Universiteit Leiden
The Netherlands

Opleiding Informatica

Exploring Optimal Strategies for Dice of Doom

Perri van den Berg

Supervisors:
Rudy van Vliet & Jeannette de Graaf

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

25/08/2023

Abstract

Dice of Doom is a strategy game similar to Dice Wars and Risk. Since it is a game of chance, finding an optimal strategy can be challenging. In this thesis, we explore optimal strategies for Dice of Doom. The research is divided into two parts. In the first part, different strategies are compared to a greedy strategy. The strategies examined are the random strategy, the return zero strategy, the Monte Carlo strategy, the Monte Carlo Tree Search strategy, and the expansion strategy. These strategies are compared to the greedy strategy using simulations. In the second part, the focus is on determining the probability of winning when both players play optimally. The difficulty is that the game contains cycles, which breaks the brute force method. To solve this issue, policy iteration is applied. This approach resulted in a very effective strategy, which was then compared to the greedy strategy.

In conclusion, this research found a strategy for the game Dice of Doom that is close to optimal. One observation is that the greedy strategy outperforms any of the five compared strategies when there is a maximum number of five dice allowed. However, when only two dice are allowed per tile, the other strategies outperform the greedy strategy. The close-to-optimal strategy was successful against the greedy strategy in both cases. Regarding the probability of winning, we found that the starting player has a slight advantage over the other player. The game becomes more 'fair' as the number of dice allowed on a tile increases.

Contents

1 Introduction

Table top games have always been enjoyed by many people. It stimulates the brain to come up with better and better strategies to compete against other players. One of those games is Dice of Doom. This game, as explained in the book 'Land of Lisp' [Bar11], is very similar to the classic game Risk. The goal of the game is to own the highest number of tiles on the map. To do this, players have to make calculated decisions and take into account the odds of winning. Dice of Doom introduces gameplay features different from Risk that make the game easier to learn, but hard to master.

1.1 Thesis Goal

This thesis explores the optimal strategies for the game Dice of Doom, consisting of two parts. In the first part, a comparison is made between a greedy strategy and other strategies, to answer the question: "How can a computer beat a human player who uses a greedy algorithm to play the game 'Dice of Doom'?" In the second part, the research focuses on the question: "What is the probability that a player wins, given a game state in 'Dice of Doom' when both players play optimally?" With the outcomes of this research, we aim to develop a strategy that is optimal or close to optimal.

1.2 Related Work

For the scope of this research, we will be looking at games that are similar to Dice of Doom, and research in those fields. There have been many studies to explore the battle outcomes, strategies, and decision-making processes in games similar to the classic game Risk. In this section a short overview is given about relevant research and key findings of the research.
In [Geo04], the analysis focuses on examining the factors that influence the probabilities of winning battles in the game Risk. The paper provides insights into strategic decision-making by considering various elements, like the troop deployment, dice outcomes, and the impact of different regions. By analyzing battle outcomes, this study shows the dynamics of the game and offers guidance for players aiming to improve their strategies.
In [Blo20], an application of the AlphaZero algorithm to the game of Risk is explored. The authors developed an AI agent capable of learning to play Risk through self-play. By using deep neural networks and Monte Carlo Tree Search, the AlphaZero agent achieved a remarkable performance, surpassing human players. This research shows the potential of using AI techniques to enhance game play and provide challenging opponents for players.
Furthermore, the use of Monte Carlo methods in board games, including Risk, has been extensively studied. The paper [EMAS10] provides an overview of how Monte Carlo (MC) simulations can be used to estimate winning probabilities and evaluate strategies in various games, including Risk. MC simulations involve running numerous random simulations of game outcomes to approximate winning probabilities and find optimal moves. This approach enables players to make better moves based on statistical analysis, contributing to a deeper understanding of the game dynamics.
There was also an analysis of different agents in the game of Risk, as described in [Dro18]. This research compares the performance of four agents, being Knapsack, Angry, Evaluate, and Random, in a two-player version of Risk without neutral factions. The thesis extensively examines the characteristics and strengths of each agent, considering factors such as aggressiveness, planning moves in advance, and the behavior of greedy agents. Based on the analysis, the Angry agent was

the most successful. This study provides insights into agent-based game play and highlights the significance of agent characteristics in determining success in Risk.
In summary, previous research has examined various aspects of games like Risk, including battle outcomes, the application of AI algorithms such as AlphaZero, the use of Monte Carlo simulations, and the comparison of different agents. These studies contribute to a deeper understanding of the dynamics, strategies, and the potential for AI agents. The insights gained from these works, including strategies and algorithms, were used in our research on the game Dice of Doom.
There was no research found that directly involves the game Dice of Doom.

1.3 Thesis Overview

Section 2 of this thesis contains the in-depth rule set of the game Dice of Doom. In Section 3 the methodology is described. In this section the implementation of the game is explained, as well as the different strategies, how to obtain the best winning probability, and how to get a close-to-optimal strategy. In Section 4 the results of this research can be found for both the strategies and the probability of winning. In Section 5 the conclusion and further research are described.

2 Rules of Dice of Doom

Dice of Doom is a simple game, but some gameplay features within the game are more complex. In this section we explain how the game works and what the rules of the game are. Note that some rules listed below are slightly different from the rules as explained in [Bar11]. In particular, we added some limitations and exceptions to make the research more concrete.

2.1 The Basic Rules

We start with a list of basic rules of Dice of Doom. Exceptions are clarified in Section 2.2.

Setting up the Game

- There are two players, playing against each other.
- There is a hexagonal grid of 5×5 hexagonal tiles. The tiles are placed in such manner that the complete grid is diamond shaped.
- Each tile on the hexagonal grid is owned by one player.
- On each tile, there is a minimum of one 6-sided die belonging to the player who owns the tile. Each die has an equal probability of rolling numbers 1 through 6. There is an upper bound of five dice allowed on a tile, and any additional die on a tile will be removed from the game.
- While setting up the game, the tiles on the board are randomly distributed among the players. This means that the ownership of the tiles is determined randomly, and the number of dice placed on each tile is also chosen randomly. Of course, the number of dice does not exceed the upper bound we just mentioned. It's worth noting that due to this randomness, one player might end up with a significantly greater number of tiles than the other player.
- Although it is not explicitly stated in [Bar11], in our implementation of Dice of Doom, the player with ID 0 (represented by the color red) starts the game. However, in the 2-player version of Dice of Doom, it doesn't matter which player starts, as the ownership of the tiles is randomly assigned.

The Turn of a Player

- A turn consists of a series of attacks followed by placing reinforcements before ending the turn. The player is required to make at least one attack before placing reinforcements. Both attacking and placing down reinforcements are regarded as moves.
- To initiate an attack, the player must select a tile adjacent to an enemy tile, and this chosen tile must have a minimum of two dice on it. During an attack, the dice stacks on both tiles are rolled independently. The sums of the rolled dice for both players are then compared. If the sum of the attacker's dice is greater than the sum of the defender's dice, the attacker wins the battle. If the sum of the attacker's dice is less than or equal to the sum of the defender's dice, the attacking player loses the battle.

- If the attacker wins, the defending player's dice are removed from that tile and the attacking player now owns that tile. All dice but one are moved to the newly owned tile.
- If the defender wins, the attacker removes all dice but one from the attacking tile. That is, one die remains, and no dice of the defending player are removed.
- After the attack (regardless of the outcome of the attack), the player can choose to make another attack, or place reinforcements and end his turn (the skip option).
- After the completion of a turn, the next player takes the turn.

Reinforcements

- Before a player can end his turn, he has to place down reinforcements on his tiles. These additional dice can be placed down in any manner, but any dice exceeding the maximum of 5 dice on a tile are removed. When all dice are placed down, the turn ends.
- The number of reinforcement dice that the player must place down is equal to the number of tiles in the player's largest contiguous territory.

How to Win the Game?

- If a player cannot perform a single attack during his turn, the game ends. This can be the case either if the player does not own any tiles, if the player owns all the tiles, or if all of the tiles that are adjacent to enemy tiles have only one die.
- When the game ends, the player owning the most tiles wins.
- If the numbers of tiles owned are equal, the game results in a tie.

2.2 Exceptions

For the game Dice of Doom, there are not many exceptions, as the game is quite straightforward. However, there is one tricky situation that can occur, which is the possibility for a game to continue infinitely. To avoid this issue, a new rule is introduced. If a game of Dice of Doom exceeds 100 turns it results in a tie.

2.3 Game Example

This section demonstrates a simple example using the rules of Dice of Doom. The game is played on a 2×2 board with a limit of two players and five dice per tile. The red player starts the game. The random initialization can be seen in Figure 1, where the numbers on the tiles represent the numbers of dice.

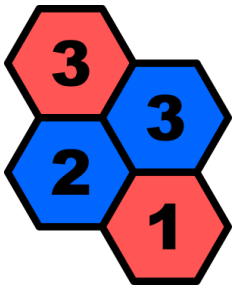

Figure 1: Random board initialization. The red player starts.

The red player begins his turn by attacking the blue player, as seen in Figure 2. Both players roll the dice on his respective tiles. The red player rolls three dice, while the blue player rolls two dice. The sum of the red player's dice is 8, whereas the sum of the blue player's dice is 7. Therefore, the red player wins the battle with a higher total. As shown in Figure 3, the red player can now take over the blue tile, leaving behind one die.

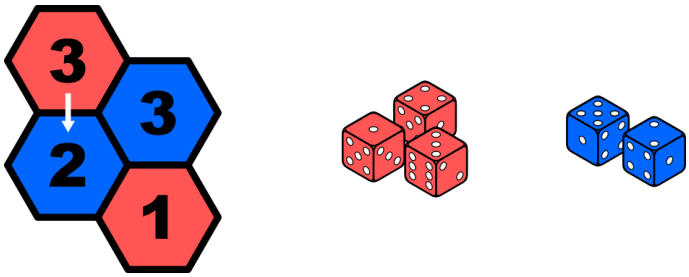

Figure 2: The red player attacks the blue player indicated by the white arrow. The outcome of the dice roll is displayed beside the board.

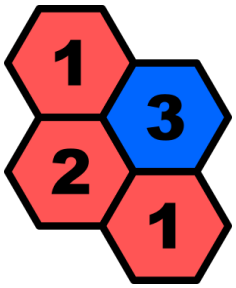

Figure 3: The red player has won the battle and takes over the blue tile, leaving one die behind.

After winning the battle, the red player can attack the other blue tile, but chooses the skip option (end his turn). Now, reinforcements are added to the red player's tiles. The number of dice added is

equal to the total number of tiles in the largest contiguous territory owned by the red player. In this case, it is equal to 3. As depicted in Figure 4, the dice are added to the tiles.

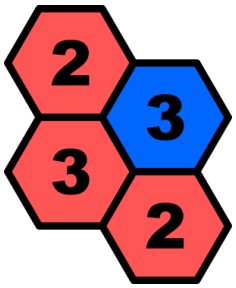

Figure 4: The red player ends his turn and gains reinforcements. A total of three reinforcements are added to the red tiles.

Now it is the blue player's turn. The blue player attacks one of the red player's tiles using the three dice on his tile, against the two dice of the red player. Both players roll their dice. As shown in Figure 5, the blue player obtains a total of 9, and the red player obtains a total of 9 as well. Since the totals are equal, the blue player loses the battle.

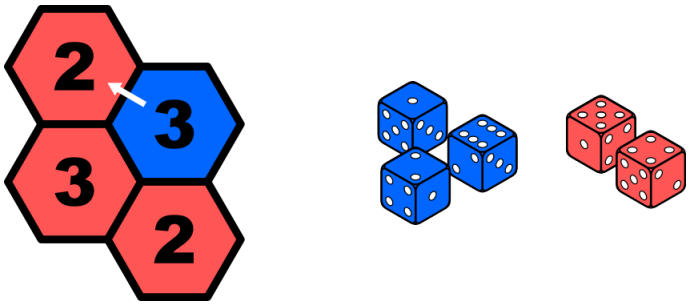

Figure 5: The blue player attacks one of the red player's tiles. The dice show the outcome of the battle.

Since the blue player lost the battle, the dice on the attacking tile are reduced to 1, as seen in Figure 6. A player needs at least two dice on a tile in order to initiate an attack. This means that he is unable to attack. As the blue player has already attacked during his turn, the player can end his turn. In Figure 7, reinforcements are added to the blue tile, and it is now the red player's turn.

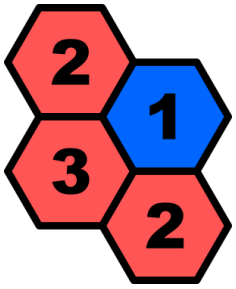

Figure 6: The blue player loses two dice.

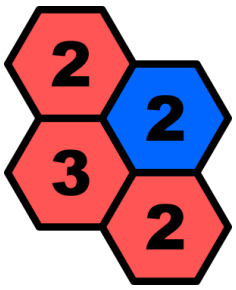

Figure 7: The blue player ends his turn, and reinforcements are added. Now it is the red player's turn.

The red player attacks the last remaining tile of the blue player, as shown in Figure 8. The red player rolls three dice and obtains a total of 9, while the blue player rolls two dice and obtains a total of 6. With a higher total, the red player wins. Now all dice except for one die move to the newly obtained tile.

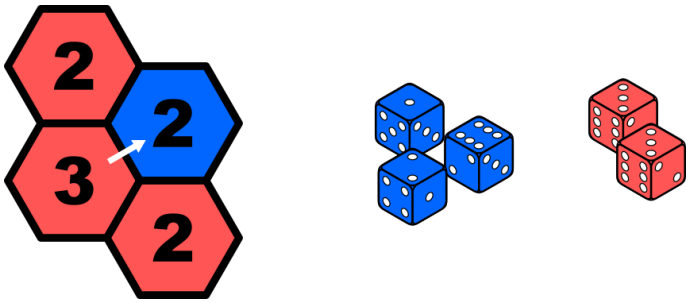

Figure 8: The red player attacks the last tile of the blue player. The result of the thrown dice are shown.

Since the red player can no longer attack, he is forced to pass the skip option, as seen in Figure 9. He can place down a total of 4 reinforcements.
The blue player has no remaining tiles on the board, meaning that the blue player can no longer attack. As a result, the game ends. The tiles of both players are counted and compared. In Figure 10, the outcome of the game is shown.

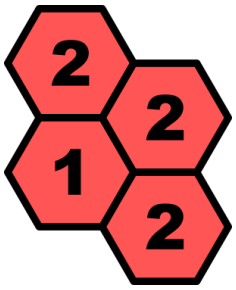

Figure 9: The red player can add a total of 4 reinforcements.

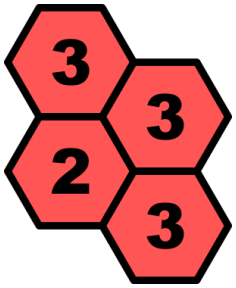

Figure 10: The red player has won the game.

2.4 Variations

The game Dice of Doom can also be played with a few variations. These variations can change the way the game is played, and influence the best strategy to play. The different variations mainly deal with the initialization, variables and limits, and reinforcements.
For the initialization in the current game, the ownership and dice on a tile are set randomly. However, this can be varied. One of the variations that we will examine in this thesis, is the random initialization where the numbers of tiles owned by players are (nearly) the same. This way, the games will be more interesting, as there is a smaller probability that the random initialization will result in a nearly finished game. We also vary in the size of the board.
There are also some different variations that will not be considered in this thesis. This includes the shape of the board. This for example can be in a not diamond-shaped lay-out, like a triangle or even the shape of a country. One other variation that can be added, is to merge certain hexagons together, meaning that tiles may have more neighbors than the maximum of 6 neighbors. This way the game looks more like the games Risk and Dice Wars.
The game Dice of Doom has a lot of variables and limits that can be changed. One of the variations that can be created using these limits, is a game with more than two players. There is also an option to change the shape of the dice. All this however, will not be looked at in this thesis. We will, however, be looking at the option to add fewer dice on the board.
Finally the game can be varied in the way that the reinforcements are calculated and distributed. As described in [Bar11], one could also calculate the number of dice that are obtained by attacking, minus one. This means that you get no reinforcements if you have no winning attacks. For example, if the red player has already obtained three dice, and wins an attack in which he obtains an additional two, the red player would have a total of five dice. If the red player now ends his turn, he would get four reinforcements to place down. This will not be looked at in this thesis.
As for the distribution of reinforcements, there are two more variations for the game. The first variation is deterministic. In this version the dice are placed in a deterministic order from left to right and from the top to the bottom. Every tile gets at most one die added to it. The other version is the weighted random distribution. In this version, the dice are randomly distributed over the tiles, but the tiles with more dice, have a bigger probability to get an additional dice.

3 Methods

In this section of the thesis, we discuss our implementation of the game Dice of Doom. The research requires the development of three distinct components: the base game, strategy simulations, and an algorithm to calculate the probability of winning. All of the code is written in the programming language C++.

3.1 Implementation of the Game

To implement the game Dice of Doom, we make use of object-oriented programming (OOP) due to the ease of separating the game's components and the minimal need for shared information.
The game consist of the following components:

- The game board
- The game loop
- The different strategies
- The variables and upper bounds

A brief explanation is provided for each component below, along with an explanation of how these components interact with each other.

3.1.1 The Board of the Game

The board component keeps track of the length and width of the board, as well as who owns which tile and how many dice each tile has.
Besides storing information, the board component also has a few functionalities that are used by the other components. These functionalities mainly consist of the moves a player can make, adding reinforcements, and calculating the winner of the game.
There are also a lot of functionalities to make the game easier to understand for a human player. For example, the ability to print the current board lay-out and the moves the current player can execute.
Other functionalities are not necessarily needed for the base game, but are used by different strategies later on in the research. These are functionalities like the probability a move has of succeeding.

3.1.2 The Game Loop

In order to play the game, a game loop is required. This component is responsible for determining the moves of a player using a certain strategy, passing the turn to the next player, and ending the game when a player cannot perform any actions.
The game loop component uses the board component as a foundation to build on, as the game requires a board to function. Furthermore, the board is used to execute the players' moves and determine the winner of a game.
This component also keeps track of the current player's turn and the current turn that the players are playing. Situations may arise where the game enters an infinite loop, as explained in Section 2.2. To prevent these infinite games from occurring, the game results in a tie after one hundred turns.

3.1.3 The Different Strategies

In order to play the game, we also need a way to execute different strategies. That is why there is a strategy component. This component is used during the game loop to calculate the moves that the player should execute according to the chosen strategy.
For the base game, there are only two different strategies: the human strategy and the greedy strategy. More strategies can easily be added for the first research question, as we see in Section 3.2.

The Human Strategy

The human strategy consists of a human player who uses his insight to make a move. This strategy requires the game to be visualized and shown to an actual human. However, this strategy is not used in this thesis due to the challenges of conducting a significant number of experiments involving human participation.

The Greedy Strategy

The other strategy is the greedy strategy. This strategy uses a greedy approach to calculate the best move given a game state. The approach only looks one step into the future. To determine the best move, the strategy examines all possible moves the player can make on the current board.
In order to find the best score for each move, the strategy looks at the following variables: the probability p of achieving victory in an attack, the number of dice x on the board when winning an attack, and the number of dice y on the board after losing an attack. Using this information, a score can be calculated for every move:

$$score = x \times p + y \times (1 - p)$$

The numbers of dice x and y also include the amount of reinforcements that would be gained by passing a skip move after the attack. This way, the growth of the largest contiguous area is also taken into account and rewarded.
After the first move has been made, there is also an option to pass the turn to the next player. The score of this move is equal to the number of dice resulting from adding reinforcements to the current board. This score is used as a lower bound, as executing 'bad' moves makes it easier for the opponent to counter-attack.
The algorithm prioritizes attacking over skipping, as having a larger area at the end of the turn results in more reinforcements. If a move has the same score as the lower bound of the skip option, the turn is passed. Since with the skip option no dice can be lost.
Once all the scores have been calculated, the strategy executes the move with the highest score. A pseudocode description of the greedy strategy is given in Appendix A.

3.1.4 The Variables and Limits

In this research, certain variables and limits are used. As explained in Section 2, the game Dice of Doom has predefined variables that are used in our research. These variables include the 5×5 board, the maximum of five dice on a tile, and the requirement of two dice to initiate an attack.
In addition to these limits, a maximum number of turns is required. As described in Section 2.2, there is a potential for the game to continue indefinitely. To ensure computational feasibility, we

added a limit of 100 turns to the implementation. When the 100th turn is finished and there is no winner, the game results in a tie.

3.1.5 The Probability of Winning a Battle

In order to calculate the probability of winning a battle, we first need to look at all the possible battles that can happen. As explained in Section 2.1, a player can only attack from tiles that have neighboring enemy tiles and have at least two dice themselves. In general we know that the attacker can attack with a dice, and the defender can defend with b dice, with $a \geq 2$, $b \geq 1$, and a, b are integers.
Once the combinations of a and b involved in possible attacks are identified, we need to calculate the probability of winning these attacks. The condition for winning is that the sum of a dice must be greater than the sum of b dice. This can be expressed as the following probability:

$$P(Sum(a) > Sum(b))$$

Here, the formula $Sum(z)$ represents the sum obtained by rolling z dice randomly and summing the numbers shown on the top faces of the dice. The notation $P(X)$ denotes the probability of event X to occur.

The Math behind the Winning Probability

Given the probability formula, we can express it in a different form. This involves summing the probabilities where $Sum(a)$ is equal to a variable i, and $Sum(b)$ is less than i, for all i ranging from a to $6a$.

$$\sum_{i=a}^{6a} P(Sum(a) = i) \times P(Sum(b) < i)$$

The following recurrence relation is used to calculate the probability of $P(Sum(a) = i)$:

$$P(\text{Sum}(a) = i) = \begin{cases} 1 & \text{if } a = 0 \text{ and } i = 0 \\ 0 & \text{if } i < a \text{ or } i > 6a \\ \sum_{d=1}^{6} \left(\frac{P(\text{Sum}(a-1)=i-d)}{6} \right) & \text{if } a \geq 1 \text{ and } i \geq a \end{cases}$$

There are three base cases in this recurrence relation. The first two base cases handle the scenarios where there are zero or one dice. The third base case checks if the number i cannot be obtained with a dice. In the recursive part of the formula, the sum is calculated by throwing one die resulting in the value d. Then we evaluate the probability of obtaining the value $i - d$ with $a - i$ dice. This is done for all six possible values d. Finally we average over these values. One additional benefit of the third base case is that it also deals with the negative number i and the case where either a or i is zero, both of which can occur in the recursive step.
Using this recurrence relation, we can also calculate the probability $P(Sum(b) < i)$ as follows:

$$P(Sum(b) < i) = \sum_{j=b}^{i-1} P(Sum(b) = j)$$

The Implementation

At the start of the program, all the probabilities are calculated and stored in a table. This table is used for the required probabilities in different strategies. The table is indexed by the numbers of dice, denoted as a and b, both with a limit corresponding to the maximum number of dice allowed on a tile.

To efficiently compute the probabilities, we employ the bottom-up dynamic programming approach with the recurrence relation $P(\text{Sum}(a) = i)$. We use a 2D array of size $(a + 1) \times (i + 1)$ and iteratively calculate the probability for each entry of a row based on the values computed in the previous row. By evaluating the value at index (a, i), we obtain the probability $P(\text{Sum}(a) = i)$, which is then returned as the result.

The remaining functions used to calculate the probability $P(Sum(a) > Sum(b))$ follow the formulas mentioned earlier. The resulting probabilities are given in Table 1, where the attacking tile contains a dice and the defending tile contains b dice, with a maximum of five dice allowed on a tile. The row corresponding to attacking with one die is included to display the odds, but it is not a valid scenario in the game since a player needs at least two dice to start an attack.

a \ b	1	2	3	4	5
1	0.416667	0.0925926	0.0115741	0.000771605	2.14E-05
2	0.837963	0.443673	0.152006	0.0358796	0.00610497
3	0.972994	0.778549	0.453575	0.191701	0.0607127
4	0.997299	0.939236	0.74283	0.459528	0.220442
5	0.99985	0.98794	0.909347	0.718078	0.463654

Table 1: The probability $P(Sum(a) > Sum(b))$ when attacking with a dice and defending with b dice.

3.2 Strategies

As described in Section 3.1.3, we have defined a greedy strategy for the game Dice of Doom. For the first research question, we need to find a strategy that can beat the greedy strategy. In order to achieve this, we examine multiple strategies: the random strategy, the return zero strategy, Monte Carlo strategy, Monte Carlo Tree Search strategy, and the expansion strategy. These strategies are described below.
Note that each strategy has a random component. Once a move has been selected, executing the move involves rolling the dice. Depending on the outcome, the move may result in either a winning move (gaining a tile) or a losing move (losing dice). There is also the possibility to pass your turn after making one or more attacks. It is good to note that the newly obtained tiles are also taken into consideration for each of the strategies when deciding which move to make next. Furthermore, the strategies have no influence on how the reinforcements are placed.
The reason for using the greedy algorithm in our research is the small computational power required. Since a greedy algorithm looks only one step ahead, it requires fewer calculations compared to algorithms that consider multiple steps or the full game tree. It is also more user-friendly, as with some effort the greedy strategy can be applied by a human player. By conducting experiments, we aim to determine the best strategy to use for playing the game.

3.2.1 Baseline

In this research, we use the greedy strategy as our baseline. When we let the greedy strategy play against itself, we can observe if any apparent patterns exist. We may use such patterns to gain further insight into the results obtained when comparing the other strategies against the greedy strategy.

3.2.2 Random Strategy

For the random strategy, a list is constructed containing all possible moves a player can make. If the player has already made a move, which implies that he can decide to end his turn, then a skip option is added to the list with an equal probability of being selected as any other move. Once the list has been constructed, this strategy randomly chooses a move from the list, which is then executed on the board.
After a move is made, the strategy revisits all possible moves and reconstructs the list. This process continues until either the skip option is chosen or there are no more available moves. In the latter case, the turn is automatically passed, and the next player takes his turn.
The reason for using the random strategy is to measure the trade-off between exploration and exploitation factors. While the greedy strategy calculates a local optimum (exploitation), the random strategy makes moves at random (exploration). Additionally, the random strategy serves as a baseline for comparing against the greedy algorithm. If the greedy strategy significantly outperforms the random strategy, it implies that the greedy algorithm can be considered a valid strategy, as it represents an improvement over random moves.

3.2.3 Return Zero Strategy

The return zero strategy is a straightforward approach that selects the first available move. This move is determined in a deterministic manner by iterating over the tiles from left to right and top to bottom. If a valid move is found, this strategy selects and returns that move. If there are no more possible moves on the board, and at least one move has already been executed, the strategy skips the turn.
The idea behind this algorithm is related to the distribution of reinforcements. In one of the implementations, reinforcements are distributed deterministically from left to right and top to bottom. This aligns with the order in which the possible moves are constructed. On might expect that executing moves on tiles more likely to receive reinforcements results in an increase in the number of reinforcements obtained by the player.

3.2.4 Monte Carlo Strategy

In our implementation of the Monte Carlo (MC) strategy, we follow a five-step process similar to the one described in [Ade14], but with a slight difference. Instead of running a single instance of Monte Carlo on a given board, we run one Monte Carlo instances for each of the possible moves n, as outlined below:

1. Pseudo-Population Selection: We start by selecting a pseudo-population or model that represents the true population of interest. In our case, the model consists of the 2 boards obtained after executing move n on the root board, or 1 board if move n is the skip move.

2. Random Board Selection: From the chosen pseudo-population, we randomly select one of the boards by executing move n on the root board, which results in either a losing or winning move.

3. Statistical Value Calculation: Using the obtained board, we simulate the game using the random strategy until it reaches a terminal state (win, loss, or tie). We only consider the outcomes of winning simulations as our main focus is on identifying moves that lead to successful outcomes and maximize the probabilities of winning. An example of this step is illustrated in Figure 11, where "move 1" results in a losing board state. However, despite the initial setback, the simulation leads to a win, which is then added to the statistical value.

4. Iterative Trials: We repeat steps 2 and 3 for a specified number of trials (N = 100) to gather a collection of statistics for move n.

5. Calculation of Statistic: Using the collected statistics from the N trials, we calculate the value for the statistic of interest for move n.

After calculating scores for every move using the Monte Carlo algorithm, we compare the scores and execute the move with the highest score. This process continues until the skip option is selected or there are no more possible moves.
The advantages of the Monte Carlo algorithm, as explained in [Ade14], are its straightforwardness and its ability to approximate the optimal solution. We chose this algorithm because of these reasons.

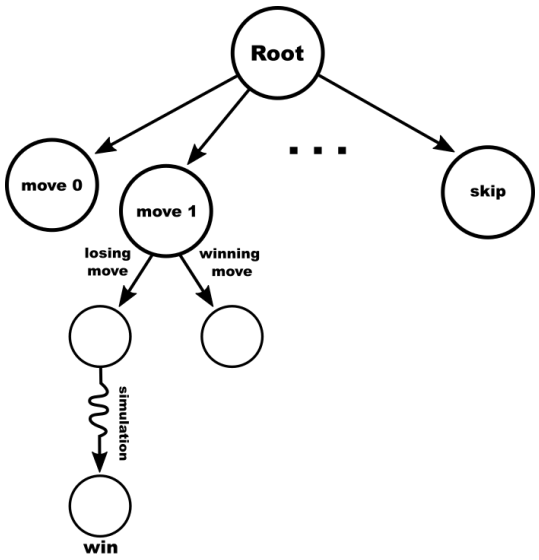

Figure 11: An illustration of a single simulation where executing move 1 on the root board results in a losing move. The simulation results in a win for the player.

3.2.5 Monte Carlo Tree Search Strategy

AlphaGo Zero [EMAS10] is an AI program designed for playing the game of Go. One of its key components is the Monte Carlo Tree Search (MCTS) algorithm, which differs from the regular Monte Carlo algorithm in an important way: it focuses on exploring the most promising paths instead of random paths. The MCTS algorithm can be applied to various games.
To initialize the MCTS algorithm, a tree is constructed with a single node as the root. This node contains information such as the total number of games won and played, as well as game-specific details like player ID and the next move to be made. The tree is then expanded using the following steps, as illustrated by Figure 12:

1. **Selection**: The optimal path from the root to a leaf node is chosen. At each node V, the best child is determined using the Upper Confidence Bound (UCB) formula:

   $$\text{UCB} = (C_{wins}/C_{games} + 0.5 \times \sqrt{\frac{\log V_{games}}{C_{games}}}) \times C_{chance}$$

   In the UCB formula, the first part represents the number of games won when simulating from the child node (C_{wins}) divided by the number of games simulated from the child node (C_{games}). The second part of the formula involves the number of games simulated from the node V (V_{games}) and the number of games simulated from the child node (C_{games}). The resulting value is then multiplied by the probability of the move leading to a winning attack (C_{chance}). If no games have been simulated from the child node, the UCB receives a default score of 10 to prioritize unexplored paths over known paths.

2. **Expansion**: Once a leaf node has been selected, it checks if the leaf is already in a final game state. If so, expansion and simulations are skipped. Otherwise, all possible moves from the current game state are computed, and a new leaf is added to the tree for each move.

3. **Simulation**: From the newly added leaf nodes, a random leaf node is chosen, and a predefined number of simulations are performed. Each simulation applies the Monte Carlo algorithm, starting with the selected move and then using the random strategy until the end of the game. After completing all simulations, the child node is updated with the number of games won and the number of games simulated.

4. **Backpropagation**: The results obtained from the child node are propagated up the tree. The parent node's total number of wins and total number of simulations are updated accordingly. In our implementation, the values in the backpropagation process are multiplied at each step by the probability of reaching the move from that ancestor. This backpropagation continues until it reaches the root node.

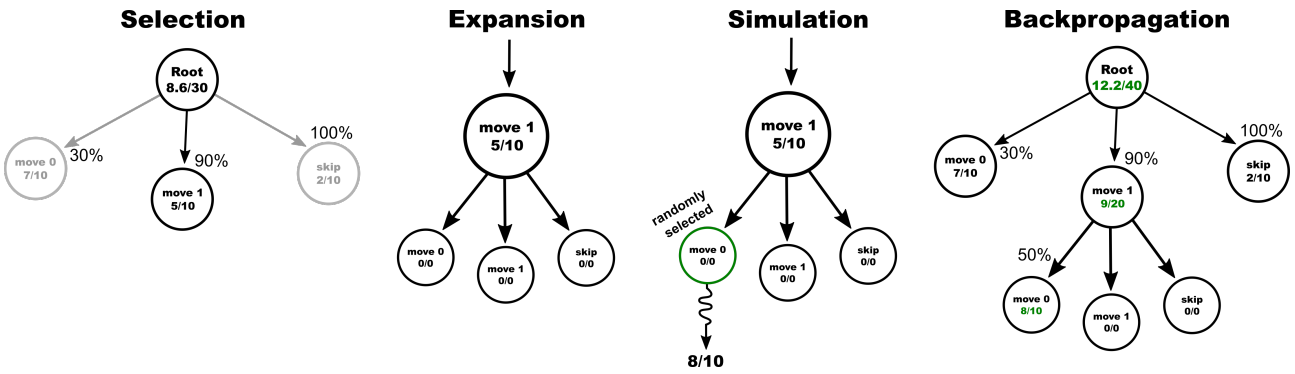

Figure 12: Steps involved in the Monte Carlo Tree Search algorithm.

The loop described above is repeated for a fixed number of iterations. After this, the best move is selected based on the total number of simulations performed for each move, rather than the win to total games ratio. This is because the selection phase already accounts for the best possible score using the UCB formula. Therefore, the move with the highest number of simulations is chosen. The process continues until the skip option is selected or no more moves are available.
During the selection phase of the MCTS algorithm's loop, there is a small issue: the construction of the tree does not explicitly consider losing moves and assumes that all moves leading to the optimal game state are winning moves. This decision was made based on AlphaGo's approach, which did not factor in the probability of winning a move but focused on games without probabilities, such as Go and chess.
In our implementation, we initially considered both winning moves and losing moves by simulating them. However, we found that including losing moves did not significantly improve the results compared to the greedy strategy. To maintain simplicity and stay closer to the original algorithm, we decided to exclude losing moves from further consideration in this strategy. Additionally, we experimented with slight changes to the algorithm, which resulted in equal or worse outcomes, which made us decide to stick as closely as possible to the original algorithm.

3.2.6 Expansion Strategy

The expansion strategy is similar to the greedy strategy in that it only looks one move ahead. However, the score calculation for a move is different. Instead of considering the average number

of dice that can be obtained, it focuses on the average increase in the size of the largest cluster of tiles achievable with a move. Figure 13 illustrates the difference between the greedy algorithm and the expansion strategy. The greedy algorithm selects the move with a winning probability of approximately 99%, depicted by the white arrow labeled by number 1. This move is favorable for the greedy strategy since it maximizes the number of dice obtained while also looking at the possible reinforcements:

$$score_1 = (38 + 13) \times 0.99 + (35 + 12) \times 0.01 = 50.96$$
$$score_2 = (38 + 14) \times 0.74 + (35 + 12) \times 0.26 = 50.7$$

This scoring formula for the greedy strategy is explained in Section 3.1.3.
However, the expansion strategy makes a different move, indicated by the white arrow labeled by number 2. Although this move may not result in the largest number of dice, it does result in the largest contiguous territory:

$$score_1 = 13 \times 0.99 + 12 \times 0.01 = 12.99$$
$$score_2 = 14 \times 0.74 + 12 \times 0.26 = 13.48$$

This strategy continues by prioritizing moves that expand the largest territory. If there are multiple territories of equal size, the algorithm treats moves that expand any of these territories equally. If no further moves are available to expand the largest territory, the strategy skips the turn. If the skip option is not yet available, meaning that no moves have been made yet, the strategy executes a random move and checks if any new moves can increase the largest area. If this is not the case, the turn is skipped, after all.

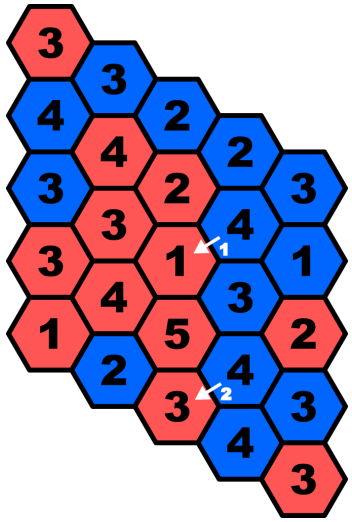

Figure 13: A 5×5 board illustrating the difference between the greedy strategy (1) and the expansion strategy (2).

The reason for choosing this strategy is to examine the relative importance of having a larger contiguous territory versus possessing more dice. If there is a significant difference, the player can adjust his strategy accordingly and focus on one of the two options. If there is no significant difference, the player can select the strategy that he finds easier to use.

3.3 Probability of Winning

In the second part of this research, we examine the winning probabilities of a player in the game states of Dice of Doom, assuming both players play optimally. In this part, we define a game state as a combination of the skip option, the current player and a possible board state that can be created given a size of the board, the number of players, and the maximum number of dice that can be on a tile. For our analysis, we have set the number of players to two and the maximal number of dice to five, as described in Section 2.1. We have chosen these values to ensure computational feasibility. However, these numbers can be changed in our method and are not fixed values. Furthermore, unlike the previous section about the strategies that used a limit of 100 turns for a game, we do not use this restriction here, since for a given game state, we cannot determine how many turns have already been played to reach that state.
To implement the optimal play of both players, we use a strategy similar to the Minimax method described in [MSZ13]. Unlike the greedy strategy, this approach considers the entire game tree and selects the best path.
In this method, we also consider the probability of a game ending in a tie. This factor is influenced by the different board sizes we test in this method. Since there is no tiebreaker in this game, no points are added to the probability of winning or losing when a tie occurs. Therefore, the probability of a tie must also be calculated.
Additionally, there is a possibility that the game continues indefinitely, which presents a challenge to calculate the exact probability and introduces uncertainties. In this section, we address this issue and suggest a solution to reduce the probability of uncertainties to zero.

3.3.1 Total Combinations of the Board

To compute the winning probabilities, we need the possible board combinations, which include all possible ways to distribute dice and assign ownership to each tile. Each tile has an owner (red or blue) and a number of dice (ranging from 1 to 5).
To construct the formula for calculating the total number of board combinations, we also need to consider whose turn it is on a given board and whether the skip option is available. We assume that it is always the turn of the red player, as there is always a mirrored version of any given board where the ownership of the red and blue players is reversed. This means that the new red player in the mirrored board was the old blue player. Therefore, we have not considered the current player in the formula. Regarding the skip option, which indicates that a player has already attacked during his turn, there is no simple way to compact this. While there are certain board states where the skip option cannot be achieved, calculating these states precisely is difficult, and they represent only a small portion of the total combinations. Taking all this information into account, the general formula for calculating the total number of board combinations is as follows:

$$\text{total} = (M_p \times M_d)^{(w \times h)} \times 2$$

In this equation, w and h represent the width and height of the game board, respectively. M_p represents the number of players, and M_d represents the maximum number of dice per tile. The multiplication by two accounts for the availability of the skip option.
For the board used in the game Dice of Doom, this equation results in a total of 2 septillion ($10^{25} \times 2$) board combinations.

3.3.2 How to Implement

To calculate the probability of winning, we employ the brute force algorithm, which involves examining all possible moves for a given board. This algorithm uses a game tree, where the leaf nodes represent final game states in which the player either wins, loses, or ties. At each node in the tree, the player can choose which move to make. The root of the tree corresponds to the starting game state for which we want to determine the probabilities.
When making optimal moves at the nodes of the tree, we select the move that obtains the highest probability of winning by examining the probabilities of winning of the node's children. The winning probability of the selected move is then stored in the current node. This process continues until the root of the tree contains its win probability.
To store the probabilities of winning, losing, or ending in a tie, we use a data structure that holds all three values. This data structure is returned by the brute force algorithm.
As explained in Section 3.3.1, we assume that in every state, the turn is for the red player. Of course, this assumption does not accurately reflect real gameplay. To correctly compute the win probabilities, we simply flip the ownership of each tile at the end of each turn. The old blue player, who now possesses the mirrored red tiles, can start his turn.

3.3.3 Board Conversion

To convert a given game state into an encoding, we examine the board state. By considering the width and length of the board, along with the ownership and number of dice for each tile, we can encode the current board. The conversion algorithm uses a variable e, initialized to 0, which will eventually hold the final encoding. We iterate over the tiles and apply the following formula:

$$e = (e \times M_p + n_p) \times M_d + n_d$$

In this formula, M_p represents the number of players, n_p represents the owner of the tile, M_d represents the maximum number of dice, and n_d represents the number of dice on the tile. The iteration over the tiles continues until all tiles have been processed. At the end, the resulting encoding e represents a number ranging from 0 to the maximum number of possible combinations divided by 2.
It is important to note that this encoding specifically aims to retrieve the board state for a given board configuration. However, it does not account for the skip option. To incorporate the skip option into the game state, the following formula is used:

$$e = e \times 2 + s$$

Here, s represents the availability of the skip option, which is represented as either 0 (false) or 1 (true).
During the decoding of the game state, the first step involves checking whether the the skip option is available by checking if the encoded number is odd or even. After obtaining the skip option, the integer e is divided by 2.
Using this remaining e, we can reconstruct the original board state by reversing the encoding process. The decoding continues until reverse iteration is complete and e becomes 0. This signifies the completion of the encoding process.

3.3.4 Array of Probabilities

To speed up the process of calculating the probability of winning, we introduce an array that stores the probabilities for each board state. In this array, each index corresponds to the encoding of a specific game state, while the value at that index represents the winning probability for that game state.
During the calculations of the winning probability for each game state, we can now look up previously calculated game states. If the value is known, it can be used as the winning probability instead of calculating it again.
It is worth noting that out of all possible board combinations, a small portion is already in a final state. This portion is typically around 10% to 20%, depending on the size of the board. For these board situations, the probabilities are relatively easy to calculate.

3.3.5 Problem of Cycles

There is a significant problem with this brute force method to compute the win probabilities.
For certain game states, the method does not halt. This means that somewhere in the brute force search space, there are loops. These loops are created by a sequence of specific moves that result in the original game board, as illustrated in Figure 14. One option is to end the game after playing a certain number of turns. This ensures that the loop eventually ends. However, this would leave a lot of uncertainty for larger game boards. Instead, we discuss two other methods to deal with these loops.

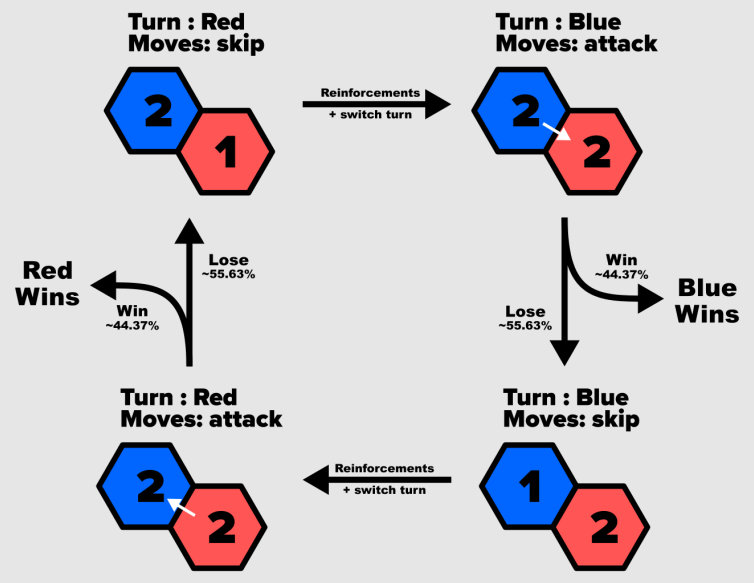

Figure 14: An example of a loop in a 1×2 board situation.

3.3.6 Conversion to Absorbing Markov Chains

We may approach these loops by considering them as Absorbing Markov chains [KS76]. In short, a Markov chain is a countable set of states connected to each other with certain probabilities. The

introduction of absorbing states turns a Markov chain into an absorbing Markov chain. Absorbing states possess a special property: instead of transitioning to another state, they transition to themself with a 100% probability, essentially acting as black holes. For every non-absorbing state, there must exist a path from that state to an absorbing state.
In our case, we can view the game loop as a Markov chain, because there are a countable number of states, as described in Section 3.3.1, and each game state has a probability of transitioning to another game state, as described in Section 3.1.5. For final game states, we can assign a transition to itself with a 100% probability, making them absorbing states. This way, all states adhere to the definition of a Markov chain.
Since the absorbing states represent the final game states and for every game state, there exists a path to a final state, we can conclude that the game can be represented as an absorbing Markov chain. Additionally, we introduce two new labels apart from absorbing and non-absorbing. One of the labels used is called 'known', which is defined by either being an absorbing state with a known probability of winning or a state where all possible children are known. If all the children of a state are known, the probability in the current state can be calculated. The other label is called 'unknown', which occurs when one or more of a state's children are unknown. The optimal winning probability of the state cannot be calculated if at least one of the children remains unknown. Such states consist of those in a loop or those dependent on the win probability of a state in a loop. Figure 14 shows an example of these states being in a loop, where not all children are known, and thus the optimal winning probabilities cannot be assigned to them.
There are ways to solve absorbing Markov chains. However, those methods cannot be simply applied to our game tree. In some cases, multiple cycles are merged, resulting in a node in the cycle that has two or more paths that eventually lead back to itself, as illustrated in Figure 15. The graph shows state A leading into states B and C, and both B and C lead back into A. Since B and C are both dependent on state A, they are both unknown. This causes the problem of not knowing which move is the optimal move from state A. We did think of a way to solve this problem, which involved trying both unknown moves and seeing which of the two gave more optimal results. However, for a 2×2 game board, the number of states similar to state A is around 300. Which meant that we would have to check an impractically large number of possibilities.

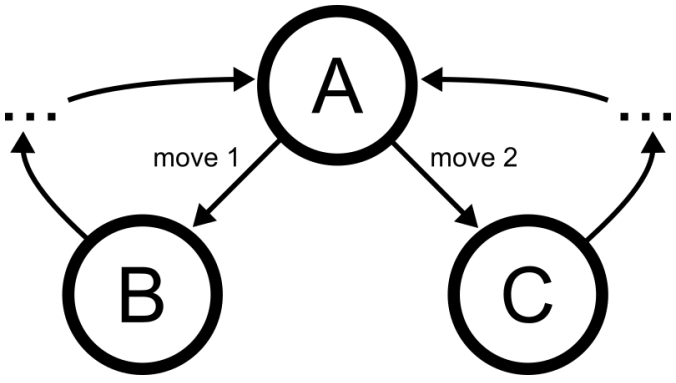

Figure 15: Cycle problem: What is the best move for state A?

3.3.7 Policy Iteration

To avoid the complexity of trying to calculate the exact probabilities in case of cycles, we use the policy iteration method as described in [How60]. This method allows us to achieve a close-to-optimal result through an iterative process. The method consists of multiple iterations. In the first iteration, the probabilities of all possible game states where the red player starts are initialized. In the following iterations, new probabilities for the states are calculated using the children of the given states.
For the initialization, the value is calculated as follows:

$$v_0(k) = \begin{cases} 1 & \text{if the red player wins} \\ 0 & \text{otherwise} \end{cases}$$

That is, for each state k, it is checked if the red player has won in that state. If he has, the value of $v_0(k)$ is set to 1, indicating a 100% winning probability in that state. If it is known to be a losing state or a tie, the winning probability is set to 0. If the node is not a final state, its value is also set to 0. In each subsequent iteration, we determine the maximum value among the children of the current node k and calculate the probability of reaching that specific node.

$$v_n(k) = \max_o \sum_l P_{k,l}(o) \times v_n(l)$$

In this formula, n is the current iteration, o represents all possible moves for state k, and l represents all game states. The value of $P_{k,l}(o)$ is the probability that the player transitions from state k to state l using move o. This probability is then multiplied by the probability of state l from the previous iteration. The sum of all $P_{k,l}(o)$ always adds up to 1 for a fixed k.
If these iterations were repeated infinitely, the system should converge to a situation where:

$$v^*(k) = \max_o \sum_l P_{k,l}(o) \times v^*(l)$$

Here, the probability $v^*(k)$ is solely dependent on $v^*(l)$ in the same iteration, making it the final answer.
However, there are some limitations to this method. First, most of the nodes will remain at a probability of 0 for the first iterations. At the start, only the final states are known, and over the iterations, that information propagates to the other nodes. Additionally, this method can be computationally expensive for large board sizes. As the board size increases, so does the number of states in the Markov chain, the number of possible moves for a state, and consequently, the number of iterations needed to obtain a more accurate result. Another limitation is that, for our implementation of the board game, the reinforcements must be placed in a deterministic manner. Otherwise, it becomes very complex to determine the new board state after a skipping move.
In our implementation, we made some adjustments to this method. First, to calculate the best possible move from a given state k, we only examine the states that can be reached from that state, rather than considering all game states l. Second, if a state becomes 'known' in iteration $n = i$, it is skipped in all iterations $n > i$. After a large number of iterations, only the states that are either part of a cycle or dependent on a cycle remain unknown. In our method, we chose to perform a total of 100 iterations to obtain an accurate result. In practice, we found that already after around 20 iterations for a small board size, the system had somehow converged.

3.4 Optimal Strategy

In addition to the strategies described in Section 3.2, we introduce a close-to-optimal strategy based on the method outlined in Section 3.3. The reason it is referred to as close-to-optimal, rather than simply optimal, is because the method approximates the optimal strategy through a number of iterations. If only a small number of iterations is used, the strategy may not perform optimally.
This close-to-optimal strategy relies on the winning probability in every game state. It determines one of the optimal moves by calculating the weighted average win probabilities of the winning and losing moves, maximizing this value. Additionally, the skip option is used as a baseline for comparison whenever it is available. In our case, the optimal move refers to the move with the highest win probability.
The structure of this strategy is relatively simple compared to other strategies. At the beginning of the game, it calculates the best move and its corresponding winning probability for all possible states, storing these values in a lookup table. Once the table has been constructed, the game can start. For any given game state, the player can simply consult the lookup table using the current board lay out and the skip option, and execute the best move suggested by the lookup table.
This method offers several advantages. In addition to its accuracy, an advantage is that the lookup table only needs to be calculated once. After creating the tables for various games with different parameters, they can be reused across multiple matches. Although this lookup table can be applied to other strategies as well, it neglects the random element present in most strategies.
This method also has a serious disadvantage. As mentioned in Section 3.3.1, the original game has an enormous number of 2 septillion game states. Calculating and storing the probability for each state over many iterations is practically infeasible. Therefore we only experiment with smaller board sizes.

3.5 Simulations

To obtain answers to the research questions, we need to generate results through simulations. In the first part of the experiments, we focus on comparing the greedy strategy to other strategies. In the second part, we analyze the probability of winning and how it varies with different limits and board sizes.

3.5.1 Strategies

The different strategies are compared to the greedy strategy. To achieve this, we simulate a total of 100.000 games (if possible), and determine which of the two strategies wins more games. To ensure fairness in the simulations, we alternate the starting player throughout the simulations, starting with the greedy strategy.
At the end of the simulations, we obtain the results, which include the number of wins, losses, and ties. As explained in Section 2.2, a game that exceeds 100 turns are also considered a tie.
Regarding the experiments, we examine variations in board sizes, board initialization, maximum number of dice, and reinforcement types. The chosen board sizes are as follows: 1×2, 2×2, 2×3, 3×3, 3×4, 4×4, and 5×5. For board initialization, we use the two methods described in Section 2.4: random and random but with an equal number of tiles ($\pm$ 1) for both players. The maximum number of dice on a tile is set to two, respectively five. The reinforcement types used, as described in Section 2.4, are random, deterministic, and weighted random.
If the simulations become too time-consuming for any of the experiments, the total number of games played is reduced to 10.000. If even that proves to be time-consuming as well, further reductions in the number of games is made. However, this may lead to less precise results, although it may still provide insights into differences between the two strategies.
For the strategy that is created by taking the close-to-optimal approach from Section 3.4. We only use the deterministic distribution, since it meets the method's requirements.

3.5.2 Probability of Winning

The second part of the simulations focuses on the probability of winning. We conduct two experiments using the lookup table. The first experiment involves determining the average win probability, which helps us assess whether it is advantage to be the starting player. In the second experiment, we compare the optimal move to the greedy move. This allows us to determine whether the greedy algorithm is optimal.
To calculate the average win probability, we use the array of probabilities explained in Section 3.3.4. This table stores the winning probability, losing probability, and tie probability. By averaging these values across all game states, we can calculate the average win probability.
For comparing the optimal move to the greedy move, we also use the lookup table. As described in Section 3.4, the lookup table contains the win probabilities from which we can deduce the optimal moves. For each possible game state in which you can make a move, we collect the win probability of the move made by the greedy strategy and the win probability of the optimal move. Then, we separately sum up these two probabilities. If the sum of probabilities are identical, we know that the greedy algorithm is optimal. If the sum of probabilities of the optimal strategy is higher than that of the greedy strategy, we know that the greedy strategy is not optimal. We assume that the greedy strategy does not outperform the optimal strategy.

In this experiment, we also include the random strategy. This strategy acts as a baseline. For every game state, we take a random move and sum the probabilities of that move. Just like we did for the greedy strategy. We assume the sum of the random strategy to be lower than both the greedy strategy and the close-to-optimal strategy.
For the experiments, we examine variations in board sizes and the number of dice. The selected board sizes are as follows: 1×2, 2×2, 2×3, 3×3, 3×4, 4×4, and 5×5. The maximum number of dice on a tile is set to two, respectively five.

4 Experiments

To evaluate our methods, we conducted simulations as described in Section 3.5. The experiments consisted of two parts: testing different strategies (Section 4.1) and analyzing the probability of winning (Section 4.2). The results of both experiments are described below.

4.1 Strategies

As explained in Section 3.2, we compared our strategies to the greedy strategy. There are six strategies, including the close-to-optimal strategy described in Section 3.4. We analyzed the results of these strategies and their performance to the greedy strategy. If any significant positive results are observed in favor of the competing algorithms, we can conclude that there is a strategy that can beat a human player using the greedy algorithm.

4.1.1 Baseline

For the baseline, we compared the greedy strategy to itself. The results, shown in Table B.1.1, yield several interesting observations. When the maximum number of dice was reduced to two, the number of ties increased. We also noticed a slight improvement in the number of wins when we used non-deterministic reinforcement types for the greedy algorithm. Additionally, more ties occurred when the board initialization resulted in an equal number of tiles for both players.

4.1.2 Random Strategy

The random strategy selects a move randomly. When comparing this strategy to the greedy strategy, the results shown in Table B.1.2 lead to some interesting findings. One such finding is that the random strategy seems to outperform the greedy strategy for smaller board sizes and when the maximum number of dice is two. This is not the case for reinforcement type 2. However, the greedy strategy makes clearly outperforms the random strategy when the maximum number of dice allowed on a tile is set to five. Additionally, there are more ties when we use non-deterministic reinforcement types.

4.1.3 Return Zero Strategy

For the return zero strategy, which always selects the first move in the list, the results are presented in Table B.1.3. When compared to the greedy strategy, the return zero strategy shows slightly better performance than the random strategy. For this strategy, as for the random strategy, we observe that it outperforms the greedy algorithm when the maximum number of dice is reduced to two. In all the other situations, the greedy algorithm clearly outperforms the return zero strategy. Additionally, we see a slight increase in wins for initialization type 2 compared to type 1. Specifically, for games with a maximum of five dice, the greedy strategy wins more often when there is an equal board initialization compared to a random board initialization. For games with a maximum of two dice, the return zero strategy has a slight advantage with an equal board initialization.

4.1.4 Monte Carlo Strategy

The Monte Carlo strategy has fewer results, as shown in Table B.1.4. This is because when the size of the board increases, the number of moves that need to be simulated increases as well. Similar to the previous strategies, the performance of the Monte Carlo strategy varied depending on whether there were two or five dice. Furthermore, we found that this strategy has a lower win to lose ratio compared to the other strategies, and the completely random board initialization seems to have a higher win rate for the Monte Carlo strategy compared to the other initialization type.

4.1.5 Monte Carlo Tree Search Strategy

The Monte Carlo Tree Search (MCTS) strategy has even fewer results, as shown in Table B.1.5. As described in Section 3.2.5, the MCTS strategy uses a Monte Carlo search in the leaf nodes of its tree, which is computationally expensive. As the board size increases and the number of possible moves grows, the tree becomes larger. Based on the limited results obtained, we found that the number of wins of the MCTS strategy are slightly better compared to the win rates of other strategies, especially for the maximum of two dice. However, we need more research results to draw a definitive conclusion.

4.1.6 Expansion Strategy

The expansion strategy is similar to the greedy algorithm. When comparing it to the greedy strategy, the results shown in Table B.1.6 indicate a significant difference between the two strategies other than the 1×2 board size. Similar to the random strategy and return zero strategy, the same occurrence is observed for the maximum of two and five dice. Overall, no significant results were found.

4.1.7 Close-to-Optimal Strategy

For the close-to-optimal strategy, the results shown in Table B.1.7 indicate that it outperforms the greedy strategy for all board sizes except for the 1×2 board size, which is as expected. Unlike the previous strategies, the close-to-optimal strategy also outperformed the greedy strategy when the maximum number of dice is five. Some results are missing due to the large number of board combinations for a 5×5 board, as explained in Section 3.3.1. For this reason it was not possible for us, storage wise, to simulate the bigger board sizes. Additionally, other reinforcement types were not fully explored, as the close-to-optimal algorithm relies on a deterministic reinforcement method. Despite these limitations, the promising results suggest that the close-to-optimal strategy may also outperform the greedy algorithm on a 5×5 board.

4.2 Probability of Winning

In the second part of our research, we examined the optimal win probability and analyzed the performance of the greedy strategy. Since the probability table can only be used during gameplay, it is difficult to experiment with the results. However, we focused on two points of interest: the average win probability of the starting player and how optimal the greedy strategy is. Note that there are some missing data points in the tables for larger board sizes. This is due to the number of board combinations, as explained in Section 3.3.1.

4.2.1 Average Win Probability

To calculate the average win probability, as explained in Section 3.5, we took the average of the probabilities from all game states. The results, shown in Table B.2.1, reveal an advantage for the starting player. For both the maximum of five and two dice, the starting player has a higher win probability compared to the second player, with this advantage increasing as the size of the board increases. This advantage also increases when the maximum number of dice on a tile is set to two. Additionally, the maximum of five dice has a significantly lower average tie probability compared to the maximum of two dice.
One other minor detail that can be seen in the table, is that for bigger board states, the probabilities do not sum up to one. As mentioned, policy iteration approximates the win probabilities. This means that for more accurate results, more iterations are required.

4.2.2 How Optimal is the Greedy Strategy?

The results regarding the optimality of the greedy strategy are outlined in Table B.2.2. The table displays three different strategies: the close-to-optimal strategy, the greedy strategy, and the random strategy. Additionally, the table shows the number of game states where a move is possible, signifying states that are not final. The percentage of these states in relation to the total number of states is also indicated. The random strategy acts as a baseline to provide further insight into the overall improvements of the greedy strategy and the close-to-optimal strategy.
By comparing the results of the different strategies, we found that the greedy strategy is not optimal. The average win probability of the greedy strategy is not greater than the average win probability of the optimal strategy. In our experiments, we also discovered that the greedy strategy does not have a single move which outperforms the close-to-optimal strategy. This implies that in all states where a move can be executed, the close-to-optimal strategy either equals or outperforms the greedy strategy.
Moreover, the results also show an increase in win probability when the board size increases. Also, when comparing the maximum number of dice that are allowed on a tile, the win probability for the maximum of five dice is lower compared to the win probability when the maximum is two dice. This result is also shared with the previous results of Table B.2.1.
Furthermore, the different strategies do not show a significant difference. This is probably caused by the number of similar moves that all three strategies executed which resulted in the same probability. When a move is made that doesn't match the optimal move, the penalty would not be significant, as the move would still have a probability close to the optimal value.

4.3 Conclusion

The main aim of this thesis is to find the optimal strategy for the game Dice of Doom. This goal has been divided into two main parts: different strategies and win probabilities. Furthermore, we implemented the game Dice of Doom in C++. This research contained different strategies, with the greedy strategy being the most important. There were several different problems we encountered when trying to obtain the optimal winning probability. In order to solve these issues, we used policy iteration. Lastly, we conducted experiments, and presented the results.

In conclusion, several key observations can be made regarding the first research question. First, we found that the greedy strategy outperforms every other strategy when the maximum number of dice is five. However, when the maximum number of dice is reduced to two, the greedy strategy seems to be outperformed by every other strategy. We have found no explanation for this yet. Another clear, though not necessarily surprising, result is that the equal initialization method leads to more ties between strategies.

Furthermore, the research revealed no significant difference between the various reinforcement methods used in the experiments. Additionally, when having our baseline, the greedy strategy, compete against itself, we found fewer ties with a maximum of five dice than with a maximum of two dice per tile.

For the second research question, we made several observations. We determined that the greedy strategy is not optimal because there is a difference in win probability compared to the close-to-optimal strategy. Moreover, the starting player has a slight advantage in a game with five dice, while starting with two dice offers a considerable advantage. This suggests that allowing more dice to be on a tile makes the game more fair for both players. Additionally, we found that there were more ties when fewer dice were involved.

Regarding the insights gained from converting the win probabilities, obtained with policy iteration, to a strategy, we found that for smaller game boards, the performance of the greedy strategy and the close-to-optimal strategy remained practically equal. However, as the size of the board increases, the close-to-optimal strategy outperforms the greedy strategy. Also in these simulations, we found that the equal initialization method led to an increased number of ties in both research questions.

4.4 Future Work

Since this research is in a relatively new field, focusing on the game Dice of Doom, we have several suggestions for further research. These ideas involve improvements or extensions to the existing research, as well as applying similar methods to other games.

First, the difference in performance of the greedy strategy when the maximum number of dice is reduced to two could be further explored. Conducting a detailed analysis of the game mechanics and evaluating the implementation of the strategy may help uncover the reasons behind this difference. This may result in useful improvements or modifications to the greedy strategy.

As discussed, the impact of the reinforcement methods made no difference. However, further research can explore the reinforcement implementation where players have full control over the placement of their reinforcements. Investigating optimal reinforcement placement and analyzing their impact on the game's outcome can improve the understanding of different strategies for Dice of Doom.

Further research can focus on studying the variations as described in Section 2.4. Analyzing how these variations affect game play, strategy effectiveness, and winning probabilities may lead to

improvements to the original game.

Given the similarities between Dice of Doom and other dice-based strategy games such as Risk and Dice Wars, further research could explore the application of the policy iteration method in these games. Analyzing the performance of policy iteration across different games can show its effectiveness in other strategy games.

While our research mainly focused on two-player games, extending the analysis to multiplayer games could be an interesting direction for further research. Multiplayer games are a lot more complex as they have to take communication between players into account. This can lead to greater diversity of strategies to use against the greedy strategy.

References

[Ade14] A.I. Adekitan. Monte carlo simulation, 2014. Retrieved from https://www.researchgate.net/publication/326803384_MONTE_CARLO_SIMULATION.

[Bar11] C. Barski. *Land of Lisp*. no starch press, 2011.

[Blo20] E. Blomqvist. Playing the game of risk with an alphazero agent. Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2020.

[Dro18] J. Drogtrop. *Comparing Different Agents in the Game of Risk*. Bachelor's thesis, Leiden Institute of Advanced Computer Science (LIACS), 2018. Retrieved from https://theses.liacs.nl/pdf/2017-2018-DrogtropJ.pdf.

[EMAS10] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, Dec 2010.

[Geo04] H. Georgiou. Risk board game - battle outcome analysis, 2004. Retrieved from http://www.c4i.gr/xgeorgio/docs/RISK-board-game%20_rev-3.pdf.

[How60] Ronald A. Howard. *Dynamic Programming and Markov Processes*. The M.I.T. Press, 1960.

[KS76] J. G. Kemeny and J. L. Snell. *Finite Markov Chains (Second ed.)*, pages 43–58. Springer-Verlag, 1976.

[MSZ13] M. Maschler, E. Solan, and S. Zamir. *Game Theory*, page 176–180. Cambridge University Press, 2013.

A Greedy Algorithm

Input: A board layout, the skip option and the current player.
Output: The suggested move to execute.

```
Function greedy_strategy():
  max = the maximum number of moves possible on the board for the current player
  best = 0
  curr_best = -1

  For i from 0 to max-1:
    winning_board = a copy of the current board
    losing_board = a copy of the current board

    chance = the probability of winning the attack for move i

    Perform a winning move for move i on winning_board
    Perform a losing move for move i on losing_board

    score_win = the number of dice on winning_board + reinforcements
    score_lose = the number of dice on losing_board + reinforcements
    score = chance × score_win + (1.0 - chance) × score_lose

    If score > curr_best:
      curr_best = score
      best = i

  If the skip option is available:
    score_skip = the number of dice on the board + reinforcements
    If score_skip >= curr_best:
      best = max

  Return best
```

B Experiment Results

B.1 Strategies

Below are the tables displaying the results. In the tables, the letter A represents the wins of the current strategy, while the letter B represents the wins of the greedy strategy. The different types mentioned in the first row of the table indicate the reinforcement types, with Type 1 representing deterministic distribution, Type 2 representing the fully random method, and Type 3 representing the weighted random method. The 'Init' column refers to the board initialization, which can be either random (0) or random with the same number of tiles for both players (1). The 'Dice' column indicates the maximum number of dice per tile in the game, which can be either five dice or two dice. The 'Size' column represents the size of the board, including the width and height.

B.1.1 Baseline

Both A and B stand for the greedy strategy.

Type 1			Type 2			Type 3			Variables		
A	B	Tie	A	B	Tie	A	B	Tie	Init	Dice	Size
44597	45043	10360	45030	44759	10211	44495	44997	10508	0	5	1x2
39695	39517	20788	39927	39299	20774	39533	39610	20857	1	5	1x2
36379	36357	27264	36548	36598	26854	36410	36651	26939	0	2	1x2
22980	23010	54010	22624	23265	54111	23188	22774	54038	1	2	1x2
4735	4819	446	4922	4943	135	5016	4799	185	0	5	2x2
4571	4576	853	4857	4713	430	4788	4783	429	1	5	2x2
4460	4436	1104	4444	4397	1159	4464	4345	1191	0	2	2x2
3608	3650	2742	3662	3660	2678	3671	3713	2616	1	2	2x2
4754	4688	558	5084	4883	33	4964	4949	87	0	5	2x3
4555	4587	858	4902	4985	113	4948	4895	157	1	5	2x3
4598	4779	623	4657	4545	798	4680	4601	719	0	2	2x3
4208	4150	1642	4099	4182	1719	4181	4144	1675	1	2	2x3
4376	4274	1350	4926	5043	31	4928	4936	136	0	5	3x3
4090	4131	1779	4906	5033	61	4924	4935	141	1	5	3x3
4270	4238	1492	4464	4301	1235	4608	4463	929	0	2	3x3
4193	4116	1691	4298	4215	1487	4465	4458	1077	1	2	3x3
4223	4179	1598	4950	4968	82	4854	5047	99	0	5	3x4
3949	3928	2123	4932	4940	128	4899	4987	114	1	5	3x4
3314	3288	3398	3708	3692	2600	4254	4204	1542	0	2	3x4
2994	2969	4037	3537	3435	3028	4041	4056	1903	1	2	3x4
3972	4026	2002	4851	4869	280	4923	4981	96	0	5	4x4
3746	3748	2506	4814	4827	359	4992	4892	116	1	5	4x4
2172	2157	5671	3005	3009	3986	3678	3670	2652	0	2	4x4
1719	1847	6434	2808	2816	4376	3621	3516	2863	1	2	4x4
326	335	339	478	421	101	486	491	23	0	5	5x5
326	294	380	428	457	115	506	468	26	1	5	5x5
80	81	839	223	229	548	244	273	483	0	2	5x5
65	70	865	179	188	633	249	233	518	1	2	5x5

B.1.2 Random Strategy

Here, A stands for the random strategy, and B stands for the greedy strategy.

Type 1			Type 2			Type 3			Variables		
A	B	Tie	A	B	Tie	A	B	Tie	Init	Dice	Size
44786	44781	10433	44868	44659	10473	44627	44978	10395	0	5	1x2
39860	39417	20723	39441	39780	20779	39623	39422	20955	1	5	1x2
36647	36210	27143	36532	36667	26801	36854	35952	27194	0	2	1x2
22980	22607	54413	22833	22937	54230	23343	22647	54010	1	2	1x2
44003	54215	1782	42850	55364	1786	41378	56174	2448	0	5	2x2
40529	55340	4131	38536	57349	4115	36482	58529	4989	1	5	2x2
45213	42801	11986	42959	44337	12704	44638	43411	11951	0	2	2x2
37613	34917	27470	34696	36892	28412	37548	34740	27712	1	2	2x2
37922	61624	454	36376	63054	570	35849	63319	832	0	5	2x3
32122	66742	1136	29842	68835	1323	29982	68283	1735	1	5	2x3
53625	40106	6269	42921	49758	7321	49380	44021	6599	0	2	2x3
51184	33276	15540	36424	46340	17236	45715	38289	15996	1	2	2x3
33176	66824	0	31099	68901	0	32363	67636	1	0	5	3x3
27470	72530	0	25052	74948	0	26956	73043	1	1	5	3x3
74240	24434	1326	48388	49353	2259	67171	28819	4010	0	2	3x3
78471	19922	1607	47386	49889	2725	70201	25001	4798	1	2	3x3
28377	71615	8	27034	72951	15	30001	69975	24	0	5	3x4
20584	79383	33	19827	80145	28	23325	76636	39	1	5	3x4
82465	15021	2514	47243	44692	8065	78031	16119	5850	0	2	3x4
85552	10130	4318	45219	43600	11181	81148	10971	7881	1	2	3x4
25134	74839	27	24257	75717	26	28132	71843	25	0	5	4x4
18293	81679	28	17592	82383	25	22160	77814	26	1	5	4x4
89649	8074	2277	49595	39409	10996	81335	8896	9769	0	2	4x4
92054	5205	2741	48924	37619	13457	83932	5479	10589	1	2	4x4
19015	80557	428	19320	80312	368	26293	73610	97	0	5	5x5
12694	86908	398	13395	86229	376	21986	77881	133	1	5	5x5
93797	4066	2137	45921	34894	19185	85750	3610	10640	0	2	5x5
95133	2914	1953	45181	33505	21314	87383	2372	10245	1	2	5x5

B.1.3 Return Zero Strategy

Here, A stands for the return zero strategy, and B stands for the greedy strategy.

Type 1			Type 2			Type 3			Variables		
A	B	Tie	A	B	Tie	A	B	Tie	Init	Dice	Size
44789	44810	10401	44788	44861	10351	44865	44742	10393	0	5	1x2
39947	39363	20690	39762	39611	20627	39601	39432	20967	1	5	1x2
36331	36609	27060	36559	36402	27039	36545	36329	27126	0	2	1x2
22763	23018	54219	22840	22954	54206	22731	22875	54394	1	2	1x2
43395	55018	1587	43378	55074	1548	41505	56878	1617	0	5	2x2
39236	56722	4042	38907	57121	3972	35975	59879	4146	1	5	2x2
51146	37087	11767	51217	37100	11683	50992	37114	11894	0	2	2x2
47096	25938	26966	46810	26321	26869	47222	25772	27006	1	2	2x2
41735	57911	354	42319	57237	444	39567	59806	627	0	5	2x3
37430	61405	1165	38339	60476	1185	35382	63143	1475	1	5	2x3
65670	28467	5863	65590	28739	5671	65756	28394	5850	0	2	2x3
68076	17172	14752	68160	16938	14902	68076	17128	14796	1	2	2x3
40095	59905	0	41863	58137	0	38100	61900	0	0	5	3x3
36488	63512	0	38575	61425	0	34597	65403	0	1	5	3x3
84232	15768	0	84109	15891	0	84267	15733	0	0	2	3x3
89488	10512	0	89485	10515	0	89547	10453	0	1	2	3x3
3511	6487	2	4068	5931	1	3634	6352	14	0	5	3x4
3056	6942	2	3715	6283	2	3255	6727	18	1	5	3x4
9121	803	76	9140	793	67	9155	774	71	0	2	3x4
9491	313	196	9426	340	234	9420	355	225	1	2	3x4
3511	6488	1	4013	5987	0	3585	6412	3	0	5	4x4
3068	6932	0	3639	6361	0	3243	6754	3	1	5	4x4
9600	387	13	9596	392	12	9636	344	20	0	2	4x4
9812	137	51	9804	139	57	9801	134	65	1	2	4x4
2957	7004	39	3757	6239	4	3529	6470	1	0	5	5x5
2347	7621	32	3475	6519	6	3324	6675	1	1	5	5x5
9896	104	0	9894	106	0	9885	115	0	0	2	5x5
9952	48	0	9957	43	0	9954	46	0	1	2	5x5

B.1.4 Monte Carlo Strategy

Here, A stands for the Monte Carlo strategy, and B stands for the greedy strategy.

Type 1			Type 2			Type 3			Variables		
A	B	Tie	A	B	Tie	A	B	Tie	Init	Dice	Size
4445	4463	1092	4467	4487	1046	4425	4582	993	0	5	1x2
3938	3998	2064	4029	3951	2020	3858	4057	2085	1	5	1x2
3620	3682	2698	3672	3641	2687	3580	3681	2739	0	2	1x2
2271	2320	5409	2214	2270	5516	2335	2256	5409	1	2	1x2
4642	5206	152	4563	5277	160	4425	5431	144	0	5	2x2
4227	5337	436	4276	5324	400	4103	5496	401	1	5	2x2
4507	4320	1173	4266	4504	1230	4507	4298	1195	0	2	2x2
3764	3455	2781	3426	3773	2801	3712	3548	2740	1	2	2x2
3402	6555	43	3238	6673	89	3206	6624	170	0	5	2x3
2616	7279	105	2547	7263	190	2435	7222	343	1	5	2x3
5653	3701	646	4541	4575	884	5558	3741	701	0	2	2x3
5592	2907	1501	3838	4200	1962	5381	2976	1643	1	2	2x3
340	658	2	334	655	11	333	645	22	0	5	3x3
246	745	9	256	724	20	255	701	44	1	5	3x3
577	370	53	569	374	57	581	356	63	0	2	3x3
543	300	157	564	298	138	547	286	167	1	2	3x3

B.1.5 Monte Carlo Tree Search Strategy

Here, A stands for the Monte Carlo Tree Search strategy, and B stands for the greedy strategy.

Type 1			Type 2			Type 3			Variables		
A	B	Tie	A	B	Tie	A	B	Tie	Init	Dice	Size
4454	4528	1018	4456	4474	1070	4489	4447	1064	0	5	1x2
3968	3943	2089	3860	4077	2063	3948	3970	2082	1	5	1x2
3679	3633	2688	3659	3577	2764	3688	3638	2674	0	2	1x2
2298	2255	5447	2308	2223	5469	2304	2256	5440	1	2	1x2
4786	5050	164	4797	5059	144	4743	5079	178	0	5	2x2
4607	4988	405	4729	4851	420	4714	4846	440	1	5	2x2
4426	4423	1151	4467	4422	1111	4429	4444	1127	0	2	2x2
3625	3627	2748	3584	3667	2749	3589	3583	2828	1	2	2x2

B.1.6 Expansion Strategy

Here, A stands for the expansion strategy, and B stands for the greedy strategy.

Type 1			Type 2			Type 3			Variables		
A	B	Tie	A	B	Tie	A	B	Tie	Init	Dice	Size
45316	44364	10320	44744	44843	10413	44851	44899	10250	0	5	1x2
39470	39740	20790	39643	39744	20613	39407	39703	20890	1	5	1x2
36628	36340	27032	36466	36546	26988	36345	36690	26965	0	2	1x2
22921	23054	54025	22720	23105	54175	22964	23125	53911	1	2	1x2
43109	55269	1622	43146	55328	1526	41606	56749	1645	0	5	2x2
38799	57150	4051	38577	57427	3996	36607	59371	4022	1	5	2x2
51679	36925	11396	51701	36729	11570	51941	36525	11534	0	2	2x2
47495	25342	27163	47462	25261	27277	47539	25435	27026	1	2	2x2
43861	55774	365	45213	54440	347	44359	55204	437	0	5	2x3
40763	58062	1175	42733	56111	1156	41550	57211	1239	1	5	2x3
66594	27796	5610	66636	27727	5637	66601	27877	5522	0	2	2x3
68436	16567	14997	68509	16460	15031	68774	16480	14746	1	2	2x3
42211	57789	0	45359	54641	0	43857	56143	0	0	5	3x3
39015	60985	0	43437	56563	0	41495	58505	0	1	5	3x3
85055	14945	0	84900	15100	0	84682	15318	0	0	2	3x3
89995	10005	0	90149	9851	0	90043	9957	0	1	2	3x3
3854	6145	1	4563	5437	0	4204	5794	2	0	5	3x4
3469	6528	3	4292	5705	3	3996	6001	3	1	5	3x4
9138	794	68	9178	758	64	9149	801	50	0	2	3x4
9476	306	218	9477	291	232	9510	273	217	1	2	3x4
3771	6228	1	4540	5460	0	4105	5895	0	0	5	4x4
3239	6761	0	4293	5707	0	3873	6127	0	1	5	4x4
9643	342	15	9692	295	13	9679	309	12	0	2	4x4
9837	102	61	9813	122	65	9837	104	59	1	2	4x4
3158	6823	19	4387	5612	1	3948	6050	2	0	5	5x5
2733	7248	19	4168	5832	0	3687	6305	8	1	5	5x5
9924	74	2	9914	83	3	9907	92	1	0	2	5x5
9966	34	0	9962	37	1	9961	38	1	1	2	5x5

B.1.7 Close-to-Optimal Strategy

Here, A stands for the close-to-optimal strategy, and B stands for the greedy strategy.

Type 1			Variables		
A	B	Tie	Init	Dice	Size
44529	45221	10250	0	5	1x2
39619	39492	20889	1	5	1x2
36347	36896	26757	0	2	1x2
22915	23286	53799	1	2	1x2
52107	46277	1616	0	5	2x2
52124	43708	4168	1	5	2x2
50784	37670	11546	0	2	2x2
46154	26475	27371	1	2	2x2
56182	43110	708	0	5	2x3
58688	39274	2038	1	5	2x3
66783	27627	5590	0	2	2x3
69127	16021	14852	1	2	2x3
-	-	-	0	5	3x3
-	-	-	1	5	3x3
84692	15308	0	0	2	3x3
89945	10055	0	1	2	3x3
-	-	-	0	5	3x4
-	-	-	1	5	3x4
74056	8577	17367	0	2	3x4
74990	4036	20974	1	2	3x4
-	-	-	0	5	4x4
-	-	-	1	5	4x4
-	-	-	0	2	4x4
-	-	-	1	2	4x4
-	-	-	0	5	5x5
-	-	-	1	5	5x5
-	-	-	0	2	5x5
-	-	-	1	2	5x5

B.2 Win Probability

B.2.1 Average Win Probability

The table below shows the average win, lose and tie probability for the corresponding board size and maximum number of dice.

Win	Lose	Tie	Dice	Size
0.518462	0.4177	0.0638379	5	1x2
0.564132	0.427367	0.00850344	5	2x2
0.58442	0.41484	0.00191217	5	2x3
-	-	-	5	3x3
-	-	-	5	3x4
-	-	-	5	4x4
0.457404	0.334841	0.207755	2	1x2
0.571355	0.348744	0.0798978	2	2x2
0.617684	0.343413	0.0363597	2	2x3
0.671319	0.317899	0	2	3x3
0.720518	0.264849	0.00335683	2	3x4
-	-	-	2	4x4

B.2.2 How Optimal is the Greedy Strategy?

The table below shows the win probabilities for the close-to-optimal strategy, the greedy strategy and the random strategy. It also shows the total number of game states in which you can execute a move, and the fraction of those states compared to all possible states. This is shown for several variations in board size and maximum number of dice.

Optimal	Greedy	Random	Move	Fraction of Total	Dice	Size
0.562088	0.560606	0.520434	140	0.7	5	1x2
0.587457	0.571895	0.561298	18040	0.902	5	2x2
0.593854	0.571531	0.560918	1933600	0.9668	5	2x3
-	-	-	-	-	5	3x3
-	-	-	-	-	5	3x4
-	-	-	-	-	5	4x4
0.531846	0.503339	0.475694	20	0.625	2	1x2
0.642072	0.618661	0.576064	412	0.804688	2	2x2
0.66309	0.642069	0.603897	7300	0.891113	2	2x3
0.690821	0.673406	0.637657	500696	0.955002	2	3x3
-	-	-	-	-	2	3x4
-	-	-	-	-	2	4x4