Utrecht University

Master Thesis Artificial Intelligence

The Feasibility of Ignoring Opponents in Multi-Player Games

Job Hartjes
Student number: 5947502

29 January 2019

This thesis was prepared in partial fulfilment of the requirements for the Degree of Master of Science in Artificial Intelligence, Utrecht University.

Project supervisor: prof. dr. P. (Pinar) Yolum Birbil - Utrecht University
Daily supervisor: dr. B.J.G. (Bas) Testerink - Utrecht University

First examiner: prof. dr. P. (Pinar) Yolum Birbil - Utrecht University
Second examiner: dr. G.A.W. (Gerard) Vreeswijk - Utrecht University

Abstract

The application of AI in board games is an interesting area in the diverse field of Artificial Intelligence research. Board games allow for experimentation in an isolated fashion. Their clearly defined rules and the wide-ranging differences in complexity levels are ideal to test the performance of different AI approaches in various settings.

Board games with a large action space complexity prove difficult for AIs. This thesis introduces Agent Impact as a method to reduce this complexity and it investigates the impact of nine proposed agent properties have on the Agent Impact. This thesis further discusses whether these properties are indicative of an opponent's opportunity to lower the player's chances of winning, and if ignoring opponents based on these agent properties can improve player strength.

The experiments show that the performance of Agent Impact combined with the examined agent properties is strongly dependent on the domain. In Rolit ignoring opponents always leads to a lower performance, while in Blokus the performance differs per agent property. This thesis further concludes that Agent Impact has potential, but its application in the investigated domains does not improve the performance of Monte-Carlo Tree Search when the algorithm it is given enough computing time.

Keywords: Agent Impact, Selective Search, Monte-Carlo Tree Search, Multi-Player Search.

Acknowledgements

Firstly, I would like to express my gratitude towards my supervisors, dr. B.J.G. Testerink and prof. dr. P. Yolum Birbil, for their willingness to support me in my preferred research topic and their professional guidance. They mentored me, helped me focus my attention and kept me motivated. Thank you both. I literally could not have done it without you.

Secondly, I would like to thank Gijs-Jan Roelofs and Gerrit Drost, for their support on the technical aspects of this thesis. Whenever I had a question on data structures, game AI or encountered a programming-related issue they took the time to point me in the right direction. Thank you for your support, it helped me a lot.

Lastly, I would like to express the deepest of gratitudes towards my girlfriend, Lotte. I know my irregular sleep patterns and stressed out appearance were a burden on you as well. Yet, you remained supportive and patient with me. You mean the world to me. Thank you.

Contents

1 Introduction

The application of AI in board games is an active field of AI research. This is because board games are clearly defined, and the different complexity levels make board games an interesting abstract platform to experiment with AI approaches in an isolated fashion.

This research field first became famous after Deep Blue [1], a chess-playing algorithm developed by IBM, beat Garry Kasparov in 1997 and the research field is still advancing. Recently, *AlphaGo* [2], a Go-playing algorithm developed by DeepMind, beat Lee Sedol, a professional Go player. This was big news because Go is still considered to be one of the hardest games for computers to master.

When playing board games each player will plan ahead, trying to reach a favourable gamestate. In multi-player board games, with more than two players, the question “which opponents pose the largest threat to me” is then a relevant one. Players who know a game well can estimate how much influence an opponent has on the progress of the game. This gives these players the opportunity to focus on opponents that can more easily invalidate a game plan. This can lead to a competitive advantage, because these players can spend more time planning ahead, giving less thought to less influential opponents.

This thesis investigates the possibility of incorporating the selective ignoring of opponents in game playing agents by introducing the concept of Agent Impact. This Agent Impact value is then used to ignore opponent moves in the decision making process of the Monte-Carlo Tree Search (MCTS) [3, 4] algorithm, in an attempt to increase AI player strength.

This thesis refers to the agent that is running the search algorithm as the ‘player’ and to the other agents as the ‘opponents’.

1.1 Motivation

Decision making is a fundamental part of agent and AI research. Without the help of others, an agent has to be able to logically reason which of its possible actions leads to the best possible outcome. A measurement of how ‘good’ an outcome is, is generally referred to as the utility, or reward, of that state for the agent. An agent might try to maximise its own utility, the utility of a group, or minimise the utility of other agents.

Board games are well suited for AI research because they allow for research in an isolated fashion and have clearly defined aspects. Those are the rules, the state of the world, the possible actions and the winning or losing conditions. Another important aspect of board games is the different levels of complexity that exist. Games that have many possible actions every turn, and therefore a high branching factor, are difficult for computers to master [5]. This makes them an interesting research domain, because an AI cannot calculate the results of all future moves to determine which move leads to the best future state.

This complexity is a important factor in AI research. The more complex a domain or game is, the more difficult it is to make a prediction on the outcome of an action. This can be visualised well by a conceptual drawing of search trees, as seen in Figure 1. Here the dark grey represents the part of the search tree that is explored, while the white area remains unexplored. In games with a large branching factor the width of the tree, and thus the complexity, increases quickly, limiting the depth a search algorithm can reach. In these games it can be beneficial to remove parts of the tree to search deeper in the remaining parts, exchanging tree width for tree depth. In Figure 1 it is shown that the regular search tree explores all possible paths equally, but the pruning search tree, which ignores paths that are known to be unfavourable, reaches a higher search depth in the parts of the game it deems relevant. Finding a balance between exploring and pruning, and on which metrics to base the pruning, is at the core of AI research in board games.

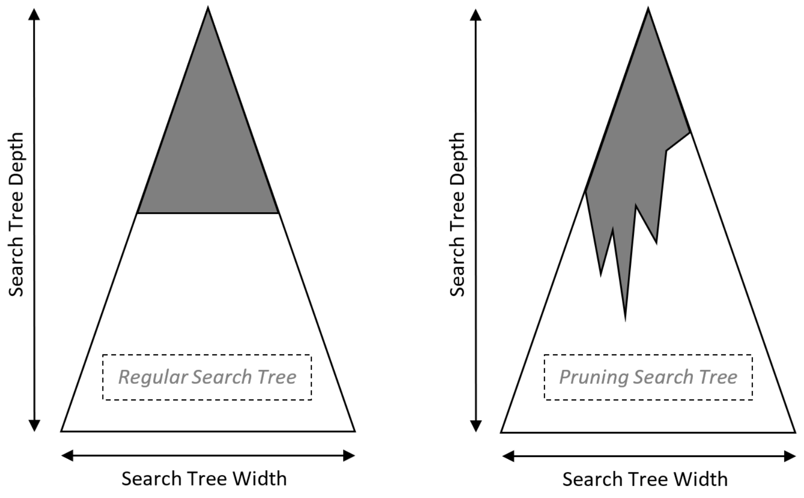

Figure 1: A conceptual representation of search depth in a regular search tree and in a pruning search tree.

The classic Minimax algorithm [6] builds a search tree and does not limit its search in any way. Many board games have an exponentially growing search tree, which results in a high search complexity. This makes the Minimax approach infeasible in these games, because it considers every node in the search tree. This makes it hard for computers to think ahead more than a couple moves. To improve the performance of computers in challenging domains, a lot of research has already been done in optimising tree search methods. This led to improvements to the Minimax algorithm, such as Alpha-beta pruning [5] in the 70's and more modern approaches, such as Monte-Carlo Tree Search (MCTS) [3, 4], in later years. Several advances in the field are discussed in more detail in the background section of this thesis.

One possible approach is the concept of selective search, introduced to improve the performance of classic tree search algorithms in high complexity situations. Selective search is generally done by evaluating how important each branch of the search tree is and prunes branches that are found to be of low importance. This gives the algorithm more time to calculate the outcomes of other branches. Best-Reply Search (BRS) [7] is an algorithm that incorporates this concept. BRS only considers the opponent with the "best" counter move, the other opponents are ignored. This algorithm is further explained in Section 5.

Another approach was the introduction of Monte-Carlo Tree Search (MCTS) [3, 4]. MCTS is an algorithm which builds a search tree where the value of any node is determined by the results of all simulations run from this or any child node. MCTS is a best-first search algorithm, which means the algorithm will, at each iteration, select the most promising action when building its search tree. By adding in a random component, it will select less promising actions a certain percentage of the time. This essentially focuses the algorithm's attention while not ignoring any option. Which action is the most promising is based on the outcome of many random simulations. One of MCTS's most notable accomplishments is its use in *AlphaGo*, created by DeepMind for mastering the game of Go [2] in 2016.

Using the selective search principle in MCTS programs seems like a promising approach. This is also mentioned by Schadd [7], who states that it might lead to an improvement in playing strength. A combination of BRS and MCTS has been researched [8, 9]. One such research paper focuses on different play-out policies for MCTS. A play-out policy is the way MCTS generates the samples it uses to evaluate a tree node. Usually this is done by making random moves from the state of the tree node until a stopping condition is reached. At which point the state is evaluated and the tree is updated with this information. This research investigated three different play-out policies, BRS, $\max^{\mathrm{n}}$ [10], and paranoid [11] in the games of Chinese Checkers and Focus. Using such algorithms for play-out policies requires more computing power than a random play-out, but the idea is that the quality of the play-out can potentially outweigh the disadvantage of less play-out samples. The results showed that although the quality of the play-outs is significantly improved, this benefit is countered by the reduced number of play-out samples and all approaches failed to greatly improve the performance. Of the three researched play-out policies paranoid performed the best.

However, this research did not address the potential performance increase that different forms of selective search can have in other aspects of the MCTS algorithm, such as the selection phase or using the selective search principle in a limited fashion during the play-out phase. Not using the full BRS algorithm can potentially retain much of MCTS's speed, because there is no expensive search being done during random play-outs. This leads to more samples and potentially better AI performance.

An idea that touches on the concept of selective search to improve the performance of the more classical algorithms, considers the impact of the opponents when choosing which search algorithm to use. This was the idea behind the MaxN-Paranoid mixture (MP-Mix) algorithm [12]. This algorithm uses a metric called the Opponent Impact to determine whether to search using $\max^{\mathrm{n}}$, paranoid or use a heuristic offensive rule-set to determine which action to execute, as not each algorithm performs the same in each situation. The Opponent Impact is defined as the number of states an opponent can reach which reduce the score of the root player, divided by the total number of reachable states. The score is given by a heuristic evaluation function.

Combining several of the ideas behind MCTS, selective search and Opponent Impact seems feasible. This thesis introduces the concept of the Agent Impact, which is an estimated level of impact an agent has on the performance of the other agents during a game. During a match, and during the algorithm search time, several agent properties can be defined. These properties are information describing a specific aspect of an agent, which can contribute to the Agent Impact value. An example of an agent property is the amount of possible actions an opponent can execute, and whether these actions increase its own score or lower the player's score. Others are the distance in turn order between agents or how well an opponent is doing overall. The previously mentioned Opponent Impact value is a possible agent property that can be used as a feature that contributes to an Agent Impact value. Which agent properties are a good predictor of the ability of an opponent's opportunity to lower the player's changes of winning is an interesting question. The exact definition of Agent Impact, its features, and how its value is calculated are detailed in Section 4.

This Agent Impact value is used to incorporate selective search to the MCTS algorithm during the selection phase, ignoring opponents that have a low Agent Impact. First the different agent properties are empirically tested separately, where the MCTS algorithm ignores opponents when the value of a single feature is below a threshold. Multiple threshold values are tested, and the best result is used to determine the weight of each feature. The weight of a feature indicates how strongly this feature contributes to the Agent Impact value. Finally, the agent properties are combined, with the found weights and thresholds, and the Agent Impact AI is empirically tested. The results show which agent properties are an indication an agent's opportunity to lower the player's chances of winning and if the combination of these features in the Agent Impact results in a good estimation of whether an agent can be ignored.

1.2 Problem Statement and Research Questions

To investigate the feasibility of selectively ignoring opponents by incorporating the concept of Agent Impact the following problem statement has been researched in this thesis:

- When is it feasible to ignore opponents in deterministic multi-player turn-based games with perfect information, and can this improve an agent's performance?

To answer this question the following research questions are investigated:

1. What agent properties are indicative of an opponent's opportunity to lower the player's chances of winning?
2. Can incorporating selective search, based on Agent Impact, into the MCTS algorithm improve player strength?

1.3 Contributions of the thesis

This thesis makes several contributions, which are listed as follows:

1. The introduction of the concept of using Agent Impact to ignore opponents;
2. The research into how much the proposed agent properties contribute to Agent Impact;
3. The incorporation of Agent Impact into the Monte-Carlo Tree Search algorithm and the performance research into this combination; and
4. The engineering of the open-source multi-game framework used to run all experiments, which is important for the reproducibility of the experiments and can be used during future research.

1.4 Outline

This thesis is organised as follows. In Section 2 the background of the topic is discussed, and an overview of the relevant research is presented. Section 3 discusses the research domains used in this thesis. Section 4 introduces the concept of Agent Impact and its proposed features. In Section 5 the relevant search algorithms are described in detail. Section 6 describes the experimental setup and Section 7 shows the results of the experiments. Finally, in Section 8 conclusions are drawn and future research is suggested.

2 Background

2.1 AI Research in Board Games

The application of AI in board games is an active field of AI research. There exist games that are relatively easy, but complex games which remain difficult for humans and computers to master exist as well. As stated in the introduction, *AlphaGo* [2], a Go-playing algorithm developed by DeepMind, recently beat Lee Sedol, a top-level professional Go player. Go is considered very hard for computers to master because of the complexity it has. On the 19 by 19 board there are too many possible actions, which makes examining the moves challenging. This complexity essentially makes it impossible to thoroughly search through all possible moves to find a good one.

On the other hand, less complex games, such as Connect Four, are solved. A game is solved when, from any gamestate, a computer can calculate if it will result in a win, a loss or a draw. This results in a boring game where the winner is predetermined if both players play optimally. For example, in the case of Connect Four the starting player always wins. Checkers is a more complex 2-player game and was weakly solved [13] in 2007. This means that for at least one player at the start of the game a path of actions is known that never loses. Chess is an even more complex game and, even though computers outperform the best human players, remains unsolved to this day.

There are complex 2-player board games, such as Go, but in general board games with more players are more challenging for computers than 2-player board games. There are generally more possible opponent actions between the player's own actions in the search tree. This makes it harder to control the progress of the game and predicting the outcome of an action requires more computation.

The complexity of multi-player games is often too large for a computer to exactly calculate the outcomes of the possible actions and select the best one. For example, an upper bound of $10^{46.7}$ possible states [14] was found for chess. Back in 1950 it was already estimated that a game of chess has about 10^{120} possible games [15]. A 4-player game of Risk has a state-space of approximately 10^{47} possible states with 200 units on the board [16]. A 19 by 19 Go board has 3 possible states per location, empty, white or black. This results in $3^{19\times19} = 1.74 \times 10^{172}$ possible states. A small percentage of these are valid states, but the remaining state space complexity is still many times larger than even a supercomputer can evaluate.

Another important aspect, next to the complexity, are the different properties that exist in the many different board games. These properties are important to understand in AI research, because certain AI techniques can be well suited given specific properties and perform worse in games with different properties. Several properties which can be used to classify games are (1) the number of players, (2) the availability of information, (3) whether the game has an element of chance, (4) whether the game is real-time or turn-based and (5) whether the decision space is discrete or continuous. Nijssen [8] also specifies the theme of the game and whether the agents have symmetrical goals as important properties. Themes can be connection games such as Hex and Havannah, territory games such as Go and Reversi, race games such as Chinese Checkers, capture games such as chess and Checkers, and tile-based games such as dominoes. If a game has symmetrical goals it means all players try to reach the same optimal state. Not all games have this property, an example of a game with a asymmetrical goals is the game of Mastermind, where one player has to choose a combination of colours and the other player has to guess this combination in a limited amount of turns.

After IBM's *DeepBlue* beat Garry Kasparov in 1997, Chess became less enticing for AI research. The attention was shifted to other games where it is, to this day, still not feasible for a computer to consider all possible actions. *AlphaGo*, for example, uses a combination of Monte-Carlo Tree Search (MCTS) [3, 4] and neural networks to decide which moves to explore further and which moves to effectively ignore in the game of Go. The field of AI research in board games is still an active one and has made important discoveries and developed useful algorithms over the years.

2.2 Two-Player Search Techniques

Many algorithms for games were developed over the years. Even before the rise of the computer. One of the most famous examples is **Minimax** [6] for two-player games, whose theory was published by von Neumann back in 1928. This algorithm represents the state space of a game as a search tree. Each child node in this tree represents the resulting state, and the edge toward that node represents an action. A search tree, such as the tree in Figure 2, is built by examining the results of each possible move. Here each square shows the reward for the starting player, whose decision states are shown in dark grey. After a search tree is built, the algorithm picks an optimal strategy by assuming both players play optimally. This is represented by a player maximising its own score, and an opponent minimising the score of the player. After simulating a game, the algorithm picks the best move that leads to the leaf node with the highest reward in its search tree.

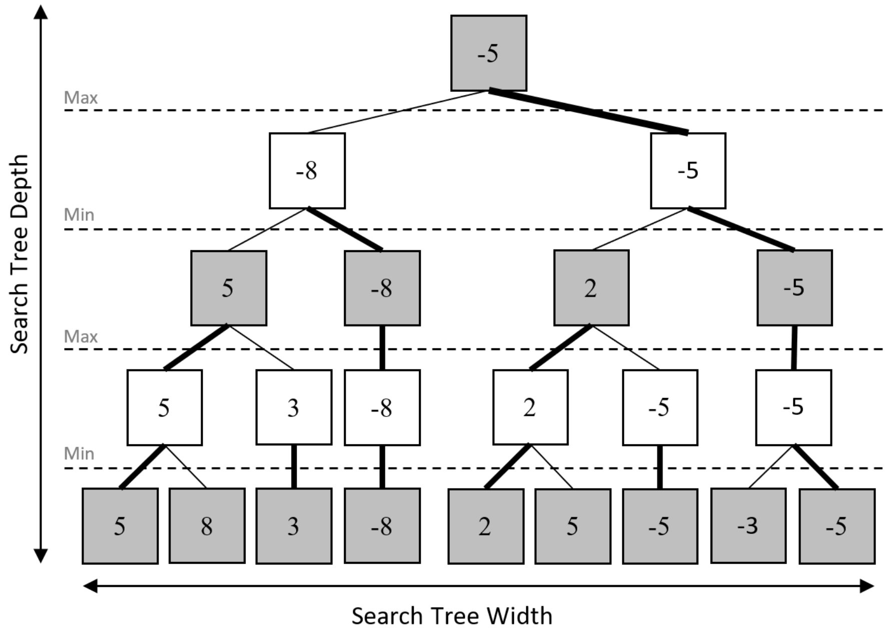

Figure 2: A typical Minimax Search Tree in a 2-player game.

Although this is a valid strategy for small games, it is infeasible for larger games. **Alpha-beta pruning** [5] improved that by eliminating entire branches from the search tree when the algorithm can prove these don't lead to an optimal strategy, making Minimax a feasible option in more games. The earlier the algorithm can prune a branch the larger the improvement in search speed, because all potential child nodes of that branch are skipped. **Negamax** is a Minimax variant for zero-sum games, where the increase in value for one player equals the decrease in value for the opponent.

In stochastic games or games with imperfect information it is not possible to find an exact path to an optimal reachable gamestate, because an action can have multiple results or because the gamestate is not fully known. **Expectimax** [17, 18], or Expectiminimax, tries to remedy this uncertainty by introducing chance nodes in the search tree. These chance nodes have as many child nodes as there are possible states from an action, with their corresponding chances of occurring. To get the score of the chance node the algorithm aggregates all child nodes' scores and probabilities of occurring.

Often it is not feasible, needed or even desired to search for an optimal decision. In time sensitive games the reward of a decision is impacted by when it is made. Because of this it can be beneficial to make a good quick decision, even when an action with a higher reward would have been found given more computing time. There also exist many games with action spaces so large it will practically take forever to finish a game by searching for optimal decisions. There are also situations where an optimal solution is not desired, even if it can be achieved within reasonable time. For example, most people would not want to play against an optimal agent, since we know we will most likely lose. In this scenario an agent with a quick response time is preferred over a stronger agent. For games that result in search trees that are too large to realistically build completely, or in situations where this is not needed, a **cut-off and evaluation function** [19] was proposed. This cut-off can come either after a certain amount of time, search depth, or search operations, resulting in a smaller search tree. At the cut-off point the evaluation function will score the resulting gamestate and this score is used to find the best decision. This algorithm will lead to faster decisions, but with less information, since the game will not be simulated until an end state. The performance of this approach is dependent on the quality and efficiency of the evaluation function.

2.3 Multi-Player Search Techniques

Multi-player games are in general a lot harder than two-player games for computers and humans alike. Classic search techniques like Minimax are not capable of simulating multiple opponents and the structure of Minimax, and exponentially growing search trees in general, can cope poorly with the increase in search space.

However, a prominent multi-player algorithm is a generalisation of Minimax. **Max$^{\mathbf{n}}$** [10], was proposed back in 1986. Max$^{\text{n}}$ builds on the idea of Minimax. Where Minimax has 1 maximising player and 1 minimising player, with max$^{\text{n}}$ each player tries to maximise its own score. Each leaf node of the search tree contains the resulting score of each player. During the traversal down the tree each agent will select the move with the highest resulting score for itself. Different methods have been proposed as tiebreakers if two nodes have the same score, one method is to resolve a tie randomly. Two downsides of max$^{\text{n}}$ are the limited amount of pruning that is possible, resulting in a large search tree, and the assumption that all opponents are merely maximising their own score can be false.

Figure 3 shows an example three-player max^n tree. Each node shows whose turn it is and the different scores for all agents beside it. When evaluating what options player 1 will have, the choice of player 2 is first examined. From node *b* player 2 would select the left branch, maximising its own score with a score of 2. From node *c* player 2 would select the left branch, maximising its own score with a score of 3. With that information player 1, in node *a* knows which options he has when selecting the left or right branch. In this case the right branch is more profitable, giving player 1 a score of 7.

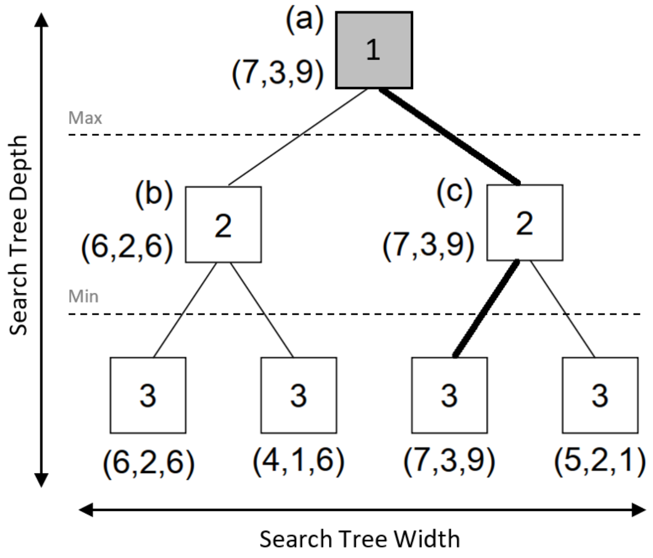

Figure 3: A typical three-player max^n tree, edited from [20].

The **paranoid** search algorithm [11], introduced in 2000, assumes all opponents form a coalition against the player. While max^n assumes all opponents try to maximise their own score, paranoid assumes all opponents try to minimise the score of the player. This allows paranoid to simulate a multi-player game with only 1 value in the leaf nodes, where max^n contains n in a n-player game, since the value is evaluated by subtracting the scores of the opponents from the player. This allows for alpha-beta pruning, resulting in a smaller search tree when compared to max^n. A downside of paranoid search is its defensive play-style, because the assumption that all opponents form a coalition against the player is often false. If paranoid search is given enough time to search until the end of a game, it is likely to assume all moves lead to a losing end state. This is because paranoid search assumes all opponents try to minimise the player's score.

A relative recent development is the introduction of **soft-max^n** [21] in 2006. This algorithm does not assume a single tie-breaking rule for an opponent, but it passes on the possible ties to its parent node. Where max^n only passes a single value up the search tree, soft-max^n passes a set of values up. These sets are compared using the dominance relationship. If a certain set strictly dominates another, the choice is clear for a player. If this is not the case, we know there are multiple options for a player. Soft-max^n incorporates this by letting an internal node be the union of all sets from its children that are not strictly dominated for the player whose turn it is. This means all possible states are passed on to the top of the tree, where it is up to the player how to use this information. It is shown an opponent model can be constructed, where soft-max^n is used to make inferences about the types of opponents we are playing [21].

Prob-max^n [22] was introduced in 2006 as well. Like soft-max^n it adds probabilities to the max^n. However, instead of adding probabilities to the values in the max^n sets, prob-max^n has multiple opponent models and tracks the utility of these models when applied to the opponents during a match. This means prob-max^n can determine which opponent model fits an opponent best throughout the game and use this model to estimate which actions this opponent will likely make. Which opponent model best matches the actual play style of the opponent is identified by using Bayesian inference.

2.4 Selective Search

There have been advancements with algorithms more closely related to the pruning approach in max^n and paranoid. Most notably the introduction of **Best-Reply Search** (BRS) [20] in 2011. BRS simulates a multi-player game as a two-player game by only allowing the opponent with the strongest move against the player to make that move during simulation, while the others skip their turn. This leads to a loss of information but allows the algorithm to search a lot deeper in the search tree, leading to better long term planning. BRS is further examined in Section 5.1.

A recent adaptation of BRS is **BRS$^+$** [23], introduced in 2014. Where BRS will search through invalid states that occur when opponents skip their turn when the game does not allow for this, BRS$^+$ only searches through valid states. This is done by using “move ordering moves” at opponent nodes that are not searched, which are essentially predetermined moves used to preserve valid gamestates and the player ordering. This does mean that the performance of BRS$^+$ heavily depends on the move ordering used in these moves where BRS would use a skip turn move, because these moves do change the state of the game. It was found that the BRS$^+$ algorithm generally performs better than BRS, although it does depend on which game is played.

2.5 Multi-Armed Bandit

The previously mentioned algorithms search for the action that will lead to the highest reward. This is done by one of two approaches, or a combination of the two:

1. Simulate all possible actions until the end of a game; and
2. Simulate all possible actions until a cut-off point and. evaluate the resulting state

However, there are many situations where both approaches are not feasible. First, there are games where the search space is too large due to a large branching factor. Second, a quick and accurate evaluation function does not always exist. In those cases the classic search techniques do not perform well, and another approach is needed that does not rely on exploring the full state space or the existence of a quick and accurate evaluation function.

One such approach is looking at a board game as a Multi-Armed Bandit [24] problem. Instead of determining the exact reward from the possible actions, the actions are viewed as having a chance of a reward. The Multi-Armed Bandit problem can be compared to a gambler at the slot machines in a casino, with a limited time to play. Not every slot machine will give the same reward, but how does he know which machine provides the highest reward over time? How often does he try a machine before moving on to another one? This perspective of looking at a board game can be used by an AI to estimate the reward of the different possible actions. By sampling from all actions for a limited time the player can estimate the pay-off any action will have. However, sampling all actions evenly is not the best solution, because this loses computing time in exploring unfavourable actions as much as favourable actions.

Monte-Carlo Methods are well suited to tackle the Multi-Armed Bandit problem and their usage lead to some of the most promising developments in AI game playing in recent years. One prominent algorithm is **Monte-Carlo Tree Search** (MCTS) [3, 4], introduced in 2006. One of MCTS's most notable accomplishments is its use in *AlphaGo*, created by DeepMind for mastering the game of Go [2]. Go is regarded as one of the hardest games for a computer to play, because of its large action and search space. MCTS aims to make a good decision by taking random samples of the action space and building a search tree. It recursively selects the most promising child node until a leaf node is reached, the action corresponding to this node is executed. Then the game is simulated to completion, or cut-off, by random actions and evaluated. The resulting evaluation is used to update the scores of all parent nodes. The concept of MCTS argues that if the initial move was a good one, the random play-out will have a higher chance of a good result. By simulating many games there is a high chance a good move is evaluated as such. MCTS is further examined in Section 5.3.

3 Research Domains

This thesis limits its scope to the domains of Rolit and Blokus. Both are deterministic turn-based board games with perfect information. Both board games are played with four players.

3.1 Rolit

Rolit is an extended version of the game Reversi, also known as Othello. Where Reversi is a two-player game, researched for decades [25], Rolit is a more modern four-player variant. The starting positions can be seen on Figure 4, where the question marks indicate the possible valid positions to take for the red player. During their turn players get to place a piece of their colour on the board. The player order is red - yellow - green - blue. A player is only allowed to place a piece at a position when that position has one or more straight lines between that position and a position occupied by the players own colour with at least one position in between, and the positions in between must be exclusively occupied by opponents. All positions between those two positions then take the players colour and the next player gets the turn. If no valid position exists, for example when a colour has been removed from the gameboard, the player can choose any unoccupied position adjacent to any occupied positions. The game ends after 15 rounds, when all 64 positions are occupied. The player with the most occupied positions wins.

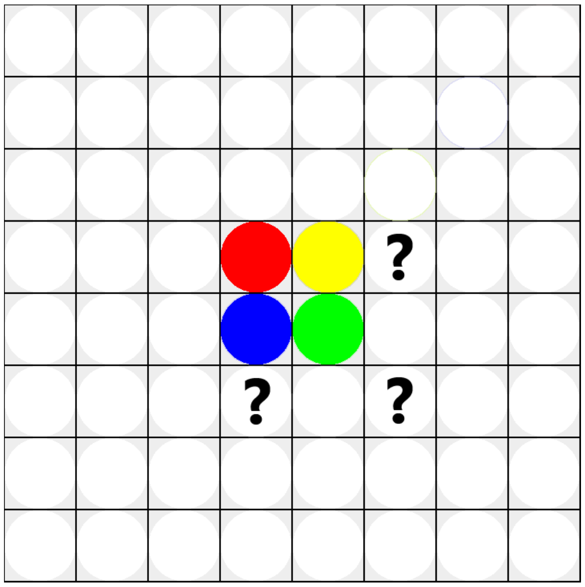

Figure 4: A Rolit board at the start of a match, with the possible actions for the red player indicated by question marks.

3.1.1 Complexity

The action space of Rolit is relatively limited, with only a total of 60 playable locations and generally 10 or less valid playable locations available to a player. This leads to a low branching factor. However, a single action can have a large effect on the game board in Rolit. The state of the game is very dynamic, as the board can look completely different in the three moves of your opponents. One single move can flip up to 18 positions, which is 28.125% of the gameboard. This property makes it difficult to construct an evaluation function that accurately determines the future worth of a gamestate, which makes Rolit a challenging game for an AI to master.

The state-space complexity of Rolit is relatively small. The board is 8 by 8, which is small. Next to the board the gamestate only has to store whose turn it is. A match of Rolit has a set length, after 60 moves the board is filled and the match is over. This allows the algorithm that checks if a game has ended to be very efficient.

3.1.2 Domain Properties

Rolit has several interesting properties. As stated before, it is a deterministic multi-player game with perfect information. It is turn-based and has a discrete decision space. The theme of the game fits best with the territory games such as Go and Reversi. These are games where a player tries to control certain territories on the gameboard to improve their chance of winning. Players have symmetrical goals, they all try to have as many locations occupied when the game ends.

3.2 Blokus

Blokus can be described as a multi-player adversarial puzzle game. Each player has a set of 21 pieces of specific shapes and sizes of either 1, 2, 3, 4 or 5 connected blocks. Players can place a piece on the board at positions where it connects to an already present piece of their own colour diagonally and does not connect horizontally or vertically to to piece of their own colour. The first piece of red should occupy the top left, for yellow the top right, for green the bottom right and for blue the bottom left corner. During their turn players must place a piece of their colour on the board if they can. The player order is red - yellow - green - blue. When a player has no more possible actions the other players are still allowed to continue. The game ends when all players are unable to place a piece on the board. The player occupying the most blocks wins. A typical finished Blokus board can be seen in Figure 5. This Blokus board shows that the different players must place pieces between each other to optimise their own score.

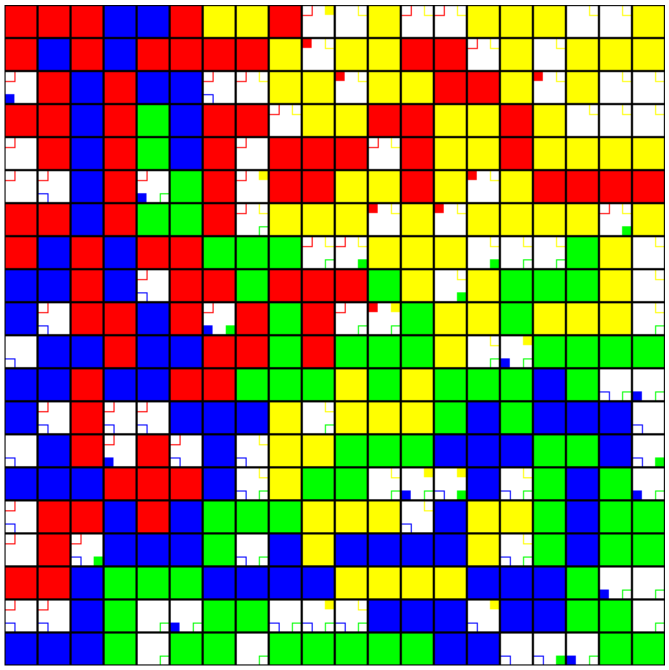

Figure 5: A typical Blokus board at the end of a match played by MCTS agents.

3.2.1 Complexity

The action space in Blokus is large, making it challenging for computers to master. During the first move of the match each player has 21 pieces. These can be played in any orientation, flipped or turned. This gives $21 \times 4 \times 2 = 168$ possible moves per player, although in practice this number is slightly lower, because symmetrical orientations don't have to be considered. Each player has a single location to play from, which makes the start of the game manageable. After 2 rounds each player can have up to 10 valid board positions to play one of the 19 pieces remaining pieces. This results in a maximum of $19 \times 4 \times 2 \times 10 = 1520$ possible moves per player. One such round of 4 players therefore has $1520^4 \approx 5,338 \times 10^{12}$ possible combinations of actions to consider. This gives Blokus a large branching factor.

The state-space complexity of Blokus is higher than Rolit. The board is 20 by 20, which is larger than Rolit's 8 by 8 board. Furthermore, where a Rolit gamestate only has to store the state of the board and whose turn it is, a Blokus gamestate also has to store which pieces are still available for each player. A match of Blokus has a variable length, the game continues as long as a player can make a valid move. Empirically the average match length was determined to be 58 moves, the shortest match took 46 moves and the longest observed match took 69 moves.

3.2.2 Domain Properties

Just as Rolit, Blokus has several interesting properties. It too is a deterministic multi-player game with perfect information. It is turn-based and has a discrete decision space. The theme of the game is hard to pin down. Its theme seems to fit best with tile-based games such as dominoes. It has characteristics fitting the territory games theme, because controlling sections of the board is an important aspect of the game. If a player can deny an opponent access to an area on the game board it limits this opponents' possible moves. Players have symmetrical goals, every player tries to maximise the number of board positions occupied by its own colour.

4 Agent Impact

The concept of selective search as proposed by Schadd in his BRS algorithm [7], tries to prune the search tree by considering only the opponent with the strongest move against the player. In essence this tries to determine the impact each opponent will have on the game, and ignore every opponent that does not have the best counter move, the maximum amount of direct impact.

Zuckerman et al introduced the **Opponent Impact** [12] factor for multi-player games in 2009, which measures the players' ability to impact their opponents' score. The important distinction in Opponent Impact is between influential states, states reachable by opponents that lower the player's game score, and normal states, where the player's score is not lowered. To reach an influential state the opponent needs to execute a counter move, a move that lowers the player's score. Opponents whose possible moves mostly reach influential states have more opportunities to lower the score of the player, which they defined as having a high Opponent Impact. The formula used by Zuckerman et al to calculate Opponent Impact can be found in Equation 1. Here G is a game and H is a heuristic evaluation function.

$$OpponentImpact(G, H) = \frac{InfluentialStates(G, H)}{TotalStates(G, H)} \tag{1}$$

The **MaxN-Paranoid mixture** (MP-Mix) algorithm introduced in the same paper uses Opponent Impact to determine whether to search using $\max^n$, paranoid or use an offensive tactic which focuses on lowering the score of the leading opponent. If the player is ahead, it will use paranoid to play more defensively, if an opponent is too far ahead, it will use an offensive tactic against that opponent, and when nobody is far ahead MP-Mix will default to $\max^n$.

Agent Impact, like Opponent Impact, aims to determine the impact an agent has on a game. Where Opponent Impact only uses the ratio of counter moves against all moves an opponent can execute, Agent Impact is not limited in what features it uses to calculates its value. Features exist that are near universal and can be applied in many domains, as seen in Section 4.1.1. Another type of features depends on domain-specific rules and characteristics to calculate the Agent Impact value, several of those as described in Section 4.1.2. A feature that considers the amount of counter moves an opponent can execute, as does Opponent Impact, is investigated as one of the agent properties that contribute to the Agent Impact value.

The proposed Agent Impact function evaluates all agents by one or more features. It multiplies the weight of a feature i by the corresponding score, and sums all features together to calculate the Agent Impact for an agent a in a given gamestate:

$$AgentImpact_a = \sum_i (Weight_i \times Feature_{ia}) \tag{2}$$

- $Weight_i$ is the weight of feature i; and
- $Feature_{ia}$ is the score of feature i for agent a.

The weights used in Equation 2 can be positive or negative values, or zero when a feature does not contribute to the Agent Impact. This approach allows features that potentially lower the Agent Impact to be considered as well.

4.1 Contributing Features

The contributing features are the most important aspect of Agent Impact. Depending on the domain a feature can be an indication that an agent has a lot of influence, while in another domain the same feature is a worse predictor of influence. The features researched in this thesis can be divided in two categories. The domain-agnostic features and the features that depend on specific domain knowledge by using a heuristic evaluation function.

These features can give an indication of how much influence an agent has on the game. If an agent has a lot of influence on the game based on these features, this will result in a high Agent Impact value. This value can be used by a search algorithm to determine which agents to ignore and which agents to consider in its search tree. Not every feature is as important as the other. Therefore, the weight of a feature indicates how much it contributes to the overall Agent Impact. These weights need to be tuned empirically. This is done by MCTS sampling, as explained in Section 6. If a feature does not predict agent influence correctly, the algorithm ignoring opponents based on this feature will suffer a performance decrease, which will result in a negative weight for this feature.

In the pseudocode provided the features return the next agent to be considered. Agents who's feature value is below the set Agent Impact threshold will be skipped. In the pseudocode the $NEXTAGENT$ function takes an agent as input and returns the next agent in the turn order. The $AGENTSCORES$ function uses an evaluation function to score a gamestate, this is done by counting the number of positions each agent occupies on the board. The $EVALUATEMOVE$ function returns the change in score before and after executing a given move for a given player and the $POSSIBLEMOVES$ function returns all valid moves a given agent can execute in a gamestate. The remaining functions used in the algorithms are detailed in their respective pseudocode.

4.1.1 Domain-Agnostic Features

Domain-agnostic features use information that requires no knowledge of the domain. This allows for these features to be used in any game without adaptation of the algorithm. Only the weights will have to be tuned per threshold per domain.

The position of agents in the player queue can be an important agent property. Opponents just before or after the player in the turn ordering might have more influence on the reward for the player. A domain might have fixed starting positions, where the interaction between agents differs based on these positions. There are several possible situations where this can influence the player strength and is therefore worthy of investigating. This turn ordering feature, which considers the position of an agent in the player queue, is described in Algorithm 1. Here *orderinglist* is a given parameter when the algorithm is started, this parameter contains the positions of opponents which the algorithm will ignore.

Algorithm 1 Feature: Turn Ordering

```
1: TurnOrdering($gamestate, player, orderinglist$)

2: nextagent ← NEXTAGENT(player)
3: turn ← 1
4: while orderinglist does not contain turn AND nextagent $\neq$ player do
5:     nextagent ← NEXTAGENT(nextagent)
6:     turn ← turn + 1
7: return nextagent
```

Another domain-agnostic agent property is the number of possible actions agents can execute. The intuition behind this feature is that often an agent with many options has a better board position than an agent with fewer options. However, this is not always the case, as with the game of Risk. When an agent has conquered North-America and South-America without any other territories, there are only 3 positions to attack from, but it is a strong position to be in. On the other hand, in the games of Rolit and Blokus it seems to be the case that having more possible actions is favourable for an agent. This can either be the absolute number of possible actions, as seen in Algorithm 2, or the percentage of actions an agent has compared to its opponents, which is specified in Algorithm 3.

Algorithm 2 Feature: Move Count

```
MoveCount($gamestate, player, threshold$)

nextagent ← NEXTAGENT(player)
moves ← POSSIBLEMOVES(gamestate, nextagent)
while moves < threshold AND nextagent ≠ player do
    nextagent ← NEXTAGENT(nextagent)
    moves ← POSSIBLEMOVES(gamestate, nextagent)
return nextagent
```

Algorithm 3 Feature: Move Percentage

```
MovePercentage($gamestate, player, threshold$)

nextagent ← NEXTAGENT(player)
movespercentage ← MOVEPERCENTAGE(nextagent)

while movesPercentage < threshold AND nextagent ≠ player do
    nextagent ← NEXTAGENT(nextagent)
    movespercentage ← MOVEPERCENTAGE(nextagent)
return nextagent

function MOVEPERCENTAGE($nextagent$)
    for all Agents a do
        if a ≠ nextagent then
            allMoves ← allMoves + POSSIBLEMOVES(gamestate, a)
        else
            moves ← POSSIBLEMOVES(gamestate, a)
            allMoves ← allMoves + moves
    return moves / allMoves
```

4.1.2 Domain-specific Features

Domain-specific features are features that use domain knowledge to calculate their values. These features must be implemented separately per game. A heuristic evaluation function is used to determine how good a gamestate or move is. An evaluation function is domain-specific and there are many ways to implement such an evaluation function within a domain. This thesis uses evaluation functions that count how many positions are occupied by each player. This is done to keep the work needed to apply the results to other domains minimal and keep the impact on computation time minimal.

The domain-specific features have the potential to determine the Agent Impact better when specific domain-specific knowledge is used. For many games, for example games that are solved, it is known which board positions are strong and what tactics give good results. If these factors are considered in the heuristic evaluation function, it is likely that these features end up with a higher weight in the Agent Impact value than the domain-agnostic features. The domain-knowledge based features in this thesis are aimed to be more widely applicable and therefore use little specific domain-knowledge.

One feature considers the overall game score of an agent, which seems to be an important agent property. A high game score is an indication that an agent has been playing well so far. It is possible that this is an indication that this agent will have more influence on the rest of the game compared to other agents. Logically, there exists a correlation in many games between having a high score and winning. The game score can be considered as an absolute number, as seen in Algorithm 4, or in comparison with the game scores of the other agents, which is specified in Algorithm 5.

Algorithm 4 Feature: Game Score

```
1: GameScore(gamestate, player, threshold)

2: nextagent ← NEXTAGENT(player)
3: score ← AGENTSCORES(gamestate, player)
4: while score < threshold AND nextagent ≠ player do
5:     nextagent ← NEXTAGENT(nextagent)
6:     score ← AGENTSCORES(gamestate, nextagent)
7: return nextagent
```

Algorithm 5 Feature: Game Score Percentage

```
1: GameScorePercentage(gamestate, player, threshold)

2: for all Agents a do
3:     totalscore ← totalscore + AGENTSCORES(gamestate, a)
4: nextagent ← NEXTAGENT(player)
5: scorepercentage ← AGENTSCORES(gamestate, nextagent) / totalscore
6: while scorepercentage < threshold AND nextagent ≠ player do
7:     nextagent ← NEXTAGENT(nextagent)
8:     scorepercentage ← AGENTSCORES(gamestate, nextagent) / totalscore
9: return nextagent
```

Another agent property that is likely to be indicative of a high opportunity to lower the player's chances of winning, is the ability of an opponent to increase its own game score. In many domains the agent with the highest overall game score wins when the game is finished. If opponents have higher game scores, the player will have more trouble reaching a high enough game score to win. This agent property can be considered in multiple ways. An agent can have a single move that clearly maximises the score of himself, the feature considering this scenario is specified in Algorithm 6. It is possible an opponent has multiple good moves, so taking the average of its moves seems to be a valid approach, which is formalised in Algorithm 7.

Algorithm 6 Feature: Best Move Score

```
BestMoveScore($gamestate, player, threshold$)

nextagent ← NEXTAGENT(player)
maxmovescore ← BESTMOVESCORE(gamestate, nextagent)
while maxmovescore < threshold AND nextagent ≠ player do
    nextagent ← NEXTAGENT(nextagent)
    maxmovescore ← BESTMOVESCORE(gamestate, nextagent)
return nextagent

function BESTMOVESCORE($gamestate, nextagent$)
    allmoves ← POSSIBLEMOVES($gamestate, nextagent$)
    maxmovescore ← 0
    for all Moves v ∈ allmoves do
        movescore ← EVALUATEMOVE(v)
        if movescore > maxmovescore then
            maxmovescore ← movescore
    return maxmovescore
```

Algorithm 7 Feature: Average Moves Score

```
MovesScore($gamestate, player, threshold$)

nextagent ← NEXTAGENT(player)
avgmovescore ← AVGMOVESCORE(gamestate, nextagent)
while avgmovescore < threshold AND nextagent ≠ player do
    nextagent ← NEXTAGENT(nextagent)
    avgmovescore ← AVGMOVESCORE(gamestate, nextagent)
return nextagent

function AVGMOVESCORE($gamestate, nextagent$)
    allmoves ← POSSIBLEMOVES($gamestate, nextagent$)
    avgmovescore ← 0
    for all Moves v ∈ allmoves do
        avgmovescore ← avgmovescore + EVALUATEMOVE(v, nextagent)
    return avgmovescore
```

Best-Reply Search showed that only considering the best counter move of an opponent can be a valid strategy in certain domains. Where the previous features considered the increase in game score for the opponents themselves, the following features consider the decrease in game score for the player. Opponents with a strong counter move can lower the player's chances of winning, if they execute that counter move. Considering this as a feature for Agent Impact seems like a promising approach. Opponent Impact showed that it can be beneficial to consider the amount of counter moves, moves that lower the player's game score, an opponent has. These two approaches are specified in Algorithm 8 and Algorithm 9.

Algorithm 8 Feature: Best Counter Move Score

```
1: BestCounterMoveScore($gamestate, player, threshold$)

2: nextagent ← NEXTAGENT(player)
3: bestcountermovescore ← BESTCOUNTERMOVESCORE(gamestate, nextagent)
4: while bestcountermovescore < threshold AND nextagent $\neq$ player do
5:     nextagent ← NEXTAGENT(nextagent)
6:     bestcountermovescore ← BESTCOUNTERMOVESCORE(gamestate, nextagent)
7: return nextagent

8: function BESTCOUNTERMOVESCORE($gamestate, nextagent$)
9:     allmoves ← POSSIBLEMOVES($gamestate, nextagent$)
10:    maxcountermovescore ← 0
11:    for all Moves v ∈ allmoves do
12:        countermovescore ← EVALUATEMOVE(v, player)
13:        if countermovescore > maxcountermovescore then
14:            maxcountermovescore ← countermovescore
15:    return maxcountermovescore
```

Algorithm 9 Feature: Counter Moves

```
1: CounterMoves($gamestate, player, threshold$)

2: nextagent ← NEXTAGENT(player)
3: countermoves ← COUNTERMOVES(gamestate, nextagent)
4: while countermoves $<$ threshold AND nextagent $\neq$ player do
5:     nextagent ← NEXTAGENT(nextagent)
6:     countermoves ← COUNTERMOVES(gamestate, nextagent)
7: return nextagent

8: function COUNTERMOVES($gamestate, nextagent$)
9:     countermoves ← 0
10:    allmoves ← POSSIBLEMOVES($gamestate, nextagent$)
11:    for all Moves v $\in$ allmoves do
12:        countermovescore ← EVALUATEMOVE(v, player)
13:        if countermovescore $>$ 0 then
14:            countermoves ← countermoves + 1
15:    return countermoves
```

5 Search Algorithms

This thesis combines concepts of different search algorithms. On the one hand the Multi-Armed Bandit problem is considered, by using MCTS as the main search algorithm. On the other hand, the more classical concept of tree search is considered. Specifically, the idea of selective search is considered. This is done by incorporating concepts from the BRS algorithm into MCTS.

Where BRS only looks at the direct counter-move of the opponents, this thesis will consider the Agent Impact value when deciding to prune a branch from the search tree or not. To understand these algorithms fully they are described further in this section, including the corresponding pseudocode.

5.1 Best-Reply Search

Best-Reply Search (BRS) is a more classical approach to decision making than MCTS. Instead of taking many samples, BRS builds a search tree like Minimax, but omits nodes it deems less interesting by applying a selective search method. This approach allows it to prune more and search further ahead in the game, but often not until the end states. This is why a limited search depth and good evaluation function are still important.

Max^n assumes that all players try to optimise their own score, and the paranoid algorithm assumes that all players try to minimise the root player's score. The assumption from max^n is more realistic than the paranoid assumption, but the paranoid algorithm allows for more pruning of the search tree. This allows the paranoid algorithm to search further ahead in the search tree, even though it might be less accurate. BRS tries to retain the large pruning capability of paranoid but is less paranoid. BRS tries to strike a good balance between depth and width by omitting all but one opponent's move per simulating step, in favour of a deeper search tree. Only the opponent's move with the worst score for the player is simulated. This means that in a game with more than 3 players most opponent moves are ignored, even if they might lead to interesting gamestates.

BRS considers all moves of its opponents and only the 'worst' opponent move for the player, the opponent with the highest direct impact, is added to the search tree. By omitting all other opponent moves BRS minimises the space between nodes where the player can make a move but does consider the move that most impacts its game performance and retains the ability to prune more branches. This approach essentially prioritises search tree depth to search tree width, omitting information in order to search further ahead. By omitting moves from opponents, the distance between an agent's own moves in the search tree is minimised, letting the agent focus more on its own performance, and that of the most threatening opponent, than that of all agents. This approach gave promising results [20]. Against max$^{\mathrm{n}}$ BRS won between 72% and 88% of games in Chinese Checkers and in Rolit BRS won approximately 65%. Against paranoid BRS performed well in Chinese Checkers and Focus, winning about 60% of matches, but in Rolit BRS performed worse when little computation time was allowed and performed about evenly when more computation time was allowed.

Figure 6 shows a typical BRS search tree for a three-player game where each player has two possible actions. Root node a represents the turn of player 1, and nodes b and c represent the turn of players 2 and 3. The edges are labelled to indicate which agents move was used to reach the new node. In the bottom row it is the turn of player 1 again, so he considers the possible states. Starting at node b player 1 considers the four possible results and the second move from player 3, as this gives a score of 2. Which is the lowest score and therefore the best counter move from player 2 and 3. Then node c is considered by player 1. When looking at the first node this has already a score of 1, thus player 1 knows that if he selects node c he is worse off than when he selects node b. Therefore, the other nodes are not considered and pruned from the search tree.

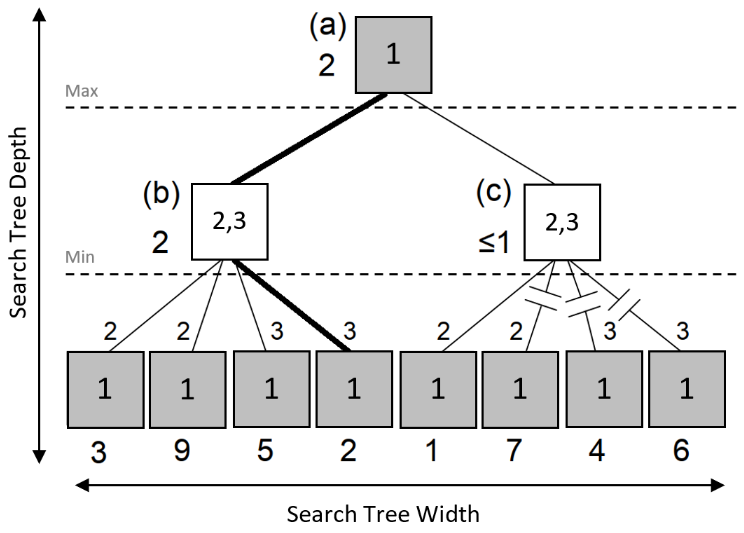

Figure 6: An example 3-player BRS tree, edited from [20].

This thesis takes an important concept from the BRS algorithm, the idea of selective search. By incorporating this into MCTS, combined with Agent Impact, the MCTS algorithm could be steered in a promising direction.

5.1.1 Pseudocode

The pseudocode for Best-Reply Search algorithm can be found in Algorithm 10.

Algorithm 10 Best-Reply Search

```
1: BRS($alpha, beta, depth, turn$)

2: if depth $\leq 0$ then
3:     return eval()

4: if turn == MAX then
5:     Moves = GenerateMoves(MaxPlayer)
6:     turn = MIN
7: else
8:     for all Opponents o do
9:         Moves += GenerateMoves(o)
10:    turn = MAX

11: for all Moves m do
12:    doMove(m)
13:    v = -BRS(-beta, -alpha, depth-1, turn)
14:    undoMove(m)

15:    if v $\geq$ beta then
16:        return v
17:    alpha = max(alpha, v)

18: return alpha
```

5.1.2 Complexity Analysis

To place the complexity of BRS in perspective, the complexities of $\max^{\mathrm{n}}$ and the paranoid algorithm are considered. When $\max^{\mathrm{n}}$ cannot prune its tree, the complete search tree will have to be explored. Given a branching factor b and search depth d, this gives a total of $O(b^d)$ nodes to explore. If the optimal path is taken and the maximum amount of $\alpha\beta$ pruning can be used, this best case results in $O(b^{d/2})$ explored nodes. Doubling the search depth d when keeping the explored nodes the same. The number of nodes explored in a best-case scenario for the paranoid algorithm in a n-player game is proven to be $O(b^{d(n-1)/n})$ [11]. However, this is done by assuming all opponents will form a coalition against the player.

The paper introducing BRS [20] proves the best case performance, in a similar way as the paranoid proof, to be $O\Big((b\times(n-1))^{\lceil\frac{2\times d}{n}\rceil/2}\Big)$ explored nodes. Where b is the branching factor, d the search depth and n the number of players. BRS increases the branching factor b to $b\times(n-1)$, because it considers the moves of all opponents at once. However, BRS reduces the search depth d to $\lceil\frac{2\times d}{n}\rceil$, because the layers of n players are reduced to strictly 2 layers. Adding $\alpha\beta$ pruning the search depth $\lceil\frac{2\times d}{n}\rceil$ becomes the $\lceil\frac{2\times d}{n}\rceil/2$ found in the final formula.

This complexity shows that BRS can generally search further ahead in the search tree than both $\max^n$ and paranoid. Paranoid reaches its complexity, which is better than $\max^n$, by assuming all opponents make the move that minimises the score of the player. BRS goes one step further and searches the tree by only allowing one opponent to make a move before giving the turn to the player again. Both paranoid and BRS sacrifice accuracy for the ability to search deeper down the search tree. This trade-off between accuracy and search depth yields different results in different domains.

5.2 Monte-Carlo Search

This thesis uses Monte-Carlo Tree Search (MCTS) during experimentation. Like MCTS, the Monte-Carlo Search (MCS) algorithm is based on a Monte Carlo method and a good starting point. It is used to evaluate the possible actions and determine which of those is a good action to execute. It takes random samples from the possible action space. MCS evaluates all actions an agent can currently take. When enough samples are created of those moves, the algorithm has an estimate of how good each move is, and it selects the best one. The algorithm has three phases, as seen in Figure 7.

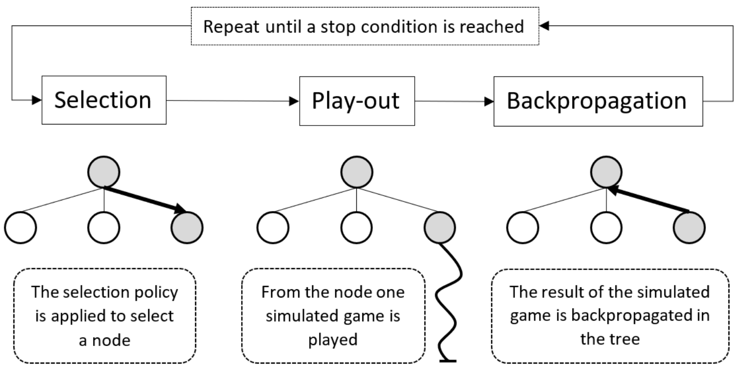
Repeat until a stop condition is reached
Selection
Play-out
Backpropagation
The selection policy is applied to select a node
From the node one simulated game is played
The result of the simulated game is backpropagated in the tree

Figure 7: A visualisation of Monte-Carlo Search.

During the **selection phase** MCS decides which of the possible moves to explore. There exist multiple selection policies. Selection can be done in a uniform manner, via Upper Confidence Bounds (UCB) [26], or another selection policy. Usually the UCB selection policy is used, as this was found to be a good selection policy, making MCS a best-first algorithm. This policy selects all possible actions at least once. When this is done it selects the most promising node with a probability of $100 - c$, where c is for the exploration parameter. The exploration parameter ensures MCS also considers actions that looked bad initially, but are truly quite good. This parameter c is often tuned empirically for the best results.

After an action is selected, MCS enters the **play-out phase**. During this phase the selected action is executed, and the game is simulated with random actions until an end-state is reached. It is possible to stop the simulation before an end-state is reached. When such a cut-off play-out is used, MCS requires an evaluation function to score the resulting state. This typically must be an efficient and simple evaluation function, because the performance of MCS depends heavily on the number of simulations that can be run. During the play-out phase, MCS can use a play-out policy different from the default purely random actions.

When the simulation has ended, the resulting state is evaluated. When an end-state was reached this is a simple check for who won, scoring the state as either 1, for a win, or 0, for a loss. When the simulation was cut-off the evaluation function must evaluate the resulting state and score it between 0 and 1, depending on how good the result is for the agent. In the **backpropagation phase** the resulting score is propagated back up the search tree.

If at this point the algorithm does not reach a stopping condition, such as a maximum on iterations, running time, or explored nodes, the MCS algorithm will enter its selection phase again. When a stopping condition is reached, the MCS algorithm will look at the scores it gave to all possible actions and returns the action with the highest score.

5.3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a best-first tree search algorithm. It builds upon the MCS algorithm by adding the expansion phase. This allows MCTS to add the newly selected state to a search tree structure, as seen in Figure 8. The most widely used selection policy is UCB applied to trees (UCT) [27].

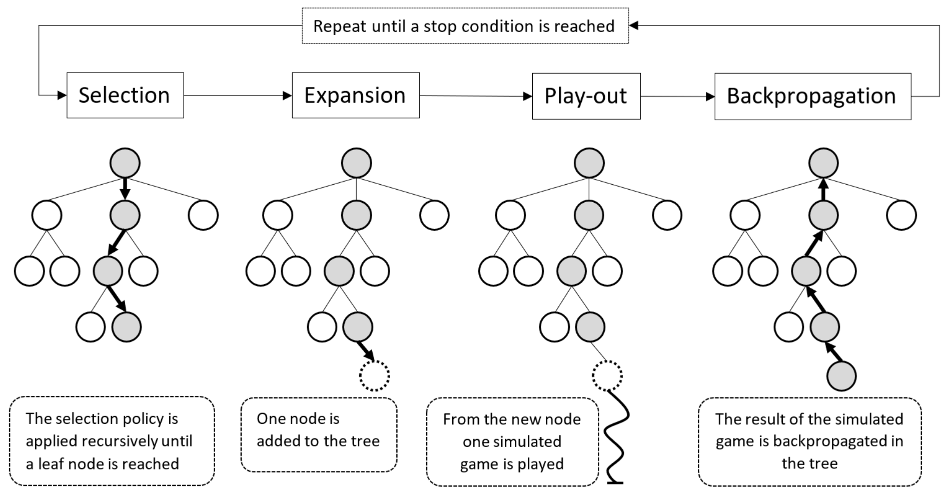
Repeat until a stop condition is reached
Selection
Expansion
Play-out
Backpropagation
The selection policy is applied recursively until a leaf node is reached
One node is added to the tree
From the new node one simulated game is played
The result of the simulated game is backpropagated in the tree

Figure 8: A visualisation of Monte-Carlo Tree Search.

The UCT algorithm recursively selects the most promising child node most of the time. By tuning the exploration parameter, the algorithm can be tuned to explore nodes that are not as promising more or less often, effectively tuning the balance between exploring other nodes and exploiting a promising node. The most promising node is selected the most to ensure the search tree is expanded mostly in a promising direction.

Selecting the most promising node is done by recursively selecting a child node j, until a leaf node is reached. This is done by maximising the following formula:

$$\underset{v' \in children of v}{\operatorname{argmax}} \left(\frac{Q(v')}{N(v')} + c \times \sqrt{\frac{\ln N(v)}{N(v')}} \right) \tag{3}$$

- $Q(v')$ is the current value of child node v';
- $N(v')$ is the number of visits of a child node v';
- $N(v)$ is the number of visits of a parent node v; and
- c is the exploration parameter.

The current value of a node is calculated by dividing its total value by the number of times the node was selected. As a result, when a child node was not yet visited it will have an infinite value. This ensures Equation 3 explores all child nodes of a parent node at least once before it starts selecting the most promising child node. If a leaf node was selected the algorithm enters the **play-out phase**, which works the same as with the MCS algorithm.

After the play-out phase the algorithm enters the **expansion phase**, where the information gained from the play-out is added to a new node. After this, during the **backpropagation phase**, the parents of the added node are recursively updated with the new information. Then the algorithm will either reach a stopping condition, such as a maximum on iterations, running time, or explored nodes, or start the selection phase again starting from the root node. When the algorithm stops, it will check the child nodes of the root node and select the action that shows the most promising results.

As an example, an MCTS search tree corresponding to the Rolit start state, seen in Figure 4 from Section 3.1, can be found in Figure 9. The red player has 3 possible actions and the MCTS algorithm does not favour one over the other a lot. It seems that the first action is explored more often, and thus selected in the end, but the difference in visits between the actions seems limited. This indicates that all possible actions have similar chances of producing a good result.

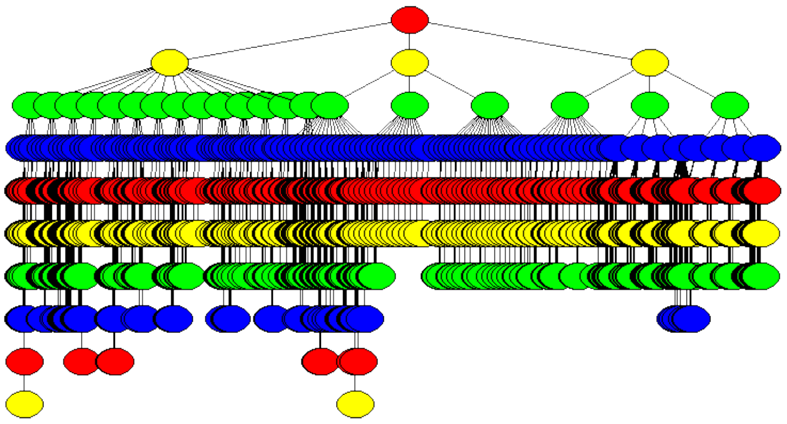

Figure 9: An MCTS Tree at the start of a Rolit match.

The MCTS search tree corresponding to the Rolit gamestate seen in Figure 10 looks completely different. This tree, seen in Figure 11, favours one of the seven possible actions. The 6th action is explored many more times than the other actions, meaning this is the most promising action and in the end this action is selected by the MCTS algorithm.

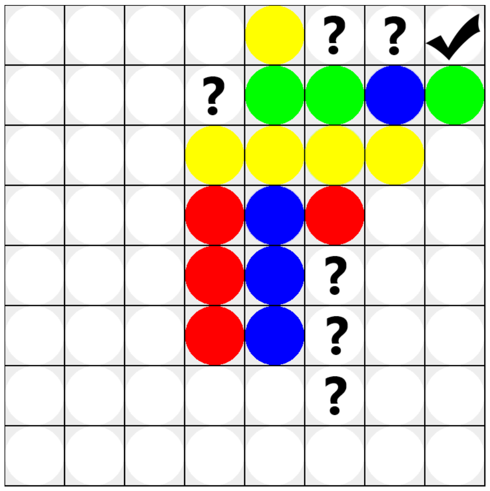

Figure 10: A Rolit board, with the possible actions for the red player indicated by question marks and the most promising action by a tick mark.

Intuitively this makes sense, looking at the Rolit board in Figure 10. The top right position allows the red player to maximise its score in the short term by claiming three positions from other players. In addition it is a powerful position itself, as it cannot be claimed by other players and it creates many future possibilities.

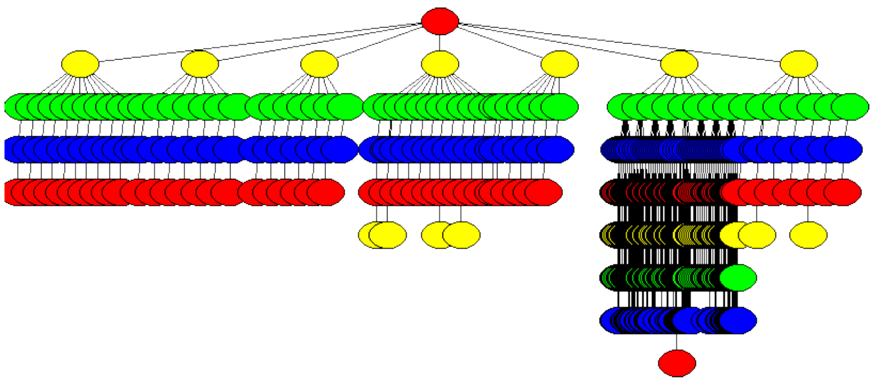

Figure 11: The MCTS Tree corresponding to the Rolit board seen in Figure 10. The 6th action is visited most often and the most promising.

This is a good example of MCTS exploring the more promising actions more often. When MCTS is given the opportunity to create a large enough sample size, relative to the action space, it achieves good results.

5.3.1 Pseudocode

The pseudocode for Monte-Carlo Tree Search algorithm using the UCT selection policy can be seen in Algorithm 11. Here s_0 is the state of the game when the agent gets to execute an action. The pseudocode is based on previous work [4] and adapted to reflect the implementation used during the research done for this thesis.

Algorithm 11 Monte-Carlo Tree Search with UCT

```
1: function MCTS($s_0$)
2:     create root node $v_0$ from state $s_0$
3:     while stop condition not reached do
4:         $v_1 \leftarrow$ TREEPOLICY($v_0$)
5:         $\Delta \leftarrow$ PLAYOUTPOLICY($v_1$)
6:         BACKPROPAGATE($v_1$, $\Delta$)
7:     return $a$(BESTCHILD($v_0, c$)

8: function TREEPOLICY($v$)
9:     while $v$ is nonterminal do
10:        if v not fully expanded then
11:            return EXPAND($v$)
12:        else
13:            $v \leftarrow$ BESTCHILD($v$, $Cp$)
14:    return $v$

15: function EXPAND($v$)
16:    choose $a \in$ untried actions from $A(s(v))$
17:    add a new child $v'$ to $v$
18:         with $s(v') = f(s(v), a)$
19:         and $a(v') = a$
20:    return $v'$

21: function BESTCHILD($v$, $c$)
22:    return argmax$_{v' \in childrenofv} \left( \frac{Q(v')}{N(v')} + c \times \sqrt{\frac{\ln N(v)}{N(v')}} \right)$

23: function PLAYOUTPOLICY($s$)
24:    while $s$ is nonterminal do
25:        choose $a \in A(s)$ uniformly at random
26:        $s \leftarrow f(s, a)$

27: function BACKPROPAGATE($v, \Delta$)
28:    while v is not null do
29:        $N(v) \leftarrow N(v) + 1$
30:        $Q(v) \leftarrow Q(v) + \Delta(v, p)$
31:        $v \leftarrow$ parent of $v$
```

5.3.2 Complexity Analysis

The amount of explored nodes of BRS can be defined by search depth, branching factor and number of players. This cannot be done with MCTS. MCTS mostly selects the most promising node when traversing down the search tree but does not prune the others. Therefore, the branches of an MCTS tree are not evenly build, making it difficult to specify the amount of examined nodes in the same terms as used in the complexity analysis of BRS.

MCTS is an anytime algorithm, meaning it can be terminated before it is truly finished and still return a valid answer. However, MCTS will generally find a better answer when given more time. Where BRS can have a stopping condition in the form of a limited search depth, MCTS commonly has a stopping condition in the form of a time limit or node limit. When a node limit is used as a stopping condition, the amount of explored nodes becomes $O(v)$ where v is the amount of nodes in the stopping condition. When a time limit is used the number of explored nodes is $O(\frac{s+e+p+b}{t})$, where s the time needed to select, e the time needed to expand, p the time needed to simulate a play-out, b the time needed to backpropagate the result, and t is the time limit. The number of explored nodes is dependent on the efficiency of the play-out phase, as this generally is the phase of the algorithm where most time is spent.

5.4 Monte-Carlo Tree Search with Agent Impact

Agent Impact, or one of its contributing agent properties, is used as a selection-phase amendment to the MCTS algorithm, as a tree policy. Agents with a low Agent Impact value are ignored when selecting the next node to explore. This allows MCTS to skip tree nodes where an agent with a low Agent Impact gets the turn. This can be seen in Figure 12. When comparing this to Figure 11 the most obvious difference is that a row is no longer a specific colour. The shown search tree is created by an Agent Impact AI using the Move Count Feature, as specified in Algorithm 2, with a threshold of 2 moves.

From the tree in Figure 12 it is clear that yellow has a relatively low Agent Impact. During normal play the player order is red - yellow - green - blue, but in this tree, yellow is almost completely skipped after red's first move. Only when red takes one specific action yellow's impact is high enough to warrant a tree node to be expanded. However, this move did not give the best results during the play-out phase, as it is not the most explored. The third node was explored most by the MCTS algorithm, which represents the best move to make.

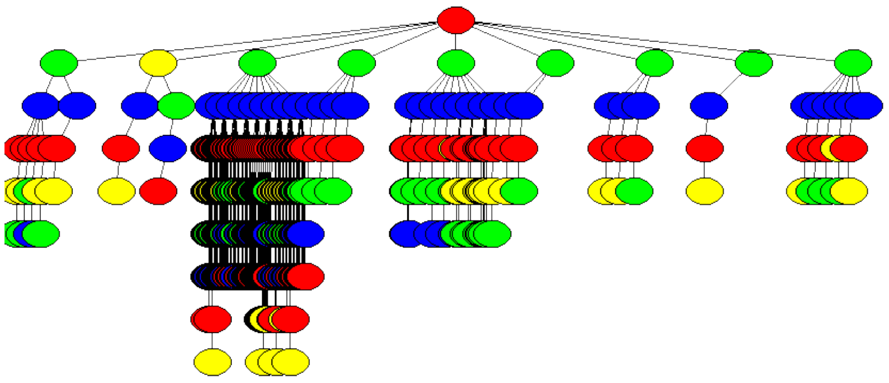

Figure 12: An MCTS Agent Impact tree corresponding to the Rolit board position seen in Figure 13.

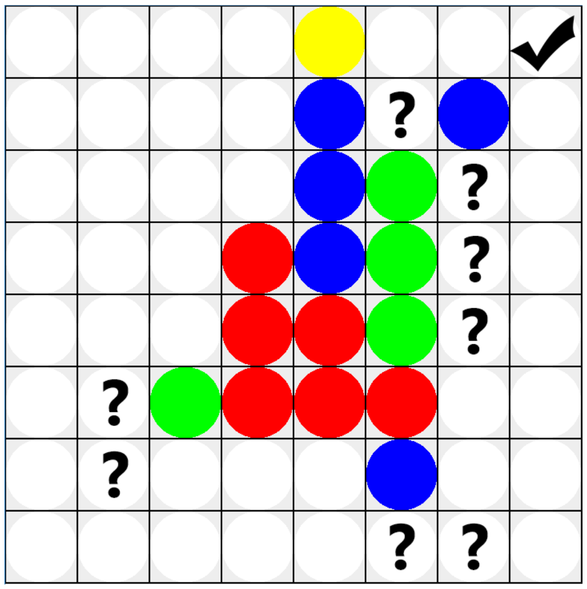

Figure 13: A Rolit board, with the possible actions for the red player indicated by question marks and the most promising action by a tick mark.

The tree graph in Figure 12 shows that the concept of incorporating Agent Impact into the MCTS algorithm is functional. However, which of the agent properties described in Section 4.1 are good indicators of the agent's opportunity to lower the player's chances of winning is still to be determined. As well as whether this approach improves player strength. Which combination of weights and agent properties works best to determine the Agent Impact, and when the

Agent Impact value warrants an agent to be ignored, is addressed in Section 6 and 7.

5.4.1 Complexity Analysis

MCTS with Agent Impact has the same complexity formulae as normal MCTS. When a node limit is used as a stopping condition, the amount of explored nodes is $O(v)$ with v being the amount of nodes in the stopping condition. When a time limit is used the number of explored nodes is $O(\frac{s+e+p+b}{t})$, where s the time needed to select, e the time needed to expand, p the time needed to simulate a play-out, b the time needed to backpropagate the result, and t is the time limit.

Even though the amount of created nodes is the same, the depth of the tree represents something different. In a regular MCTS tree, each level of search depth represents a move by the player or an opponent, as seen in Figure 11. Here a row of nodes always has the same colour, which represents the player whose turn it is. In a search tree constructed by MCTS with Agent Impact, a level of search depth represents a move with possibly one, two or three skipped opponents. This means a row of nodes can have different colours, different players whose turn it is, as seen in Figure 12.

The more agents are deemed to have a low Agent Impact, the further ahead the algorithm can search, but the more potentially relevant information is omitted. It is important the algorithm has a good indication which opponents can be ignored. Creating a good trade-off between searching further ahead and retaining the opponent information is important for the performance of the algorithm.

6 Experiments

6.1 Node Limited vs Time Limited

The performance of MCTS tends to increase when more time is given to create more random samples. When given enough time, or when a match is nearing the end of a game, MCTS can even construct the entire remaining search tree. This property is dependent on the hardware used to run the simulations. Therefore, many MCTS related papers specify the used hardware and the set limit on the run time. However, this is not a precise description of the testing environment for multiple reasons. There are still many undefined parameters. For example, modern processors tend to adjust the core clock speed to save power or perform better in short bursts, a computer will have background processes running, hardware defects influencing performance, and many more parameters. Using an iteration, or node, limit for the MCTS algorithm produces results that are easier to reproduce and verify.

To keep this thesis comparable to other works in the field of MCTS research the used hardware and average run time to reach the iteration limit is specified as well. The simulations were run on a Windows 10 Desktop, with an Intel Core i5-4430S CPU and 32GB RAM. While this CPU has a base clock speed of

2.70Ghz, the hardware monitor reported that it was clocked at 2.80Ghz during the simulations.

In Table 1 and Table 2 the relation between the number of nodes expanded and the run time is shown for an MCTS agent during its second turn. These are the average results over 10 runs. The aforementioned tables show that searching through Blokus nodes takes longer than searching through Rolit nodes. The higher state complexity and decision complexity of Blokus can explain this. The larger board size is one factor, but the state also contains all pieces agents are still holding. Cloning this state takes longer than the Rolit state, which only needs to store the 8 by 8 board. In addition, it takes longer to generate all possible actions from a state, which greatly increases the time needed to do a random play-out.

Run time in ms	**Nodes Expanded**
154	500
281	1000
351	1500
471	2000
594	2500

Table 1: The ratio between run time and explored nodes in Rolit.

Run time in ms	**Nodes Expanded**
786	500
1504	1000
2060	1500
2716	2000
3450	2500
4217	3000
4757	3500

Table 2: The ratio between run time and explored nodes in Blokus.

6.2 Tuning Monte-Carlo Tree Search

Before the different features of Agent Impact, described in Section 4.1, are experimented with, MCTS will be tuned. The exploration constant is a tuneable parameter, as is the number of iterations the MCTS algorithm performs before deciding which action to perform. These parameters are tuned empirically to determine which values produce the best results. Since the parameter tuning is not the core research topic of this thesis it is handled here in the experiments section and not in the results section. The results of the parameter tuning can be found in Appendix A.

For Rolit the results suggested that the best exploration constant is 0.4 and the results do not significantly increase after 1500 nodes, while the simulation time does increase. The Rolit experiments were therefore run with an exploration constant of 0.4 and with a node limit of 1500 nodes.

For Blokus the results suggested that the best exploration constant is 0.2. The number of nodes for Blokus was set to 2000 nodes. This number is larger than for Rolit, because of the larger branching factor of Blokus. Additionally, the simulation time of a Blokus experiment is higher, as it takes MCTS about 13.5 times longer to generate the required number of nodes per turn. As a consequence, the Blokus experiments were run less times and the results generally have a larger confidence interval. These experiments were run with an exploration constant of 0.2 and with a node limit of 2000 nodes.

6.3 Hypothesis

6.3.1 Agent Properties

Several agent properties have been described in Section 4.1. How well these indicate an opponent's opportunity to lower the player's chances of winning is very dependent on the domain in which they are applied. Overall, it is expected the domain-specific features give a better indication of an opponent's opportunity to lower the player's chances of winning.

The Turn Ordering feature is expected to show an increased influence of opponents just before or after the player in the turn ordering. The features considering the amount of moves opponents can execute are not likely to be a good indication in the domains considered in this thesis, because these features do not consider the result of these moves. It is possible an opponent has many moves, while only a few negatively impact the player. The Game Score features are believed to be indicative of Agent Impact, because a high game score directly correlates with winning a game. The Move Score features are also expected to be indicative of Agent Impact, because good moves lead to a higher score, although less than the Game Score features. It is likely that a high overall game score is a better predictor of winning than the ability to make good moves during one turn. The Counter Moves features are expected to be the clearest indication of an opponent's opportunity to lower the player's chances of winning. Even though an opponent does not have to execute the moves that lower the player's game score, the fact that they can lower the player's game score means they will do it if this increases their chance of winning.

6.3.2 MCTS Selective Search

It is expected that incorporating selective search, based on Agent Impact, into the MCTS algorithm does not improve player strength if MCTS is given the time to create enough samples. This is because MCTS is a best-first tree search algorithm. It naturally prefers to explore more promising nodes. Because of this property, pruning nodes based on Agent Impact is likely to only improve player strength if the used features accurately determine the opponent's opportunity to lower the player's chances of winning, or if MCTS cannot create enough samples to accurately estimate the value of the available actions.

The results are expected to show which features are a good indication of an opponent's opportunity to lower the player's chances of winning, and which features had little influence. However, given the nature of MCTS, it is expected that most features are not indicative enough to warrant a removal of the MCTS search tree. The volatile nature of Rolit makes it unlikely that ignoring opponents improves upon MCTS, while the more stable nature of Blokus can improve upon MCTS if the feature is indicative enough of an opponent's opportunity to lower the player's chances of winning.

6.3.3 The feasibility of ignoring opponents

The main research question in this thesis is "When is it feasible to ignore opponents in deterministic multi-player turn-based games with perfect information, and can this improve an agent's performance?".

Rolit is a very dynamic game where each move can have a large impact on the status of the gameboard. It is expected that ignoring opponents based on the specified features does not increase agent performance if MCTS is given the time to create enough samples. If a feature is indicative enough of an opponent's opportunity to lower the player's chances of winning, ignoring opponents might increase player strength when MCTS is limited to create fewer samples, or in more classical algorithms with limited search depth.

Blokus is a game with a large branching factor where at the start of a game the agents have little interaction with each other. In the mid game and late game, the agents can influence each other's performance greatly. This characteristic of the game suggests that features that allow ignoring agents in the beginning of the game increases player strength, because large parts of the search tree can be ignored. Ignoring agents in the mid game and late game by the features described in Section 4.1 is expected to lower agent performance, if MCTS is given the time to create enough samples.

6.4 Experimental Setup

To establish a baseline, the performance of a normal MCTS agent is tested against a random AI, greedy AI and offensive AI. The random AI executes random moves, the greedy AI tries to maximise its own score in 1 action and the offensive AI tries to minimise the score of the leading agent in 1 action. The resulting win ratios of the MCTS agent will serve as a baseline for the rest of the experiments. The offensive AI is not tested in Blokus, because this AI aims to lower the leading's player score and the score of an agent never decreases during a match.

In a 4-player game a win ratio of 25% is expected when agents are evenly matched. The results in Table 3 show that MCTS outperforms the Random, Greedy and Offensive AIs by a large margin.

The selective search approach used in this thesis is based on Agent Impact, which is calculated by multiplying the different features with their associated weights for each agent. These weights indicate how important each feature is

	Random	**Greedy**	**Offensive**
Rolit	97.9% ± 0.2%	92.3% ± 0.4%	91.6% ± 0.6%
Blokus	100.0% ± 0.1%	93.2% ± 0.5%	-

Table 3: Baseline Results - MCTS Win Percentages.

to the overall Agent Impact. To determine how much each feature contributes to the impact that an agent has, an experimental setup was created based on sampling the results of MCTS agents using these features to ignore opponents.

All features described in Section 4.1 are investigated for Rolit, but the evaluation function for Blokus does not allow for a score decrease. This is due to the way the used evaluation function works. The evaluation function counts the occupied locations on the board for each player. This score cannot decrease because the blocks on a Blokus board will never disappear. Features that rely on score decreases, such as "The score decrease the best move of an opponent creates for the player" are therefore not investigated in the domain of Blokus.

To determine the weight of each feature, MCTS sampling is used. An MCTS agent that ignores opponents based on a single feature plays on average 4000 matches against three default MCTS agents, after which its performance is examined. The win ratio of the ignoring MCTS agent is then used to determine how much the used feature contributes to the Agent Impact. This is done by comparing its win ratio to the win ratio it would have had if it was evenly matched with the other agents. A higher win ratio will result in a positive feature weight, a lower win ratio will result in a negative feature weight. This results in either a positive or negative contribution to the Agent Impact value. The formula to calculate the exact weight is seen in Equation 4, where f is a feature and n is the total number of players in the game.

$$Weight_f = \frac{Winrate_f}{1/n} - 1 \tag{4}$$

For example, if an ignoring MCTS agent using feature f wins 30% of its games against three regular MCTS agents, the win ratio will be 0.55. This results in a weight of $(0.30/0.25) - 1 = 0.2$ for feature f. As different features are tested with different parameters, the parameters with the best result will be used in the combined Agent Impact. As seen in Equation 2 this weight times the score of the feature results in its contribution to the Agent Impact. Since the outputs of the proposed features are binary, either ignore or do not ignore, the feature scores are 0 or 1.

The determined feature weights can then be combined in one or more new Agent Impact MCTS agents. These are then again tested against regular MCTS agents to determine their performance. In the final Agent Impact agent, the threshold when to ignore an agent, based on the Agent Impact value, is tuned by testing the different values against each other. This tournament setup between the different MCTS agents will give an overview of agent strength from which conclusions can be drawn. This way it can be determined which features contribute the most to the Agent Impact value and if ignoring agents based on the Agent Impact value is a viable technique to improve the performance of an MCTS agent.

To help interpret the results, the average and maximum search depths of both the search tree and the actual game tree are tracked per match. The search tree depth is the depth to which the MCTS tree created its nodes. The game depth is the amount of turns these nodes represent. For example, if a branch in the MCTS tree reaches a search depth of seven and a game search depth of twelve, this means five opponents were ignored in that branch.

7 Results

The win ratios, or win percentages, of the different agents are presented with their corresponding confidence interval. The matches are played with 4 players, which would result in an expected win ratio of 25% if all agents are equally skilled. The confidence intervals are calculated for a confidence level of 95%.

7.1 Rolit

The Rolit experiments all resulted in a win ratio lower than the 25% that is expected in an equal match-up. The results in Table 4 show the results when the Counter Moves feature is used to calculate the Agent Impact value. It is important to note that as the threshold below which opponents are ignored increases, the win ratio lowers. This result is observed as well in all other Agent Impact features.

Threshold	**Win Ratio**
1	18.2% ± 1.1%
2	15.6% ± 1.0%
3	14.4% ± 1.0%
4	13.4% ± 0.9%
5	12.0% ± 0.9%
6	10.0% ± 0.8%
7	9.4% ± 0.8%
8	8.1% ± 0.7%
9	8.0% ± 0.7%
10	7.8% ± 0.7%

Table 4: Rolit Win Percentage - Ignoring Based On Counter Moves Feature.

Ignoring yourself leads to significantly better results than ignoring the agent after you in the turn order, which is an interesting result. This can be seen in Table 5, where position 0 is the player itself and position 1 the opponent who can make a move after the player. It is also shown that ignoring the opponent directly after the player gives the worst results. Ignoring the opponent directly before the player gives the best results, but this is not significantly better than ignoring yourself and still performs worse than not ignoring anyone.

Position	**Win Ratio**
0	20.5% ± 1.3%
1	17.3% ± 1.0%
2	18.9% ± 1.1%
3	21.2% ± 1.1%

Table 5: Rolit Win Percentage - Ignoring Based On Turn Ordering.

Further tables containing all results of the Rolit experiments can be found in Appendix A. All show a lowered player strength when their respective thresholds are raised, which results in ignoring more opponents.

Another example of degrading performance when more opponents are ignored is shown in the results from the agent using the Best Move Score feature to calculate the Agent Impact. To visualise these results and the underlying algorithm's search behaviour, the win ratio is visualised in Figure 14 and the average search depths are visualised in Figure 15. The win ratio lowers when the threshold higher, with a win ratio of 23.4% when the threshold is set to 1 and a win ratio of 9.8% when the threshold is set to 9.

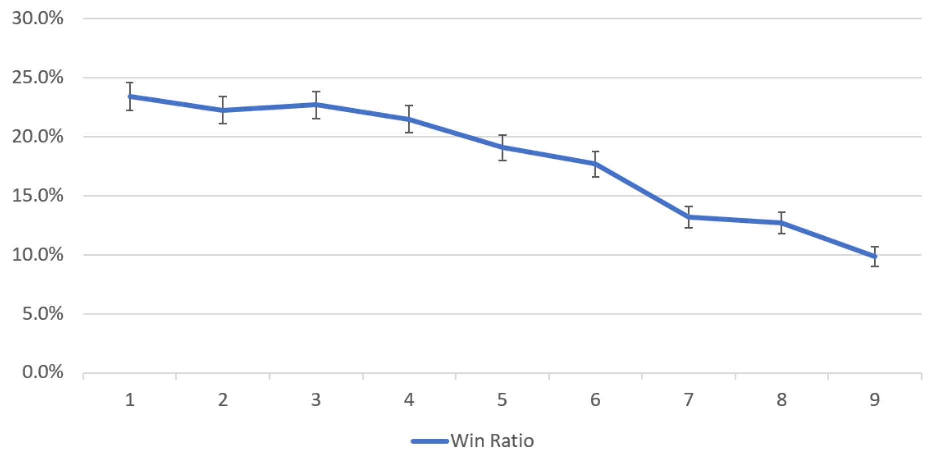

Figure 14: The win ratio of an agent ignoring opponents using the Best Move Score feature. The x-axis shows the used threshold values. This is a visualisation of Table 14.

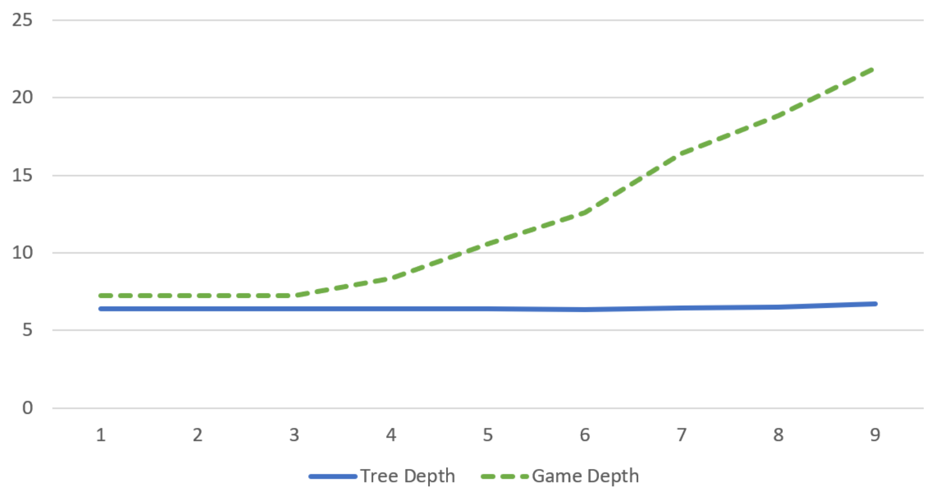

Figure 15: The average search depth of an agent ignoring opponents using the Best Move Score feature. The x-axis shows the used threshold values. This corresponds with Figure 14.

As described in Equation 4, in Section 6.4, the win ratio of the agents is used to determine the weight this feature has in the final Agent Impact. However, all features, with all parameters, ended up with a negative weight. Because of this, a final experiment for Rolit was not run, as ignoring opponents based on the investigated features is shown not to increase the performance of the agent.

7.2 Blokus

The experiments in Blokus provided more varied results than the experiments with Rolit. Some features showed a curve where first performance increased when increasing the threshold, and then performance decreased again when too many opponents were ignored. Two experiments showing this behaviour were conducted by ignoring opponents based on their overall game score, and the percentage of that game score. These results are seen in Table 6 and Table 7.

The ignoring feature based on the overall game score performs best when agents with a lower score than 10 are ignored. This results in a win ratio of 29.8%. The ignoring feature based on the overall game score percentage performs best when agents with less than 15% of the game score are ignored. This resulted in a win ratio of 28.0%. These results are significantly higher than the 25% expected in equally matched agents in a 4-player game.

Their respective weights then, as calculated via Equation 4 from Section 6.4, are 0.192 for the ignoring feature based on the overall game score and 0.12 for the ignoring feature based on the overall game score percentage. These weights are used in the final Agent Impact implementation for Blokus.

Two features from the Blokus experiments resulted in a clear increase in

Threshold	**Win Ratio**
5	27.9% ± 1.6%
10	29.8% ± 1.6%
20	26.4% ± 1.6%
30	25.1% ± 1.6%
40	23.9% ± 1.6%
50	21.2% ± 1.5%

Table 6: Blokus Win Percentage - Ignoring Based On Game Score Feature.

Threshold	**Win Ratio**
0.05	26.4% ± 1.6%
0.10	27.1% ± 1.6%
0.15	28.0% ± 1.6%
0.20	26.0% ± 1.6%
0.25	24.2% ± 1.5%
0.30	13.1% ± 1.2%
0.35	12.5% ± 1.6%

Table 7: Blokus Win Percentage - Ignoring Based On Game Score Percentage Feature.

player strength. These are the feature based on the overall game score and the feature based on the overall game score percentage. These two were combined, with weights of respectively 0.192 and 0.12, in a final Agent Impact AI. The results of this AI can be found in Table 8. One important observation to make is that, even though ignoring more opponents does seem to lower player strength, all win ratios are within the error margin away from the 25% score, which is expected if agents are evenly matched.

Threshold	**Win Ratio**
0.05	26.3% ± 1.4%
0.10	25.5% ± 1.4%
0.15	25.1% ± 1.4%
0.20	24.3% ± 1.4%
0.25	24.6% ± 1.5%
0.30	24.2% ± 1.4%

Table 8: Blokus Win Percentage - Ignoring Based On Game Score & Game Score Percentage Features.

8 Conclusion and Future Research

In this thesis the concept of Agent Impact was proposed and the feasibility of ignoring opponents in deterministic multi-player turn-based games with perfect information was investigated. Agent Impact allows a search algorithm to estimate the ability an opponent has to lower the player's chances of winning, based on multiple agent properties. The Agent Impact value can then be used to determine if an opponent is relevant to find a good action, or that this opponent can be ignored. This concept was incorporated into the MCTS algorithm to test its performance.

The first conclusion we may draw is that ignoring opponents, using Agent Impact features in MCTS, in the game of Rolit lowers an agent's chance of winning. The results indicate a strong correlation between ignoring more opponents and a lower win ratio for the MCTS agent using the feature Ignoring Based On Best Move Score. This can either be an indication that the feature is not a good enough predictor of Agent Impact, or that ignoring opponents in the game of Rolit lowers player strength overall. Looking at the performance of the rest of the features, none of which increased player performance, it is clear that ignoring opponents in the game of Rolit lowered an agent's chance of winning. This result can be explained by the volatile nature of Rolit. A single move can change a large portion of the board. Ignoring an opponent therefore removes a lot of information from the search tree. This clearly outweighs the potential benefit of the increased search depth.

The second conclusion is that ignoring opponents can increase player performance in the domain of Blokus, depending on the used Agent Impact feature and parameters. One feature that increased player strength are the features using the overall game score and the overall game score percentage. The first of these features shows that ignoring opponents in the beginning of the game results in an improved player strength, which can be explained by the lack of interaction players have in the beginning of a Blokus match. The second features indicates that players with less blocks on the gameboard have less opportunity to lower the player's chances of winning. This too can be explained by the lower interaction if an opponent controls a smaller portion of the gameboard. The fact that ignoring opponents does not necessarily lower player strength can be an indication that more specific Agent Impact features can achieve even better results.

The third conclusion we draw is that the investigated Agent Impact features are too general to greatly improve upon MCTS. MCTS inherently allocates less computing power to less promising actions. The investigated features are not a good enough indication of an opponent's ability to lower a player's chance of winning to warrant the removal of a node from the MCTS tree. Agent Impact features that are better indicators could improve performance further, but these features would require more specific domain knowledge. The results showed that the features incorporated in the final Agent Impact AI performed on par with regular MCTS. This implies that the Agent Impact AI did an equally good job of predicting if an opponent could be ignored as that the regular MCTS agents did.

With respect to the introduction of Agent Impact we conclude that it has potential, but it performance depends heavily on the features, domain and used search algorithm. A more classic tree search algorithm could potentially benefit more from the incorporation of Agent Impact than MCTS.

For future research the following approaches are suggested:

1. Incorporate Agent Impact into a more classical tree search algorithm, such as max^n, paranoid or BRS;
2. Tune the weights of Agent Impact features via an automated approach, such as a genetic algorithm, as this would allow for more precise tuning;
3. Introduce a version of Agent Impact that can mutate feature weights during different moments throughout a game;
4. Incorporate a more domain-specific evaluation function per domain; and
5. Add new Agent Impact features, several suggestions are listed below.

Agent Impact features can be specified for any domain and it is expected that features which use more domain knowledge have more potential to achieve a larger improvement in player strength. The set of such features is large and very dependant on the domain. When Agent Impact is implemented in a specific domain it is suggested such features, relying on very specific game rules, are also researched.

The suggested more broadly applicable Agent Impact features to investigate in future research are:

1. The distance on the game board between pieces of the opponent and the player;
2. The number of possible opponent move locations that are shared with the player. This considers how many blocking and interfering moves an opponent has; and
3. The reduction in future available moves for the player by an opponent's move.

9 Discussion

The Agent Impact features introduced in this thesis are easily applicable to other domains. The domain-agnostic features work without any adaptation, and the domain-specific features only require a domain-specific heuristic evaluation function. It is likely that features using more domain-specific knowledge will outperform the features discussed in this thesis. Creating very domain-specific features is an interesting subject for future research. The approach in this thesis was decided upon to have an initial indication of the Agent Impact approach is feasible and to keep a wide applicability of the features.

As stated in this thesis, the weights used in Equation 2 can be positive or negative values, or zero when a feature does not contribute to the Agent Impact. This approach allows features that potentially lower the Agent Impact to be considered as well. This design, where the weight of each feature can be tuned, resembles a perceptron, which is a vital part of neural networks, which have become very popular. Perceptrons in neural networks allow for fine grained tuning when the appropriate automated tools are used. This approach could be applied to the concept of Agent Impact to improve the results.

The research done in this thesis was performed with a node limited MCTS implementation. This MCTS implementation was chosen to improve the reproducibility of the research. Time limited algorithms are dependent on the exact implementation, used programming language, background processes on the computer and the hardware on which it is run. This means a time limited MCTS implementation will perform differently on different hardware, even if the same time limit is used. Therefore, the research done in this thesis used a node limited MCTS implementation and provided a reference table to a time limited version, such that this research remains comparable to related works that use time limited implementations. To improve the reproducibility even further, the entire framework used in this thesis is made available as open source software at https://gitlab.com/jobhh/.

References

[1] F. Hsu. Ibm's deep blue chess grandmaster chips. *IEEE Micro*, 19(2):70–81, 1999.

[2] D. Silver et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[3] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo Tree Search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[4] C. Browne et al. A survey of Monte Carlo Tree Search methods. *Intelligence and AI*, 4(1):1 – 49, 2012.

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

[6] J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

[7] M.P.D. Schadd. *Selective search in games of different complexity.* Maastricht University, 2011.

[8] J.A.M. Nijssen. *Monte-Carlo tree search for multi-player games.* PhD thesis, Maastricht University, 2013.

[9] J.A.M. Nijssen and M.H.M. Winands. Playout search for monte-carlo tree search in multi-player games. In *Advances in Computer Games*, pages 72–83. Springer, 2011.

[10] C. Luckhart and K.B. Irani. An algorithmic solution of n-person games. In *AAAI*, volume 86, pages 158–162, 1986.

[11] N.R. Sturtevant and R.E. Korf. On pruning techniques for multi-player games. *AAAI/IAAI*, 49:201–207, 2000.

[12] I. Zuckerman, A. Felner, and S. Kraus. Mixing search strategies for multi-player games. In *IJCAI*, volume 9, pages 646–652, 2009.

[13] J. Schaeffer et al. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[14] S. Chinchalkar. An upper bound for the number of reachable positions. *ICGA Journal*, 19(3):181–183, 1996.

[15] C.E. Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[16] M. Wolf. An intelligent artificial player for the game of risk. *Unpublished doctoral dissertation). TU Darmstadt, Knowledge Engineering Group, Darmstadt Germany. http://www. ke. tu-darmstadt. de/bibtex/topics/single/33*, 2005.

[17] D. Michie. Game-playing and game-learning automata. In L. Fox, editor, *Advances in Programming and Non-numerical Computation*, pages 183 – 200. Pergamon, Oxford, 1966.

[18] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[19] C.E. Shannon. Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer, 1988.

[20] M.P.D. Schadd and M.H.M. Winands. Best reply search for multiplayer games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):57–66, 2011.

[21] N. Sturtevant and M. Bowling. Robust game play against unknown opponents. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 713–719. ACM, 2006.

[22] N. Sturtevant, M. Zinkevich, and M. Bowling. Prob-maxˆ n: Playing n-player games with opponent models. In *AAAI*, volume 6, pages 1057–1063, 2006.

[23] M.P.D. Schadd and M. Lanctot. Improving best-reply search. In *Computers and Games: 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*, volume 8427, page 125. Springer, 2014.

[24] P. Whittle. Multi-armed bandits and the gittins index. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 143–149, 1980.

[25] M. Buro. The evolution of strong othello programs. In *Entertainment Computing*, pages 81–88. Springer, 2003.

[26] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[27] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

10 Appendix A

Appendix A contains the Rolit results tables.

Threshold	**Win Ratio**
2	22.9% ± 1.2%
3	19.9% ± 1.1%
4	18.6% ± 1.1%
5	16.7% ± 1.0%
6	15.0% ± 1.0%
7	14.3% ± 1.0%
8	13.0% ± 0.9%
9	12.0% ± 0.9%
10	10.4% ± 0.8%

Table 9: Rolit Win Percentage - Ignoring Based On Number Of Available Moves.

Threshold	**Win Ratio**
1	22.1% ± 1.1%
2	22.2% ± 1.1%
3	21.5% ± 1.1%
4	17.0% ± 1.0%
5	12.2% ± 0.9%
6	8.3% ± 0.8%
7	7.8% ± 0.7%
8	7.2% ± 0.7%
9	7.2% ± 0.7%
10	7.3% ± 0.7%

Table 10: Rolit Win Percentage - Ignoring Based On Average Move Score.

Threshold	**Win Ratio**
1	23.3% ± 1.2%
2	21.7% ± 1.1%
3	20.2% ± 1.1%
4	18.1% ± 1.1%
5	17.4% ± 1.0%
6	15.8% ± 1.0%
7	14.4% ± 1.0%
8	14.0% ± 0.9%
9	11.1% ± 0.9%
10	10.4% ± 0.8%

Table 11: Rolit Win Percentage - Ignoring Based On Overall Game Score.

Threshold	**Win Ratio**
0.01	24.1% ± 1.2%
0.02	23.3% ± 1.2%
0.05	21.4% ± 1.1%
0.10	19.8% ± 1.1%
0.15	17.1% ± 1.0%
0.20	15.6% ± 1.0%
0.25	12.5% ± 0.9%
0.30	9.9% ± 0.8%
0.40	7.6% ± 0.7%
0.50	6.8% ± 0.7%

Table 12: Rolit Win Percentage - Ignoring Based On Overall Game Score Percentage.

Threshold	**Win Ratio**
1	19.2% ± 1.1%
2	14.9% ± 1.0%
3	12.7% ± 0.9%
4	10.1% ± 0.8%
5	8.3% ± 0.7%
6	7.8% ± 0.7%
7	6.2% ± 0.7%
8	6.6% ± 0.7%
9	6.5% ± 0.7%

Table 13: Rolit Win Percentage - Ignoring Based On Best Counter Move Score.

Threshold	**Win Ratio**
1	23.4% ± 1.2%
2	22.2% ± 1.1%
3	22.7% ± 1.2%
4	21.5% ± 1.1%
5	19.1% ± 1.1%
6	17.7% ± 1.0%
7	13.2% ± 0.9%
8	12.7% ± 0.9%
9	9.8% ± 0.8%

Table 14: Rolit Win Percentage - Ignoring Based On Best Move Score.

Threshold	**Win Ratio**
0.05	24.6% ± 0.9%
0.10	21.5% ± 0.9%
0.15	18.7% ± 1.0%
0.20	17.4% ± 1.0%
0.25	17.0% ± 1.2%
0.30	13.4% ± 1.0%
0.35	10.6% ± 0.9%

Table 15: Rolit Win Percentage - Ignoring Based On Available Moves Amount Percentage.

11 Appendix B

Appendix B contains the Blokus results tables.

Position	**Win Ratio**
0	25.9% ± 1.6%
1	24.6% ± 1.5%
2	26.7% ± 1.7%
3	27.1% ± 1.8%

Table 16: Blokus Win Percentage - Ignoring Based On Turn Ordering.

Threshold	**Win Ratio**
25	21.1% ± 1.5%
50	16.9% ± 1.4%
75	15.8% ± 1.3%
100	15.2% ± 1.3%
125	14.6% ± 1.3%
150	14.2% ± 1.3%

Table 17: Blokus Win Percentage - Ignoring Based On Number Of Available Moves.

Threshold	**Win Ratio**
2.5	23.8% ± 1.6%
3.0	19.0% ± 1.5%
3.5	15.6% ± 1.4%
4.0	16.4% ± 1.4%
4.5	12.3% ± 1.3%

Table 18: Blokus Win Percentage - Ignoring Based On Average Move Score.

Threshold	**Win Ratio**
0	25.5% ± 1.7%
1	27.5% ± 1.7%
2	26.9% ± 1.6%
3	26.9% ± 1.6%
4	25.1% ± 1.5%
5	22.0% ± 1.4%

Table 19: Blokus Win Percentage - Ignoring Based On Best Move Score.

Threshold	**Win Ratio**
0.05	27.1% ± 1.6%
0.10	26.8% ± 1.6%
0.15	26.6% ± 1.6%
0.20	26.2% ± 1.6%
0.25	26.5% ± 1.6%
0.30	23.1% ± 1.5%
0.35	19.6% ± 1.4%

Table 20: Blokus Win Percentage - Ignoring Based On Available Moves Amount Percentage.