board and above. This letter surveys a number of state-of-the-art approaches to Othello game playing including CNNs and adversarial models. Performance is measured against computational efficacy, model memory size and Monte Carlo simulation against a benchmark greedy algorithm. A further investigation into heuristics is also presented. Results indicate that the adversarial models outperform the CNN but is highly dependant on the heuristics adopted. A further analysis on Bayesian and reinforcements methods is provided.

Keywords: Othello, Reversi, Mini-max, alpha-beta pruning, Scout, Artificial Neural Network, CNN

1 Introduction

Othello is a modern adaption of an older game of Reversi[1]. In the domain of artificial intelligence it is considered an interesting game due to its perceived unsuitability for Neural Networks. It was previously assumed to be a somewhat trivial game to solve via analytical methods. In 2002, an influential survey by Van Den Herik predicted that the game of Othello would be solved by the year 2010 [1]. While smaller versions of the game board have been solved, no perfect solution exists for the classic 8×8 board.

Game playing is a subset of AI where an agent is tasked with maximising its score in a specified, dynamic game environment against a human or other game playing opponent. Game playing research against human opponents is separable into two distinct research regimes, the study of human though processes and inferences, and the transformation of these processes to be represented in a digital setting [2].

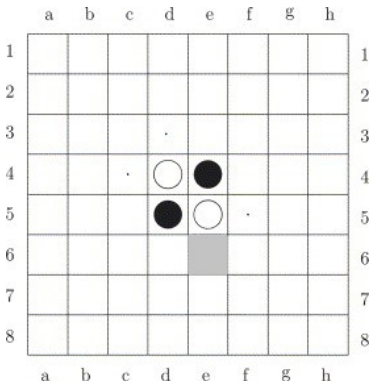
As Othello is considered a poor candidate for Neural Networks, it is an interesting environment to utilise and test more classical approaches to game playing in AI. Finding advancements in effective game playing strategies in Othello could potentially lead to the discovery of an increased performance general game playing schema. In this letter, the implementation, testing and analysis of various adversarial search methods alongside CNNs is undertaken [3],[4] [5] [6].

The remainder of this paper is outlined as follows. Section 2 discusses the Othello board game and other preliminaries. Section 3 and Section 4 discuss a number of different game playing strategies. Section 5 discusses the evaluation setting for the algorithms, and the results produced. Section 6 examines other algorithms which were not tested in this paper. Section 7 concludes the paper.

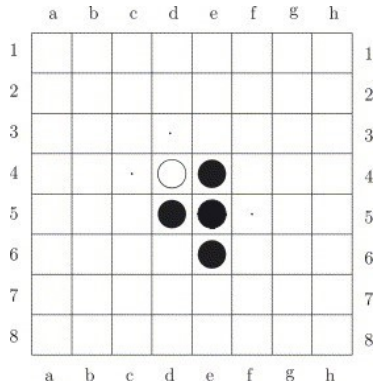
[1] https://www.mastersofgames.com/rules/reversi-othello-rules.htm

2 Othello Game

Othello is a two-player, deterministic, zero-sum (i.e. the total reward is fixed and the player's score is negatively related) board game which has perfect information[1]. The game is played on an 8x8 board using 64 dual colored discs. Each disc is colored black on one side and white on the other. Initially, the entire board is empty except the central 4 squares. In the main diagonal, white discs are kept on d4 and e5, and on the other diagonal black discs are kept i.e. on e4 and d5. This initial board configuration is shown in Figure 1a.



(a) Initial game board with highlighted tentative move.



(b) Game state after tentative move completed.

Fig. 1: Example first game move.

The game begins with player black (henceforth referred to as just black or white) making the first move. A legal move is made by placing a disc on an empty square so that in at least one direction from the square played on, there is a sequence of one or more of the opponents discs followed by the player's own disc [1]. The opponent's discs, in such a sequence, are then flipped and become the current player's color. For example, if the player moves e6(as highlighted in Figure 1a), the white disc on e5 will be turned over to black (as highlighted in Figure 1b).

If a player cannot make a legal move, they must forfeit their turn. The game ends when neither player can make a legal move, i.e. either when all 60 squares are filled or, if squares are left, when neither player can legally place their disc in one. The player with most discs on the board wins, or if the number of the discs are equal, then the game is deemed as a draw. The proceeding sections will outline adversarial methods and CNN approaches to Othello game playing.

3 Adversarial Search Othello

One of the most rudimentary and effective Artificial Intelligence algorithms are based on adversarial searches. Adversarial search is a search in which we plan ahead of the environment and other agents that are planning against us. Since in this environment (game) more than one agent searching for a solution in the same search space, each agent needs to consider the action of another agent as it affects their performance. Games where multiple agents with conflicting goals that are exploring the same search space (Othello Board) for the solution are called adversarial searches. Othello is a

Deterministic Game with Perfect Information. Meaning it follows a strict set of rules and there's no randomness in them. Agents can also look at the complete board. Agents have all the information about the game and they can see each other move also. The games is defined with the following variables:

- *Initial State*: Specifies how game is set up, we run our tests with varying random board states as initial states, to measure success of search in start, middle and end game scenarios.
- *Players*: White and Black or 1 and -1. The player pieces on board and its position on the board aids in evaluating state of maximizing player.
- *Action*: Legal moves in state space
- *Result*: Transition Model, specifies result of moves in state space. Such as what are the other players moves available moves now
- *Terminal State*: Game ends once all empty spaces have been filled or player has no more legal moves left.
- *Utility*: Utility function assigns a numeric value to terminal states based on rules of the game.

3.1 Heuristics

The evaluation function, also known as the heuristic or the static evaluation function is used to assess the state of the environment for the player. This evaluation allows a player to traverse the search space to their benefit over time. In Othello, there are several different factors that determine favorability of a position. Two main papers were used to create the aggregate heuristic function [7] [8], a brief summary of the heuristics follows: Component-Wise Heuristic Function

- **Coin Party** - Difference in coins between maximising and minimising player.
- **Corner Occupancy** - Corners are the most valuable pieces on the square. This heuristic calculates number of corners occupied
- **Corner Closeness** - Squares adjacent to corners are a huge disadvantage as it gives the opponent the opportunity to capture corner. Therefore, we avoid capturing close corner squares.
- **Mobility** - Measures how many moves player has. Restrict opponents mobility, restricts their reward paths.
- **Stability** - measure how vulnerable coin is to being flanked.
- **Frontier Disks** - Discs adjacent to empty squares have a greater chance of being flipped by opponent. Therefore we minimize number of frontier discs we have.

The two papers also proposed a separate utility function, a statically weighed board for each coin position, the heuristic is calculated by adding weights of the squares. The second papers weighed matrix performed better than the second one. Could be most likely due to higher weights on corner, given that, most games won, had the winning player with majority corner pieces. WE also played the heuristic against each other at a depth of 1 to assess their success relative to each other. Figure 4 shows the table of the averaged relative win percentage. However, playing at different board states, it could be seen that some heuristics were more effective in start game than end games scenarios as seen in Fig 5.

3.2 Search Algorithms

MiniMax Theorised in 1928 by John von Neumann [6] , it is the heart of most search algorithms for Sum-Zero games. It is used for fully observable and deterministic games such as Othello. but has also been extended to more complex games and to general decision-making in presence of uncertainty. Minimax is a Depth-First Search recursive algorithm. The search tree is made of states that

arise from the respective players' turn. The states are then evaluated with respect to the maximising player. During Max Players turn, it chooses the move with the highest evaluation and the Min Player choosing the move with the lowest evaluation.

```
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 := −inf
        for each child of node do
            value := max(value, minimax(child, depth−1, FALSE))
        return value
    else (* minimizing player *)
        value := +inf
        for each child of node do
            value := min(value, minimax(child, depth−1, TRUE))
        return value
```

Fig. 2: MiniMax Pseudocode

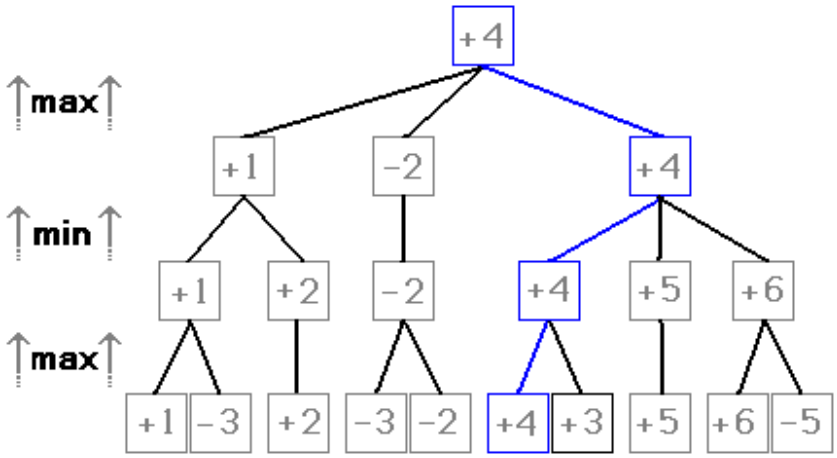


Fig. 3: MiniMax Example [9]

A-B Pruning Alpha-Beta [4] was published after Minimax to decrease the number of nodes that are evaluated by the minmax algorithm. It stops evaluating a move when it finds the move to be worse than an already examined moce. Such moves need not be evaluated and are hence cut of from the search along with their subsequent children. Alpha-Beta returns the same move but prunes

away branches that have no influence in final result. This pruning is facilitated by keeping track of parameters alpha and beta which create the cut-offs, and prevent the recursion through that node.

```
function alphabeta(node, depth, alpha, beta, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −inf
        for each child of node do
            value := max(value, alphabeta(child, depth−1,alpha,beta, FALSE))
            alpha := max(alpha, value)
            if alpha >= beta then
                break (* beta cut−off *)
        return value
    else
        value := +inf
        for each child of node do
            value := min(value, alphabeta(child, depth−1,alpha,beta, TRUE))
            beta := min(beta, value)
            if alpha >= beta then
                break (* alpha cut−off *)
        return value
```

Fig. 4: Alpha-Beta Pruning Pseudocode [10]

NegaScout (Principle Variation Search) NegaScout was invented by Alexander Reinefeld decades after Alpha-Beta pruning. NegaScout is a negamax algorithm that can be faster then alpha-beta pruning. It dominates by never examining a node that can be pruned by alpha-beta. However, it relies on accurate node ordering, otherwise it performs equally or worse than A-B pruning due to re-searching of nodes on proof fails. Move ordering is often determined by previous shallower searches. NegaScout produces more cutoffs than alpha-beta with the assusmption that the first explored node is the best (principle variation). Then, it tries to prove it by searching remaining nodes with a null window (scout window) which is faster than with a regular alpha-beta window. If the proof fails, first node wasn't the principle variation and the algorithm continues as normal alpha-beta.

If our assumption of the principle variation being the best move, we save time by searching every move other than the best move with a null window.

```
score := −pvs(child, depth−1, −alpha−1, −alpha, −color)
```

-alpha-1 and -alpha are the alpha beta values we pass to the next recursion. Since width of the window is only 1, the search will always fail if:

- It fails below alpha - the move is worse than we already have, so we ignore it
- It fails above beta - move is too good to play, so we can ignore it
- Otherwise, we need to do a new search properly

The three search algorithms were applied to the game of Othello and played against a decision rule AI that played random moves and stuck took the corner pieces if it had a chance. The heuristic

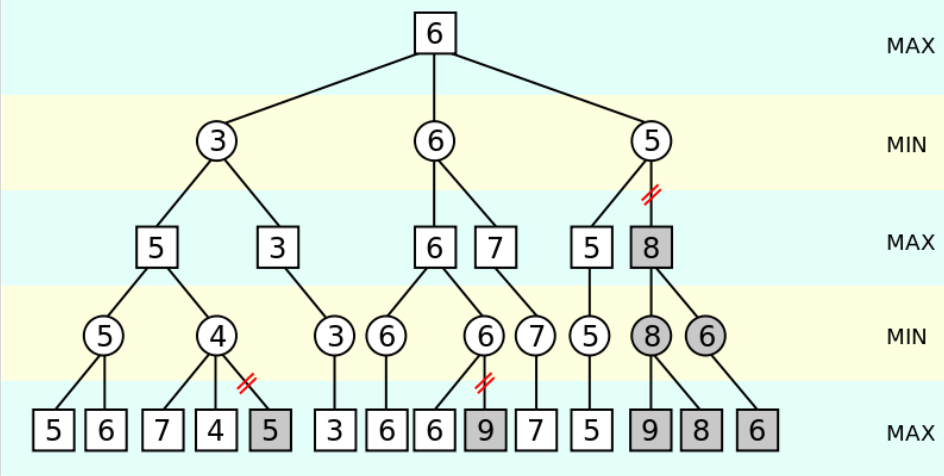


Fig. 5: A-B Pruning Example [10]

```
function pvs(node, depth, alpha, beta, color)
    if node is a terminal node or depth = 0
        return color x the heuristic value of node
    for each child of node
        if child is not the first child
            # search with a null window
            score := -pvs(child, depth-1, -alpha -1, -alpha, -color)

            # if it failed high, do a full re-search
            if alpha < score < beta
                score := -pvs(child, depth - 1, -beta, -score, -color)
        else
            score := -pvs(child, depth - 1, -beta, -alpha, -color)
        alpha := max(alpha, score)
        if alpha >= beta # beta cut-off
            break
    return alpha
```

Fig. 6: NegaScout Pseudocode [11]

is rendered less effective since the opponent is not following the same heuristic, but it would demonstrate how well the heuristic plays in guiding the game to its advantage against a random player. In the results section we demonstrate the efficiency of the algorithms in terms of nodes visited during a whole game and we demonstrate effectiveness of different heuristics when they are played against each other.

4 Convolution Neural Network Othello

Machine Learning, and particularly Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs), have seen a renaissance in the last decade. The introduction of techniques such as skip connections and the Rectified Linear Unit (ReLU) activation function have allowed deeper Neural Networks to be built. These have yielded large performance gains in the imaging areas of classification, object localisation, object detection, and image segmentation.

The power of CNNs lie in their ability to capture complex spatial patterns and have even been used in the landmark success of the AlphaGO program, achieving super-human performance [12]. Seeking to utilize these capabilities, Liskowski *et al* developed a DNN with 8 hidden layers to perform move prediction on the game of Othello [3]. To train their CNN, the authors used WThor[2], the French Othello League game dataset. This comprised 119,339 games between expert players, resulting in 6,874,503 board-move combinations, 4,880,431 of which were unique. On testing, the proposed solution achieved state-of-the-art move prediction, outperformed all 1-ply player algorithms, and also defeated the 2-ply version of Edax[3], quoted as "the best open-source *Othello* player" [3] at the time of publishing (2018).

Given the huge success of the aforementioned CNN, it serves as an ideal AI against which to compare some of the more classical approaches, such as tree-search and scout. In this work, we develop a CNN inspired by that presented in [3]. The design of the implemented network is laid out in Table 1 while further implementation details are presented in Table 2. The network was built in PyTorch, while fastai[4] was used for data-loading and training. Each Conv layer in the network is composed of: 1) A 2D convolution with a 3x3 Kernel, stride=1, and zero-padding. 2)Batch Normalisation.3) ReLU() activation.

It is worth noting that no max-pooling is performed to keep the feature maps at an 8x8 resolution. To train the network, it was desired to use the same WThor dataset used in [3]. This however was not possible without paying to access the Win-Test[5] software. Instead, another database containing 1569 games from 5 years was used. This resulted in 89,823 board-move combinations. To load these into the network, 3-channel images of size 8x8 were used. The first layer contains 1s where Player 1's pieces lie, with zeros everywhere else. The second layer contains 1s where Player 2's pieces lie, with zeros everywhere else. The third layer served as an optional matrix where legal moves could be embedded. This however was not ultimately used and was thus left empty. Examples of these boards are shown in Figure 7. Training required just 9 epochs, achieving a top accuracy of 48%.

[2] http://www.ffothello.org/informatique/la-base-wthor/
[3] https://github.com/abulmo/edax-reversi
[4] https://docs.fast.ai/
[5] http://www.win-test.com/

Layer	**Output Size**
Conv1	$8 \times 8 \times 64$
Conv2	$8 \times 8 \times 64$
Conv3	$8 \times 8 \times 128$
Conv4	$8 \times 8 \times 128$
Conv5	$8 \times 8 \times 256$
Conv6	$8 \times 8 \times 256$
Conv7	$8 \times 8 \times 256$
Conv8	$8 \times 8 \times 256$
fc1	1×128
fc1	1×60

Table 1: The 10 layers of the CNN presented in [3] and replicated in this research.

Loss function	Cross-Entropy
Optimizer	Adam
Learning rate	1×10^{-3}
Weight decay	0.01

Table 2: The network and training parameters used to train our CNN move predictor.

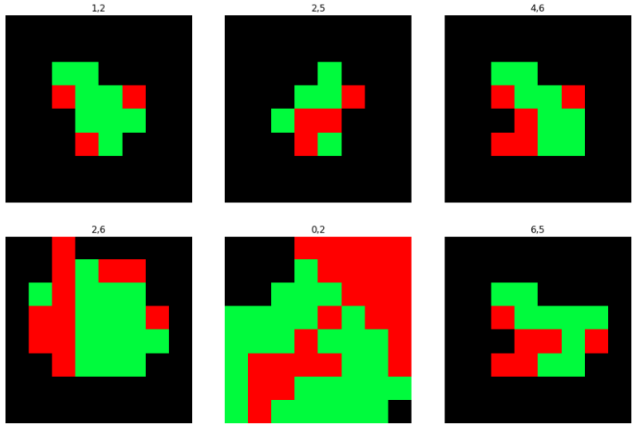


Fig. 7: Six example boards used in training the CNN. Shown in red are the locations containing the current player's discs while in green are the locations of the opponents disks. Black regions denote empty spaces. Written above each board are the x,y coordinates of where the player placed their disc.

5 Evaluations

5.1 Testing Environment

In creating a suitable bookmarking algorithm in which to compare the performance of all game playing strategies, a greedy algorithm was identified. This greedy algorithm, for each move, will always pick a corner tile if available or else picks the move from a list of legal options, which maximises the game score for the current turn. For each testing scenario, a Monte Carlo experiment was conducted for each operating condition (1000 runs per point). All simulations were built in a Python 3 environment, using the Numpy linear algebra tool.

Evaluations were conducted over a number of metrics and operating conditions. The first measures the expected win rate of the target strategy against the benchmark greedy algorithm. As the operating condition, the game turn start point was varied (i.e. for start point $T = 0$, it is a normal game, for start point $T = n$, the game is initialised at a random and legal game board state n turns in). The motivation for testing the algorithms under different start points are two-fold. 1) Othello is a deterministic game which makes Monte Carlo experiments trivial when the game starts from the same start position as the same game result will be achieved under every simulation run (given that the algorithms are also deterministic). 2) It is known that different strategies preform to different degrees of effectiveness for various points in the game, i.e. different strategies perform better or worse for beginning, middle or late game.

The second evaluation inspects the number of nodes considered by each of the adversarial search algorithms. This is a good metric to deduce the relative computational intensity for each algorithm. Each adversarial game playing strategy plays 1000 games against the greedy algorithm. The mean number of nodes considered (with a certainty interval of one standard deviation) is then recorded.

5.2 Node-Visit Comparison

?? and **??** show results of each algorithm against a standard Greedy game player benchmark. Figures — show the expected number of nodes visited by the adversarial algorithms per game, giving a mutual metric to compare computational overhead. The CNN cannot be compared under this metric as it adopts an empirical learning strategy rather than an adversarial search scheme.

_	All	Coin_Party	Stability	Frontier_Discs	Weight_Matrix	Corner_Closeness	Corner	Mobility
All	0.46	0.19	0.44	0.45	0.18	0.61	0.36	0.44
Coin Party	0.81	0.48	0.78	0.75	0.43	0.87	0.52	0.62
Stability	0.56	0.22	0.77	0.66	0.35	0.85	0.44	0.54
Frontier_Discs	0.55	0.25	0.34	0.44	0.19	0.64	0.22	0.29
Weight_Matrix	0.82	0.57	0.65	0.81	0.48	0.81	0.64	0.69
Corner Closeness	0.39	0.13	0.15	0.36	0.19	0.45	0.13	0.19
Corner	0.64	0.48	0.56	0.78	0.36	0.87	0.51	0.55
Mobility	0.56	0.38	0.46	0.71	0.31	0.81	0.45	0.5

Fig. 8: Heuristic vs. Heuristic. Statically assigned Weight Matrix outperforming individual and aggregate heuristic.

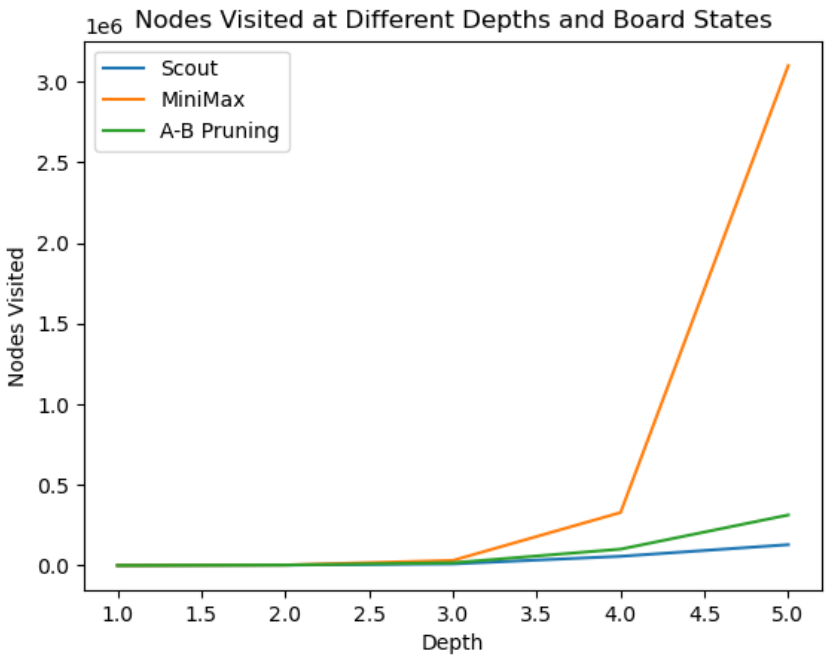


Fig. 9: Total Nodes Visited for a whole game, at random board states, depths and search Algorithms.

5.3 Relative Game Playing Performance

Figure 10 shows the expected win rate of each AI scheme against the benchmark greedy algorithm. A result of 0.5 indicates that the AI preforms wins 50% of the time against the greedy algorithm.

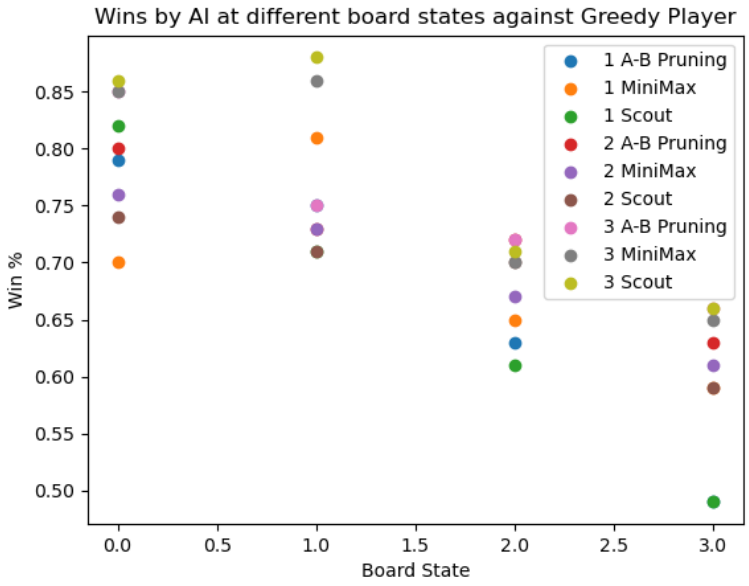


Fig. 10: Increasing depths resulted in more wins.

5.4 Discussion

The results of the average nodes visited for a whole game by a search algorithm demonstrates effectively the efficiency of the pruning techniques by alpha-beta and scout and the benefits of effective move ordering. Each one trump the other by almost a factor of a half. However, the search algorithm is only effective if an appropriate heuristic is chosen. From the papers we analysed individual heuristics, aggregate heuristic and a static weighed heuristic. Its shown clearly that a third dimension is required to evaluate a state, move/time. This was because some heuristics would perform worse at start game than end game and vice versa. To create a better weighed aggregate heuristic, one could apply a genetic algorithm that tries different weights on a linear combination of the individual heuristics. The improved performance of search algorithms depends on schemes in place that provide better move ordering such that more cut-offs can be made, these schemes include the use of killer move and history heuristics.

CNN is about 8MB while adversarial search methods are trivially sized. Computationally CNNs have a very small overhead while adversarial have a large overhead (talk about node stuff).

6 Other Algorithms

6.1 MTD(f)

Since Negascout, there have been several other relatively better performing algorithms that have been developed, such as MTD(f) [13], which derives its efficiency from zero-window alpha beta-searches and uses a transposition table to store and retrieve previously searched portions of the tree in memory, reducing space complexity. However it requires a transposition table to work will. Otherwise, the sub-tree has to be re-expanded.

Transposition Tables To further improve performance of search algorithms it is imperative to reduce repeated computations by recording already seen information. Since same position and hence subtree, is likely to be re-visited multiple times. This table would typically store score, search depth and best move and whether its an upper or lower bound.

Iterative Deepening (IDDFS) Iterative deepening Depth-First Search is a widely used state search strategy within which a depth-limited version of DFS is run repeatedly with increasing depths until goal is found. IDDFS is optimal like Breadth-first search, but uses much less memory, it visits nodes in same order as DFS but the cumulative order is effectively Breadth-First. IDDFS also provides good control of time, because results of previous iterations improve move ordering of new iteration, critical for efficient searching.

6.2 Bayesian Othello

Lee and Mahajan propose a Bayesian framework for Othello game playing [14] which adopts a set of 4 features/heuristics based off board mobility and board positions. Training games were labeled turn-wise as a win or lose move. Although not described (or even mentioned) in the paper, an inverse Wishart distribution must have been employed in conducting Bayesian inference on the unknown parameters of the generative function. Bayesian inference adopts the likelihood function (along with a presumed diffuse prior) to find parameters which makes the moves considered the most likely. The authors publish the (presumed MAP) parameters identified.

For decision making, the authors choose the move which maximises the maximum Likelihood (ML) ratio. This approach is consistent with maximum likelihood principle [15], but ignores information on the game known apriori (general) which is Incorporated in the Maximum A Posteri estimate (MAP).

The Bayesian algorithm performs well in general, competing in a number of world Othello competitions. Besides this, it is sensitive to feature selection and training data alongside the problem of accurately describing individual moves as win or lose.

6.3 Reinforcement Learning

Reinforcement Learning (RL) is is an AI strategy concerned with finding an optimal policy to maximise a given reward for any particular situation. Jan van Eck and van Wezel proposed in 2008 using RL to play the game Othello [16]. The motivation for RL was that to maximise payoff in a given game, the decision-maker might have to sacrifice immediate payoff for a greater future payoff (a task that the implementation of the CNN described in this paper cannot explicitly do). The approach adopts Q-learners alongside Neural Networks to build the model. Results found showed that the RL model was capable of beating human opponents. Besides results showing performance against different versions of itself and a rudimentary mobility and mobility decision rule AIs respectively.

7 Conclusions

Mention Oisin carols idea of symmetry in game board and how this can be taken advantage of (briefly). Talk about using player specific training data in NN. Talk about dynamic adding of weights to heuristic in minimax algos.

Future work in CNNs include increasing the information provided to the training model. Player skill should be accounted for when training. Methods to decide when a move is good or bad should be investigated further (our method is rather primitive). Investigations into building hybrid AI which uses different strategies for beginning middle and end game.

Bibliography

[1] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2):277–311, 2002.

[2] Hendrawan Armanto, Joan Santoso, Daniel Giovanni, Faris Kurniawan, Ricky Yudianto, et al. Evolutionary neural network for othello game. *Procedia-Social and Behavioral Sciences*, 57: 419–425, 2012.

[3] Paweł Liskowski, Wojciech Jaśkowski, and Krzysztof Krawiec. Learning to play othello with deep neural networks. *IEEE Transactions on Games*, 10(4):354–364, 2018.

[4] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[5] A Reineveld. Negascout-a minimax algorithm faster than alphabeta, 2003.

[6] Albert William Tucker and Robert Duncan Luce. *Contributions to the Theory of Games*, volume 4. Princeton University Press, 1959.

[7] Vaishnavi Sannidhanam and Muthukaruppan Annamalai. An analysis of heuristics in othello, 2015.

[8] Michael Korman. Playing othello with artificial intelligence. 2003.

[9] Variation tree. URL https://en.wikipedia.org/wiki/Variation_(game_tree).

[10] Wikipedia Contributors. Alpha–beta pruning, 03 2019. URL https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning.

[11] Eder Dıaz and Magdalena Flores. Evaluation of min-max and alphabeta variations on implementation of the strategy play: Dots and boxes. *Sistemas Inteligentes: Reportes Finales Ene-May 2014*, page 31, 2014.

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529 (7587):484, 2016.

[13] Aske Plaat. Mtd (f), a minimax algorithm faster than negascout. *arXiv preprint arXiv:1404.1511*, 2014.

[14] Kai-Fu Lee and Sanjoy Mahajan. The development of a world class othello program. *Artificial Intelligence*, 43(1):21–36, 1990.

[15] Ronald A Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 222(594-604):309–368, 1922.

[16] Nees Jan van Eck and Michiel van Wezel. Application of reinforcement learning to the game of othello. *Computers & Operations Research*, 35(6):1999–2017, 2008.