INTELLIGENT AGENTS FOR SOLVING THE GAME DOTS AND BOXES

Article · January 2020

CITATIONS
0

READS
2,039

2 authors:

Eliott Simon
Maastricht University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Cas Giepmans
Maastricht University
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Maastricht University

ARTIFICIAL, INTELLIGENT AGENTS FOR DOTS AND BOXES

Dührsen, Malte (i6119135) Fabris, Ionas (i6191631) Feyzi, Florène (i6199569)

Gauthy, Louis (i6188059) Giepmans, Cas (i6230772) Simon, Eliott (i6184618)

Project 2-1

January 22, 2020

Department of Datascience and Knowledge Engineering

Abstract

Interest in artificial intelligence is growing. Its application in games is hoped to produce new approaches that will be able to transfer to real-world applications. The present paper presents an investigation into AI-techniques in the context of the classic board game Dots and Boxes. A rule-based approach, Monte Carlo Tree Search, Q-learning, MiniMax, including alpha-beta pruning, were developed, and their performance compared. All new AI approaches struggled to compete with the traditional rule-based AI. Different variations of the above mentioned techniques were investigated, yet even with incorporation as these insights helped challenging the status of the rule-based AI. The implemented MCTS performed poorest.

1 Introduction

In recent years, public interest in artificial intelligence has been at an all-time high [Margasoft, 2017],[Hardy, 2016]. Despite being a topic of debate amongst public officials and industry leaders, the application of AI techniques has led to major improvements and developments in computer science, mechanical engineering, medical diagnostics, portable technology, and marketing.

This also holds true for the classic board- and card games. Since their invention, games like Chess, Checkers, and Black Jack and more recently Risk have been topics of research amongst mathematicians and game theorists. These types of games usually find their inspiration in some aspect of society or a problem in daily life; more often than not they resemble a battlefield or a war between factions.

Consequently, finding the best way of playing games and teaching artificial players how to play games has gathered interest within the field of mathematics and computer science. Examples of this phenomenon include matches and tournaments where the top human players and trained AI's representing a research group or company play against each. A recent example of this is the match between the world's top Go player Lee Sedol and Google's AlphaGo AI [Borowiec, 2016], where an AI beat a human in a game previously thought to be too hard for AI's to play.

Although AlphaGo beat Se-dol in the first match, it was Se-dol who beat the AI three matches later with an entirely unexpected move, after which the AI made a disastrous counterplay and consequently lost the match. This showed that there was still room for improvement within the AI. While other games are generally less complicated than Go, it remains up to researchers to find out which AI techniques fit each game best. This article focusses on the pen-and-pencil game Dots and Boxes, where the considered AI techniques are a MiniMax algorithm (both with and without Alpha-Beta pruning), Monte Carlo Tree Search, Q-Learning and a rule-based agent.

2 The Game Dots and Boxes

Dots and Boxes, also known as "La pipopipette", is a classic pen-and-paper game where two players try to fill a grid of dots with lines in order to create boxes. It has bee documented by several mathematicians and game theorists, such as Lucas Edouard in his book *"La pipopipette: nouveau jeu de combinaison", l'arithmétique amusante* published in the 19th century.

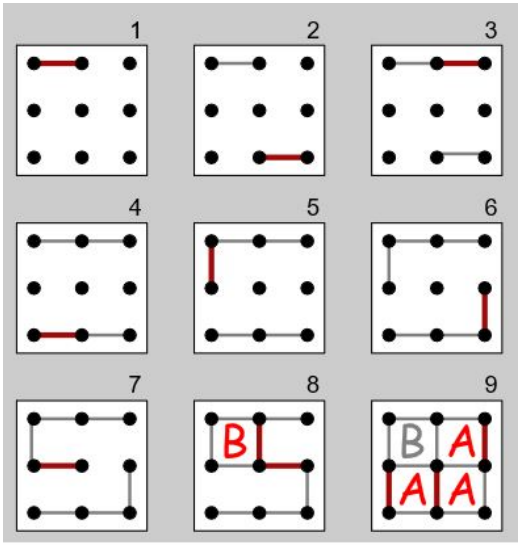

Figure 1: A Dots and Boxes games on a 2x2 grid. The most recently played line is marked in red. The player making the second move (frequently referred to as player B) is the first one to complete a square, and it is marked accordingly.

2.1 Rules and Characteristics of Dots and Boxes

Every player is subject to several rules whilst playing the game. Firstly, every player has to draw one line in every turn. The line can only connect two adjacent dots; diagonal lines and lines that connect two dots that are further apart are not allowed. Every fourth line between four dots that form a square completes a "box". When a player completes such a box and scores a point, he will get another turn, and thus he draws another line. After all the possible lines have been drawn by the players, the game is over. The player that completed most boxes wins. Draws can occur, but only on grid sizes that create an even number of squares (e.g., $2x2$, $3x4$).

Dots and Boxes defines itself with respect to other games by being deterministic: there is no chance-element present, such as a die or a shuffled deck of cards. Even the first lines that are drawn by the players, which might seem random at first, draw into the equation of who will eventually win the game. A second aspect of the game is that it is a combinatorial game. Combinatorial games are games that are played sequentially, meaning that every player performs an action after the previous player has done so, never both players in the same turn. It also has perfect information, which means that there are no hidden elements; all information about the state of the game is available to both players at any time.

2.2 Important Terms and Definitions

The following is a list wherein the most important terms and definitions that are used throughout this

article and relate to Dots and Boxes are explained:

- **Dot:** A point in the grid to which a line connects.
- **Line:** A connection between two dots. A line that has not yet been drawn is defined as being "empty".
- **Grid:** The playing field of Dots and Boxes. The grid size is given in how many boxes can be made in every row and every column.
- **Valence:** This is a measure of how many empty lines a box has. It can, therefore, vary between 0 and 4.
- **Box:** A box is an area in the grid that, when bounded by four lines, awards a single point to whichever player drew the fourth and final line.
- **(Long) Chain:** A chain is a sequence of boxes with valence 2, where every empty line is part of two adjacent boxes, except for the two outermost empty lines, which exist on the side of the grid. A long chain is a chain with a length of at least 3.
- **Loop:** A loop is a chain where the two outermost empty lines are the same empty line.

2.3 Strategy and Advanced Concepts

Before describing strategies and advanced concepts of the game, we define the player who has the first turn as player 1 and the player who has the second turn as player 2 for the remainder of the article. Every match of Dots and Boxes consists of two distinct phases: In the first phase, both players draw lines without completing any boxes. In this fashion, they divide the grid into chains, loops, and loose boxes. Once the drawing of any line results in the creation of a box with a valence of 1, the second phase starts. In this phase, both players try to complete as many boxes as possible by maintaining control of the game. Control is an attribute that a player is said to have if he can force the opposing player to get a second turn in which he initiates a (long) chain that can then be claimed by the controlling player.

Long Chain Rule

The Long Chain Rule states that player 1 should try to make the number of initial dots and eventual long chains even, while player 2 should try to make this number odd. Following this rule, the players might take control away from one another.

Double-dealing

One strategy a player can employ in order keep control is called "Double-dealing". To perform this strategy, the player in control completes all the boxes in a chain, except for the last two, creating two adjacent boxes of valence 1 that share their empty line. The turn now passes to the other player, who is the best possible move is to complete these two boxes and, given that he is now awarded an extra turn, has to initiate another chain. This is then exploited by the controlling player, who repeats and keep up this strategy until the game is finished. While making use of this strategy, it should be taken into account how many chains remain and what their length is. Otherwise the strategy might not be beneficial to the player using it, but to his adversary.

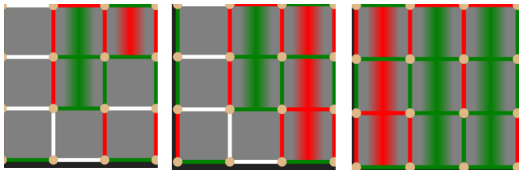

Figure 2: In the left grid, the agent forces the player to take the two boxes at the bottom right and then the player is forced to open the the last chain for the agent.

There are more strategies and special situations a player can encounter whilst playing, which apply the double-dealing strategy in similar forms. These may be found in [Berlekamp, 2006],

3 Method

3.1 Implementation

As part of the research, a digital version of the game was created where a human player can play against the AI's. Besides the possibility to play normally, we implemented the functionality, which allows us to simulate any number of matches between two AI's in order to perform experiments empirically. Given that reinforcement learning was also included in the scope of the research, we added the possibility of training a Q-learning agent against a random agent, as well as all the other algorithms we treat.

3.2 Agents

3.2.1 Game Tree

The Game Tree. Two of the AI's utilize a structure called a Game Tree. A game tree is a decision tree for which all of its nodes contain a game state. For Dots and Boxes, the game state is composed of the information on which lines have been drawn as well as the players' scores. The root of the game tree thusly contains the state of the starting point of the game. Every child node contains the state of a move played on the parent node state. The game tree is necessary for predicting and computing which moves are going to help a player to win. Using this tree, the AI's then have access to the different states of the game and search through them. It is not feasible to compute the full game tree as the number of With the memory space

provided through Java, all game states up to a depth of 11 can be stored, starting from the initial state of the game for a 3x3 board.e moves. After the first turn, 24 x 23 = 552 possible moves. Then in the turn after that: 12144 possible moves etc. With the memory space provided through Java all game states up to a depth of 11 can be stored, starting from the initial state of the game for a 3x3 board.

3.2.2 Rule-based

The first AI that we have implemented is a rule-based AI. The technique entails the implementation of (strategic) rules rather than searching through all the possible game states and finding the best move set. This makes it the most "human-like" AI, given that these same strategies are learned by people who play the game.

The rules that the AI applies are determined not just by their priority, but also by the phase the game is in.

- **Rule 1:** Ignore non-empty lines.

The AI will adhere to first rule, listed in 3.2.2, during both phases. Then for the first phase we define the following rules:

- **Rule 1:** Try to be the first player to enter the second phase.
- **Rule 2:** Do not create a box with a valence of 1.

In the first phase, the AI tries to be the first player to enter the second phase since, in that phase, it can win by applying the double-dealing strategy. However, it cannot employ these same rules in the second phase, since rule 2 from the first phase is in conflict with rule 2 from the second phase:

- **Rule 1:** Order all chains by their length (small to large) and try to complete them in that order.
- **Rule 2:** If beneficial, apply the double-dealing trick.

The second rule from 3.2.2 has the AI evaluate whether or not applying the double-dealing trick will net it more points than if it did not apply the trick.

3.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an algorithm that selectively explores the game tree by following a policy based on statistics of game states. It is, therefore, a best-first search algorithm. Every iteration of the algorithm consists of four phases. These are not described in turn.

1. **Selection** At the beginning of each iteration, the algorithm selects a node by recursively applying a selection strategy at each layer of the game tree. Several selection strategies exist in the literature. The present implementation uses the Upper-Confidence-Bound for Trees strategy. This strategy computes a score for every node based on the proportion of simulations based on this node that has led to wins, the number of visits to the node, and the number of visits to the parent node.

$$x \in argmax_{i \in I} \ (v_i + C \cdot \sqrt{ln_p/n_i})$$

Where x is the child from the set of the children I of the previous selection node with the maximum UCT score. The recursive selection process continues until it meets a node that does not have any children.

2. **Expansion** The children of the selected node are added to the game tree.
3. **Simulation** A game simulation based on the game state of the selected node is launched. This simulation is also referred to as play-out. Different strategies have been conceived for this. One is a random play-out; that is, all players choose moves randomly. If players follow a strategy or set of rules during the play-out, it is considered to be 'heavier'.
4. **Back-propagation** The result of the simulation is used to update the win statistic of the selected node and all of its parent's nodes.

In dots and boxes, there are symmetric moves that can be pruned in order to decrease the searching space. Every state has at least two equivalent representations, horizontally and vertically, and on square boards, diagonal symmetries can also be observed. Whenever a node is expanded in the search tree, we make sure that only one representation of a state is developed. From the literature [Barker and Korf, 2012], this optimization can resolve in a reduction of the searching space by a factor of 8 on square boards. Figure 2, gives an example where from 24 possible moves, at the first layer of a $3x3$ board, it's possible to identify 4 unique moves.

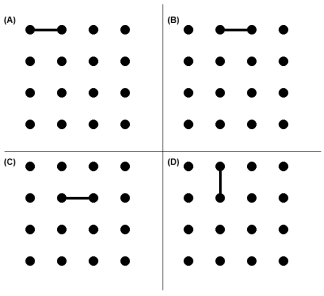

Figure 1: All the unique moves that can be taken by the first player at the start of the game. Every other line can be reached by applying one or two transformations.

Pruning the symmetric states during the expansion phase is not the only way to reduce the searching space. When using a classical tree implementation for MCTS, the search tree is going to expand nodes containing the same state more than once on the same layer.
This can be prevented by using a directed

acyclic graph implementation. Before a child of a node n is added to the tree, a search is done to check whether the child already exists as a child of another node. If such a node is found, instead of creating a new node, the child found during the search is used as a child of node n.

3.4 MiniMax

Overview
MiniMax is a backtracking-like algorithm that is commonly used to solve zero-sum games. The goal of this algorithm is to recursively investigate the game tree until we reach the last possible states of the game (last moves). Each state of the game will contain a score, given by an *evaluation function*. In the case of Dots and Boxes, this evaluation assigns a good score to a move that will give points to the player. On the other hand, it will give a bad score to a move which allows the opponent to finish the square in the future. The reason why MiniMax always gives the best solution regarding the evaluation function is that it considers that the opponent will make the best move, which means that it tries to minimize the maximum loss.

Structure
The algorithm takes as input the state of the game, the current depth of investigation of the tree and whether the move has to be played by the maximizing (our AI) or minimizing (the opponent) player. It is going to calculate the evaluation function at each level and thus determine if the move leading to the given state is a good move. As mentioned above, the algorithm will always select the opponent's best move. Hence, when investigating the opponent's level, it is always going to make a move with the *minimum* value, which, in fact, is the best move for the opponent.

MiniMax has a bad time complexity. For example, a 3x3 board already has 18! (6.4e15) states to investigate. Consequently, it is not feasible to traverse the entire tree, which is why we have to set a time limit for each move. Accordingly, this means that we are not traversing the entire tree; this algorithm might fail to find the best move.

MiniMax is deterministic; it is always going to give the same results for the same situation. However, this caused a problem when running tests against self since our results would be completely unbalanced (either wins 0% or 100% of the time). In order to solve this, we set a random permutation of the moves at each turn.

3.4.1 Alpha Beta Pruning

To optimize our time and space consumption, we have implemented a variation of the MiniMax algorithm, namely the alpha-beta pruning. As its name suggests, this algorithm is aimed to prune the MiniMax game tree. To do so, we have to keep track of two values, alpha, and beta, which will respectively represent our current maximum and minimum values. While traversing the tree, we will compare our node with this current value and prune the descendant of this node depending on whether it is lower than *alpha* or greater than *beta*. The effectiveness of the pruning depends on the order of the moves. In the worst-case scenario, the moves are ordered the completely wrong way, and no sub-tree will be pruned. The solution is to make a static evaluation of the moves at the current level. Afterward, it orders the move in such a way that we select the best node first (for the max. player), and the worst node first (for the min. player).

3.5 Q-learning

Q-learning is a Reinforcement-Learning algorithm. It is model-free, which means it does not learn from linear (or non-linear) functions. It is a reward-based algorithm that allows constructing very robust agents. Like all machine learning techniques, Q learning requires training sessions, during which the agent will be trained against another agent. The accuracy of our agent varies according to the nature of the agent it has been trained against. In the majority of the training sessions, the Q-Agent was confronted with a random agent, which draws random lines throughout the game. On the one hand, it means that it would be able to win against every player, given that the Q-Agent could learn from every possible state with enough time. On the other hand, this time would be unrealistically large. Another option is to train the Q agent against our Rule-Based AI. In this situation, it can learn how to apply the double-dealing strategy after it has been subjected to enough games.

There are two types of rewards in Dots and Boxes, and both depend on different parameters. The first type gives a reward depending on the outcome of the game *only*, which is negative if the Q agent loses the game, positive if it wins.

The second type gives rewards depending on the in-game score. Since winning Dots and Boxes depends on the amount of boxes the player has completed, it is important to add this condition to our Q-function as well. At each move, it rewards the agent when it earns a point. Since winning the game is important than earning a point, the reward associated with the outcome of the game has more influence on the Q-values than the other rewards.

The Q-function is implemented as a Q-Table. Each state that is investigated is mapped to its possible moves and the rewards associated with them. For the

sake of updating the Q-function, the following formula is computed at each state:

$$Q(s,a) = Q(s,a) * (1-\alpha) + \alpha * (\gamma * max(Q(s,a))) \quad (1)$$

where

- $Q(s,a)$ is the state mapped with its Q-values,
- α represents the learning rate of the system. In other words, it represents the correlation between our previous and next Q-values.
- γ represents the discount factor of the system. It is used to set the impact of the reward on our Q-Values.

4 Experiments

To investigate the strength of the implemented AI techniques, two types of experiments were set up. The intramethod experiments test the effectiveness of the techniques when variations are made in their parameters and/or configuration. The results of these experiments will have an effect on the other type of experiments: the intermethod experiments. Here, the strongest variations of each technique will be pitted against each other to determine the best AI.

4.1 Identifying the Best Variation

Rule-based agent
To learn about the value of adding double-dealing move to the repertoire of the rule-based agent, two variations were created. Apart from the capability of double-dealing moves, these are identical.
Hypothesis: The double-dealing variation will outperform the variation without this capability.

Monte Carlo Tree Search
The simulation step in Monte Carlo Tree Search can follow a random play-out in which the moves for both players are chosen randomly, or a 'heavy' play-out based on a set of rules. Heavy play-outs are more computationally expensive and therefore reduce the number of simulations that can be executed. Random play-outs on the other hand may consider many moves disadvantageous moves and therefore use simulations to evaluate weak moves. In our first experiment, we compare the playing strength of a random play-out and a play-out based on the rule-based agent described above. Hypothesis: The rule-based play-out will outperform the random play-out.

Q-Learning
The strength of a Q-Learning agent depends on its training time and the training environment that is used. (Anything else?) To find out whether the implemented Q-Learning agent would benefit from being pitted against the implemented rule-based agent rather than a random player, Q-Learning will be trained in both ways and compared. Hypothesis: The Q-Learning agent that is trained against the rule-based agent will outperform the Q-Learning agent trained against a random agent.

MiniMax and $\alpha - \beta$
$\alpha - \beta$-pruning generally double the depth of tree search in game playing agents. To learn whether the effect of pruning was similar in our implementation, average search-depth with and without $\alpha - \beta$ pruning was recorded. In addition, an option to randomize the moves order at each level has been implemented, in order to collect different results. Hypothesis: $\alpha - \beta$-pruning will approximately double the search depth [**?**].

4.2 Identifying the Best Method

To identify the AI with the highest winrate, we simulated 45 games for every combination of two AI's. The simulations were run on a 3x3 grid over approximately 5 hours. Since having the first or second turn has an impact on the outcome of the game, every AI will play 45 matches for both cases.

5 Results

Here we list the results using graphs and tables. You can state what you see happening in the graphs, but it has to be meaningful. For example: "Figure 5 shows the win-rate of the Q-learning agent using three sets of learning-rates and discount-factors. What becomes evident is that with a higher learning rate, the algorithm approaches it's maximum win-rate after fewer training iterations, but has an overall much lower win-rate."

5.1 Intramethod Comparisons

5.1.1 Rule-based agent results

To compare the rule-based agents, they were pitted against each other in games on different grid sizes. In Dots and Boxes, at certain grid sizes, the player to move first has and advantage or vice versa. Therefore, all games were replicated two times, changing the role of the first player. The result are listed in the following two tables.

P1: Rule-based - P2: Rule-based NoDD

Gridsize	Wins P1	Wins P2	Draws
2x2	26	13	11
3x3	18	32	-
4x4	17	30	3
5x5	15	35	-

Table 1a: Comparison of rule-based AIs. Variations without double-dealing plays second.

P1: Rule-based NoDD - P2: Rule-based

Gridsize	Wins P1	Wins P2	Draws
2x2	18	29	3
3x3	28	22	-
4x4	26	21	3
5x5	28	22	-

Table 1b: Comparison of rule-based AIs. Variations without double-dealing plays first.

5.1.2 Monte Carlo Tree Search Results

Computational Time	**1st player** MCTS	**2nd player** Stupid AI
	wins-rate	*win-rate*
0.33 seconds	10	39
0.66 seconds	8	42
1 second	7	40
30 seconds	8	41

Table 2a. Win rate of MCTS vs StupidAI depending on different computatiobnal Time

5.1.3 Q-Learning Results

Based on 20 000 simulations/line
A Q-Learning agent trained against a random player

Trained bot	Random Bot	
win	*win*	*draw*
8342	8460	3198
8388	8427	3185
8385	8463	3152
8489	8393	3118
8363	8482	3155

Table 3a. win rates of Q-Learning agent trained against a random agent

Trained bot	Rule-based agent	
win	*win*	*draw*
2324	9749	7927
2315	9668	8017
2374	9728	7898
2247	9742	8011
2367	9720	7913

Table 3b. win rates of Q-Learning agent trained against a rule-based agent

5.1.4 MiniMax and $\alpha - \beta$ Results

This experiment has been divided in two sub-experiments; with and without a randomized factor. The 1st and 2nd player have been switched to test all the possible variations. All the MiniMax and $\alpha - \beta$ experiments have been assigned 15 simulations per grid size.

With randomized factor

Board Size	**1st player** MiniMax	**2nd player** $\alpha - \beta$
	wins	*wins*
3x3	40%	60%
4x4	33.3%	66.6%

Table 4a. win rate of MiniMax vs. $\alpha - \beta$ with randomized factor

Board Size	**1st player** $\alpha - \beta$	**2nd player** MiniMax
	wins	*wins*
3x3	53.33%	46.66%
4x4	46.6%	53.3%

Table 4b. win rate of $\alpha - \beta$ vs.MiniMax with randomized factor

Without randomized factor

Board Size	**1st player** MiniMax	**2nd player** $\alpha - \beta$
	scores	*scores*
3x3	0%	100%
4x4	0%	100%

Table 4c. win rate of MiniMax vs.$\alpha - \beta$ without randomized factor

Board Size	**1st player** $\alpha - \beta$	**2nd player** MiniMax
	wins	*wins*
3x3	0%	100%
4x4	100%	0%

Table 4d. win rate of $\alpha - \beta$ vs.MiniMax without randomized factor

Depth experiments

Board Size	$\alpha - \beta$	MiniMax
	average depth	*average depth*
2x2	8	5
3x3	5	2
4x4	4	2

Table 4e. average depth of $\alpha - \beta$ vs.MiniMax

5.2 Intermethod Comparisons

[]

2nd Player →	Mcts	Rule Based	$\alpha - \beta$	MM
Mcts	0.45	0.0	0.0	0.0
Rule Based	1.0	0.45	0.5	0.4
$\alpha - \beta$	1.0	0.3	1.0	1.0
MM	1.0	0.3	0.0	1.0

Table 5a. Win-rate for the first player in a 3x3 grid.

6 Discussion

6.1 Rule-based

As observed, the rule-based agent working with the implementation of the double dealing is overall loosing against the rule-based agent deprived from this implementation. To our sense this results do not have a lot of sense since the double dealing trick is regarded as an strategical advantage. We then believe that our implementation should be modified.

6.2 Monte Carlo Tree Search

After running 50 games of the different variations of the MCTS agents. Hence, having 0.33seconds, 0.66seconds, 1seconds and 30seconds to run its calculations. It has been found that for a 3x3 grid, for all the different variations, we get when MCTS is playing against a random agent, it wins on average 15 percent of the time. Whereas if it is playing second, it wins 82percent of the time. Since there was a difference win rate of one or two games between the different variations of the MTCS agent, we considered it to be insignificant. We then observe that giving more time to MCTS agent to make its computations will insignificantly change the win rate of this agent.

6.3 Q-Learning

As discussed earlier, building a robust Q-learning agent for the game Dots-And-Boxes requires a huge amount of training games. We expect the training to be at least a million game in order to make a robust agent (for a $2x2$ grid). There is only a very small proportion of the number of states which can be stored and learn with 20000 games. Selecting the agent for the training also determines the accuracy of the Q-agent. By selecting the Rule-Based as 'model', we guarantee that our agent will master the double-dealing which is very important at high level.

6.4 MiniMax and $\alpha - \beta$

6.4.1 Randomized Factor

MiniMax and Alpha-Beta estimate the win rate of some given states. When it appears that some states have the same highest win rate, the incoming problem is "Which state to choose?". Without any randomized factor, the program always chooses the same state. Therefore, the same games are going to be played over and over. This is why a randomized factor has been introduced. This randomized factor will create diversity in the game. The 4 first table shows the win rate of $\alpha - \beta$ and MiniMax on different grid sizes depending on who plays first.
Table 4a and 4b give more random results with a slight better win rate for $\alpha - \beta$. When the board size increases, we can observe an augmentation of win rate for the second player.

6.4.2 Depth Comparison

To confirm that "$\alpha - \beta$ average depth search is approximately twice as deep as MiniMax average search depth"[MiniMax search and Alpha-Beta Pruning by Cornell University], some experiments were done comparing these two algorithms. The Table 4e. shows that the average depth of $\alpha - \beta$ is, in most cases, at twice as deep as MiniMax. These results were found by collecting all the different depths visited and an average was done on those depths for $\alpha - \beta$ and MiniMax.

6.5 Intermethod Comparison

Overall the best performance is displayed by the rule-based agent. It is followed by $\alpha - \beta$, MiniMax and MCTS in that order. The result vary depending on whether the agent plays first or second. This effect is especially pronounced for the rule-based agent. The results fall in line with analyses of the game stating that player B can win by 3 points if play is perfect. This can be observed as in the table. The rule-based agent's wins against MCTS is more pronounced when it is playing second. A surprising result is that Minimax, when competing against itself, only won when it played as second player. This effect translated when $\alpha - \beta$ competed against MiniMax, with $\alpha - \beta$ as first player. No wins were yielded.

7 Conclusion

In our current implementation, it becomes clear that all of the AI's struggle to compete against the alpha-beta agent and that the MCTS implementation is weak.

As a future improvement, another play-out for the Monte Carlo Tree Search could have been implemented. Instead of allowing the random play-out to run until the end of the game, the play-out can be stopped earlier and an evaluation function will be applied on the states to determine which are the ones that are more likely to win. This play-out is called Early Play-out Termination(EPT). This technique will be more computationally expensive than random play-out but still less than a heavy play-out which can be interesting to make experiences out of it.

Moreover, we are considering improving the QLearining agent by assigning all the symmetric states to a single state. We believe this will drastically improve the learning speed of the agent. This idea has already be implemented in the MCTS agent and applying it to the QLearning may not be a demanding task.

[]

2nd Player →	Mcts	Rule Based	$\alpha - \beta$	MM
Mcts	0.06	0.0	0.0	0.0
Rule Based	0.4	0.05	0.42	0.4
$\alpha - \beta$	0.53	0.51	1.0	1.0
MM	0.6	0.13	0.0	1.0

References

[Barker and Korf, 2012] Barker, J. and Korf, R. (2012). Solving dots-and-boxes. *Proceedings of the National Conference on Artificial Intelligence*, 1:414–419.

[Berlekamp, 2006] Berlekamp, E. R. (2006). *The dots and boxes game: sophisticated childs play*. Peters.

[Borowiec, 2016] Borowiec, S. (2016). Alphago seals 4-1 victory over go grandmaster lee sedol.

[Hardy, 2016] Hardy, Q. (2016). Artificial intelligence software is booming. but why now?

[Margasoft, 2017] Margasoft, M. C. (2017). Why is artificial intelligence becoming so popular?