BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ARTIFICIAL INTELLIGENCE FOR A BOARD GAME

UMĚLÁ INTELIGENCE PRO DESKOVOU HRU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DOMINIK TUREČEK**
AUTOR PRÁCE

SUPERVISOR **Ing. KAREL BENEŠ**
VEDOUCÍ PRÁCE

BRNO 2018

Zadání bakalářské práce/20290/2017/xturec06

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Tureček Dominik**
Obor: Informační technologie
Téma: **Umělá inteligence pro deskovou hru**
Artificial Intelligence for a Board Game
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s hrou Válka kostek a analyzujte její pravidla
2. Navrhněte a implementujte ji jako open-source
3. Navrhněte postup pro hodnocení (umělointeligentních) hráčů
4. Vytvořte několik umělých hráčů řízených pravidly
5. Navrhněte a implementujte učící algoritmus pro hraní této hry
6. Zhodnoťte výsledky a chování jednotlivých hráčů

Literatura:

- podle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Beneš Karel, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2
L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstract

This work proposes and implements AI agents for the game Dice Wars. Dice Wars is turn-based, zero-sum game with non-deterministic move results. Several AI agents were created using rule-based approach, expectiminimax algorithm, and logistic regression. To evaluate the performance of proposed agents, an implementation of the game was created. Results of the experiments have shown that it's preferable to play aggressively in two-player games and make more optimal moves in games played with more players. The agent using expectiminimax is able to win more than 60 % of games in 8-player games against random players and wins 21.4 % of games played against a mix of seven other agents created in this work. In two-player setups, the agent using logistic regression with numbers of players' scores and number of dice as features has the best performance and wins 59.4 % of games in average.

Abstrakt

Tato práce navrhuje hráče ovládané umělou inteligencí pro hru Dice Wars. Dice Wars je nedeterministická tahová hra s nulovým součtem. Bylo vytvořeno několik AI hráčů s využitím pravidlového přístupu, algoritmu expecitminmax a logistické regrese. Pro zhodnocení kvality navržených AI hráčů byla vytvořena implementace hry Dice Wars. Z výsledků experimentů vyplývá, že ve hře dvou hráčů je výhodnější hrát agresivněji než v případě vícehráčových her. Ve hře osmi hráčů vyhrává AI využívající expectiminimax přes 60 % her proti náhodným hráčům a 21.4 % her proti ostatním navrženým AI. Ve hrách dvou hráčů dosahuje nejlepších výsledků AI založená na logistické regresi, která jako příznaky používá skóre a počty kostek jednotlivých hráčů. V průměru vyhrává 59.4 % her.

Keywords

Dice Wars, Artificial Intelligence, Discrete Stochastic Games, Expectiminimax, Logistic Regression

Klíčová slova

Dice Wars, umělá inteligence, diskrétní stochastické hry, expectiminimax, logistická regrese

Reference

TUREČEK, Dominik. *Artificial Intelligence for a Board Game.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Karel Beneš

Rozšířený abstrakt

Cílem této práce bylo vytvořit hráče ovládaného umělou inteligencí pro hru Dice Wars. Dice Wars je nedeterministická tahová hra s nulovým součtem, která se hraje ve dvou až osmi hráčích na náhodně vygenerované herní ploše. Herní plocha je rozdělena do polí, kde každé pole je vždy ovládáno jedním z hráčů. Cílem hry je dobytí všech herních polí.

Každé pole obsahuje jednu až osm kostek, které reprezentují jeho sílu. Hráč ve svém kole může provést libovolný počet útoků na sousední pole v případě, že síla pole, ze kterého útočí, je větší než jedna.

Při boji útočník i obránce hází kostkami z polí účastnících se boje. Pokud útočníkovi na kostkách padne celkový součet vyšší než obránci, boj vyhrává. V takovém případě obránce přijde o své pole a jeho kostky z tohoto pole jsou odstraněny ze hry. Útočník poté přesune všechny kostky kromě jedné z pole, ze kterého útočil, na dobyté pole.

V opačném případě, kdy útočníkovi padne celkový součet stejný nebo nižší než obránci, je útok neúspěšný a všechny útočníkovy kostky kromě jedné jsou z jeho pole odstraněny. Když hráč již nemůže nebo nechce provádět další tahy, může své kolo ukončit. Poté jsou na jeho pole náhodně rozděleny kostky v takovém počtu, kolik polí je v jeho největším souvislém regionu (tomuto počtu se říká skóre hráče).

V této práci bylo navrženo několik strategií pro AI hráče. První z nich je Strength Difference Checking (SDC), která preferuje takové tahy, ve kterých má nad soupeřem největší převahu v počtu kostek. Pokud existují pouze takové možné tahy, kde mají převahu soupeři, SDC své kolo ukončí.

Single Turn Expectiminimax (STE) je strategie, která si vybírá tahy maximalizující odhadovaný počet polí, které bude hráč ovládat na začátku svého následujícího kola. K výběru využívá odhad pravděpodobnosti, že se podaří dané pole dobýt a zároveň nebude během kola protihráčů dobyto zpět. Vylepšené STE (STEi) poté optimalizuje minimální hodnotu odhadnuté pravděpodobnosti, kdy je ještě výhodné tah provést, a navíc preferuje tahy vycházející z největšího regionu hráče.

Win Probability Maximization (WPM) využívá logistické regrese pro odhad pravděpodobnosti výhry v daném stavu. Jako příznaky používá buďto skóre jednotlivých hráčů, logaritmy počtu kostek jednotlivých hráčů, nebo kombinované logaritmy skóre a logaritmy kostek. Při samotné hře WPM porovnává odhadovanou pravděpodobnost výhry pro možné tahy a vybírá ty nejvýhodnější.

Pro vyhodnocení kvality jednotlivých strategií byla vytvořena klient-server implementace hry Dice Wars, kterou bude možno v budoucnu použít pro případné přidání dalších variant AI hráčů.

Pro různé počty hráčů byly provedeny série experimentů v různých kombinacích AI hráčů. Z těchto experimentů vyplývá, že ve hře dvou hráčů je výhodnější hrát více agresivně, kdežto ve vícehráčových hrách tento přístup nevede k nejlepším výsledkům.

Ve dvouhráčových hrách mají převahu WPM a SDC jakožto hráči s vysokou agresivitou, přičemž WPM s kombinovanými příznaky logaritmů skóre a kostek poráží všechny ostatní hráče.

Ve vícehráčových hrách naopak nejlepších výsledků dosahuje STEi – proti sedmi náhodným hráčům vyhrává přes 60 % her a v osmihráčové hře proti všem ostatním variantám vyhrává přes 21 % her.

Artificial Intelligence for a Board Game

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Karel Beneš. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Dominik Tureček
May 15, 2018

Acknowledgements

I would like to thank my supervisor Karel Beneš for his guidance and all the valuable suggestions during the work on this thesis.

Contents

Chapter 1

Introduction

Games, in general, are a popular target for both analysis [4] and developing artificially intelligent agents [15]. In the case of AI development, there are two possible approaches – performing a game's state space search [6] and machine learning [15].

The first approach was used for games such as checkers [8], hnefatafl [10] or chess in IBM's Deep Blue [2]. The latter achieved a great success against human opponents as it used alpha-beta search to defeat the world chess champion Garri Kasparov [3].

The second approach using reinforcement learning was applied to checkers [7] and backgammon in TD-Gammon algorithm [13], with the latter being able to oppose world-class human opponents and it had even shown that there are some valid strategies that were previously thought unfavourable [12].

An example of a successful combination of the two approaches is the AlphaGo algorithm for the game of Go that is able to defeat other Go artificial players consistently and has defeated a European Go champion [9].

This work was inspired by the achievements of the mentioned algorithms and aims to create AI agents for the game of Dice Wars and evaluate their performance. Dice Wars is a zero-sum, turn-based, non-deterministic game played on randomly generated board divided into territories with the objective being to control all territories. There is no hidden information – each of the 2–8 players knows complete game state at any given time. As the number of possible moves each turn is quite large and their results are stochastic, it would not be very useful to examine whole state tree for a game. For this reason, a rule-based strategy, a set of two-ply search strategies, and a set of strategies using logistic regression were proposed.

The game of Dice Wars is described in Chapter 2 including the game rules and existing implementations of the game. In Chapter 3, theoretical concepts – used in this work choosing optimal moves and for estimating a probability of move results – are discussed.

Implementation of the application created as a part of this work is covered in Chapter 4 including both the client and server side and the communication protocol that is used.

Chapter 5 describes strategies adopted by the AI agents that were proposed in this work. Finally, experiments conducted with the developed AI agents are described in Chapter 6, as well as discussing their results and evaluating the individual AIs.

Chapter 2

Dice Wars

Dice Wars is a strategy game where players take turns to attack adjacent territories to expand their area. Each territory contains a number of dice determining player's presence and strength. The objective of the game is to conquer all territories and thus eliminate each opponent.

2.1 Rules

Dice Wars is played with two to eight players. The game board is randomly generated using a seed-like algorithm and consists of 29–31 *territories*. Each territory is controlled by a single player at any given time. The number of dice in a territory represents how much power a player holds over it and is referred to as the territory's *strength*. Strength of a territory can neither be lower than one nor higher than eight at any given time. A group of adjacent territories controlled by a single player is called a *region*. Player's *score* is the number of territories in his largest region. Figure 2.1a shows an example of a region as well as territories' strengths.

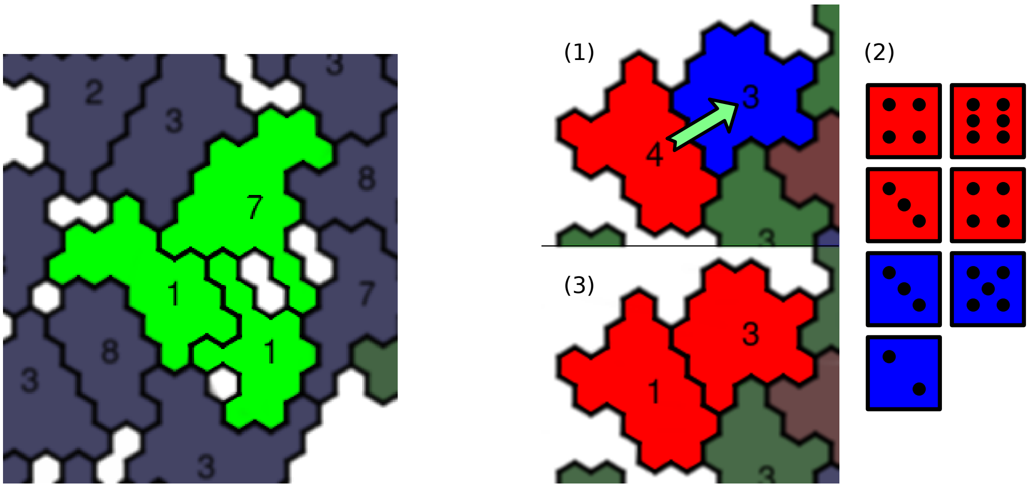
At the start of the game, territories are randomly assigned to individual players. Then, for each player, a predefined number of dice – proportional to the total number of territories – is distributed. Lastly, the player order is determined randomly.

On each turn, a player has the option to attack as many territories held by his opponents as he wants, provided that the following two rules are satisfied:

- To attack an enemy territory, it must be adjacent to a player's territory from which the attack is initiated.
- The territory from which the attack is initiated has a strength of two or more.

Whenever a player cannot or doesn't wish to make another attack, he declares that his turn has ended.

At the end of a turn, a number of dice equal to the player's score is distributed randomly across all territories he controls. In the case some dice cannot be placed, they are stored in player's *reserve*. If there are dice in a player's reserve at the end of his turn, they are added to the dice being distributed, up to the limit of sixty-four.



(a) Region.

(b) A battle.

Figure 2.1: a) A region controlled by a green player. The region consists of three territories with strengths of seven, one and one. Green player's score is three. b) 1) Red player chooses to attack the blue territory. 2) Red rolls four dice whereas blue rolls three. Red rolls higher total than blue, so he is the winner. 3) As a winner, red places three of his dice into the conquered territory and leaving one die in the territory from which the attack was initiated. Blue player loses all dice from the attacked territory.

2.1.1 Battle

When a player makes an attack, a *battle* occurs. To determine the result of a battle, both players roll a number of dice corresponding to the strength of their territory involved in the battle. The player with higher roll is the winner. In case of a tie, the defender wins.

If the attacker wins the battle, he gains control of the attacked territory. Defender's dice from the territory are removed entirely and the attacker moves there all dice from the territory from which he attacked except one.

In the case the attacker loses the battle, the number of dice in the territory from which he attacked is reduced to one and nothing happens to the attacked territory. Example of a battle's resolution is shown in Figure 2.1b.

2.2 Existing Implementations

This work is based on Taro Ito's implementation of Dice Wars[1]. Figure 2.2 shows a picture of the game. However, as that implementation doesn't offer an opportunity to create new AI agents and neither does it provide its source codes, an implementation of the game was created as a part of this work that tries to match the rules and logic of Ito's version as closely as possible.

However, there are other variants of this game, the most notable being the board game Risk from 1957. While having the same core mechanic of holding territories and combat resolution using dice corresponding to the territories' strength, it is of slightly higher complexity and offers more control in terms of initial setup and strength distribution at the end

---

[1] `http://www.gamedesign.jp/flash/dice/dice.html`

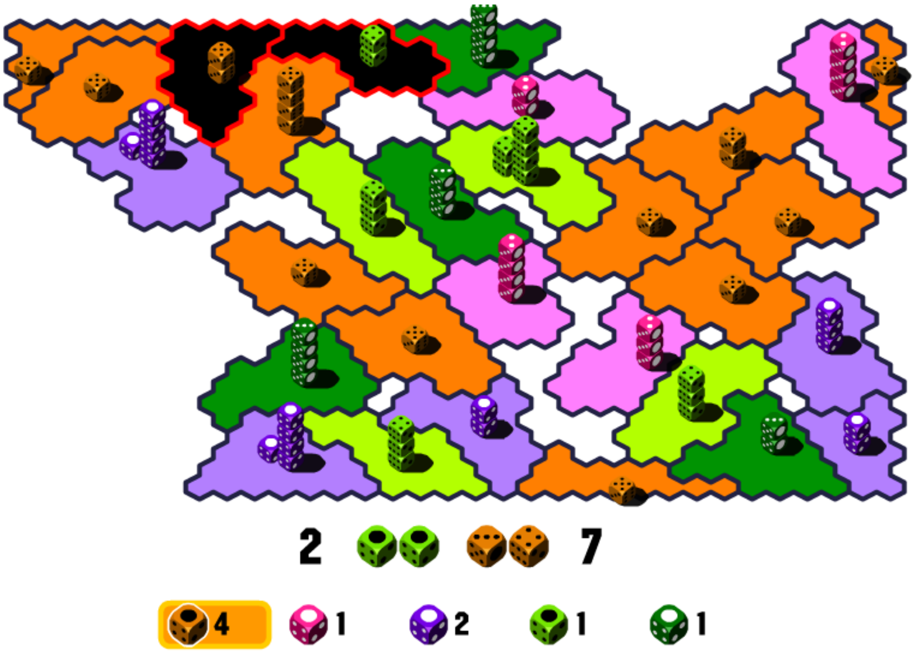


Figure 2.2: Taro Ito's implementation of Dice Wars.

of each turn, as well as using cards gained by conquering territories to gain an advantage. In addition, the number of dice rolled in Risk's combat is inferred from territory's strength and is not equal to it.

The game was released in many versions since its original creation. Risk's rules are described in more detail in Tan [11], including an analysis of probabilities of possible moves in the game [4].

Chapter 3

Concepts

This chapter describes theoretical concepts that were used for the AI agents development. First, a probability of a success in battle is analyzed. Then, the expectiminimax algorithm is introduced. Finally, linear logistic regression as a means to estimate win probability associated with a move is described.

3.1 Probability of a Successful Attack

Let $P_{A\to D}$ be the probability of a successful attack from territory A with strength a to territory D with strength d. Then the probability can be formally described as follows:

$$P_{A\to D} = \sum_{i=a}^{6a} P(i|a)\left(\sum_{j=d}^{i-1} P(j|d)\right), \tag{3.1}$$

where $P(i|a)$ is a probability of rolling a sum of i on a dice, and similarly $P(j|d)$ is a probability of rolling j on d dice.

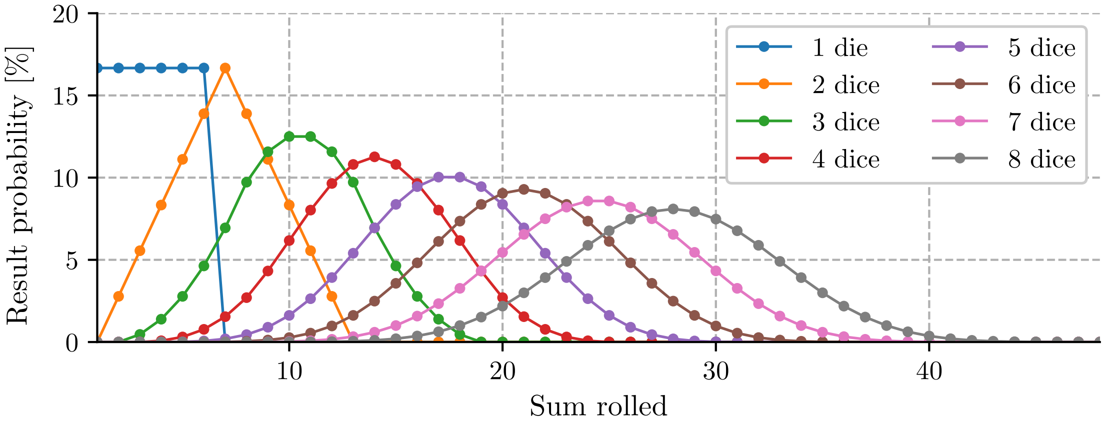


Figure 3.1: Probability distributions of the sums rolled on one to eight dice.

This probability $P(i|a)$ has following characteristics – it is always zero for $i < a$ and $i > 6a$, as a number lower than n or higher than $6n$ cannot be rolled on n dice. The

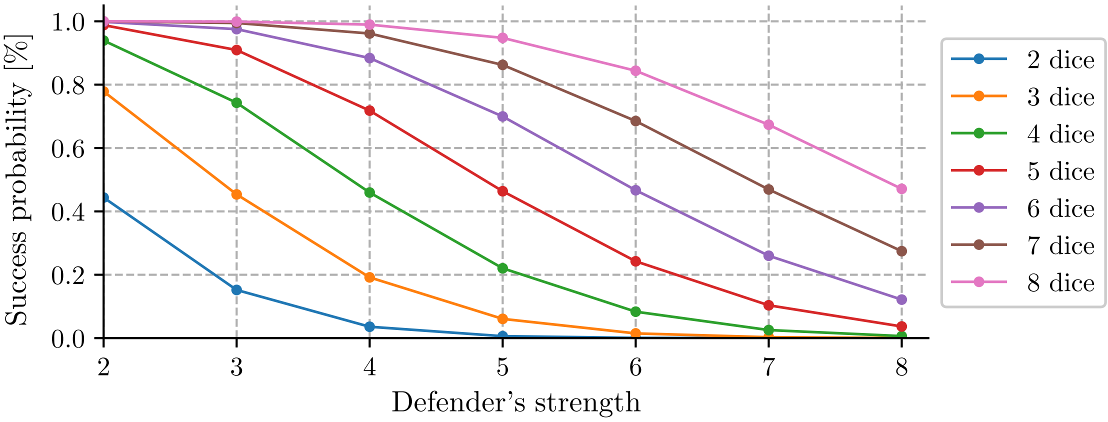


Figure 3.2: Probabilities of attack success $P_{A \rightarrow D}$ when attacking from territory A to D for all combinations of the attacker's and defender's strengths.

probability distribution is symmetrical and with increasing number of dice approaches the normal distribution. Probability distributions of $P(i|a)$ for 1–8 dice are shown in Figure 3.1.

Formally, $P(i|a)$ for all combinations of i and a follows the multinomial distribution. However, as its formal description isn't needed for this game as maximum a is limited to eight, a numerical solution of $P_{A \rightarrow D}$ is used instead. The calculated values of $P_{A \rightarrow D}$ for all combinations of A and D can be seen in Figure 3.2.

3.2 ExpectiMiniMax

Expectiminimax algorithm expands on minimax – the difference between the two is that in minimax, outcomes of all actions are deterministic, whereas in expectiminax they are stochastic [6].

Minimax uses a tree structure to describe the game, including all states of the game from its start to the end. The tree is defined by an initial state, a set of possible moves in a given state and function describing results of each move.

As an example, we use a two-player game with players called Max and Min as shown in Figure 3.3. Max is the starting player, and Max and Min alternate each turn. The minimax tree consists of nodes representing game states between turns and edges representing moves. The initial node from Max's turn has a set of child nodes belonging to Min which correspond to the game state after Max has made his move. This way, the levels (with each level being called a *ply*) alternate for Max's and Min's nodes. Each node has a value that describes how useful a state is for a player in terms of winning the game. Leaf nodes represent the terminal state when the game ended an have a value (called utility) of 1 if Max has won, -1 if Min has won, or 0 in case of a tie. Values of all non-leaf nodes can be any real number from the interval $(-1, 1)$.

Each turn, Max chooses the best possible move – this is move with the highest value, possibly leading to a leaf node with utility 1 – the win state. However, we then assume that Min will choose the best move for him with the lowest utility. Max thus has to take Min's possible choices into account.

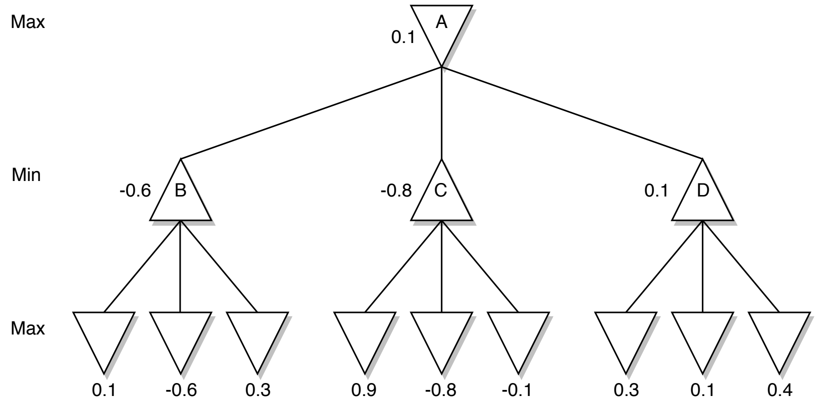


Figure 3.3: Example of a three-ply minimax tree. Max is the current player (node A) and has three options - B, C, and D. However, Min will choose such move from one of these nodes that has the lowest minimax value – this is -0.6, -0.8 and 0.1 for B, C, and D respectively. Max will then choose the best option – node D. Thus, the node A has a minimax value of 0.1.

To calculate the utility for a state s, the *minimax* function is used:

$$\text{minimax}(s) = \begin{cases} \text{utility}(s) & \text{if terminal}(s) \\ \max \text{minimax}(\text{result}(s,a)) & \text{if player}(s) = \text{Max} \\ \min \text{minimax}(\text{result}(s,a)) & \text{if player}(s) = \text{Min} \end{cases}$$

where *utility* of a terminal state is 1, -1 or 0, *player(s)* is the player whose turn it is in a state s, and *result(s, a)* returns a state of the game after a move a is taken in a state s.

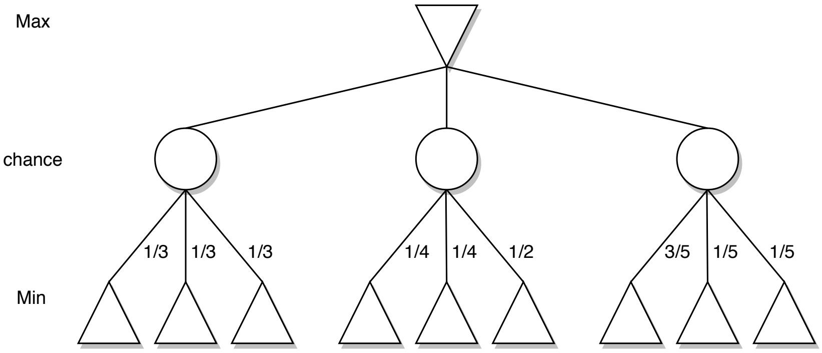


Figure 3.4: Expectiminimax has a *chance* level after each Max's and Min's level that represents non-deterministic results of each move.

However, in minimax algorithm, we possibly know all of the states in the game, and all moves are deterministic. In Dice Wars, on the other hand, outcomes of all moves are non-deterministic. For this reason, we use expectiminimax instead. The difference from minimax is that there is a new level of *chance* nodes between each Max's and Min's turn. These chance nodes represent the uncertainty of the actions.

With this in mind, the *expectiminimax* function can be defined as follows:

$$\text{expectiminimax}(s) = \begin{cases} \text{utility}(s) & \text{if terminal}(s) \\ \max \text{expectiminimax}(\text{result}(s, a)) & \text{if player}(s) = \text{Max} \\ \min \text{expectiminimax}(\text{result}(s, a)) & \text{if player}(s) = \text{Min} \\ \sum_r P(r)\text{expectiminimax}(\text{result}(s, r)) & \text{if player}(s) = \text{chance} \end{cases}$$

where r is a random event and *result(s, r)* is the state s with a specific result of event r.

Expectiminimax is used as a way to find optimal moves in AI strategy described in Section 5.3.

3.3 Logistic Regression

Linear logistic regression is used to classify data into multiple classes by modeling the probability that a given feature vector belongs to one of the classes [5]. Binary logistic regression classifies two-class data and thus can be used as a way to predict probability to win the game based on the feature describing the game state – with the two classes being either win or loss of the game.

To estimate the probability of a feature vector to belonging to a class, logistic sigmoid is used in this method:

$$\sigma(a) = \frac{1}{1 + e^{-a}}. \tag{3.2}$$

Calculation of the probability $p(C_1|\mathbf{x})$ that a given feature vector $\mathbf{x}$ belongs to class C_1 can be formally described as follows:

$$p(C_1|\mathbf{x}) = y(\mathbf{x}) = \sigma(w_0 + \mathbf{w}^T\mathbf{x}), \tag{3.3}$$

where w, $\mathbf{w}$ are coefficients of the regression. The probability for the second class is then simply $p(C_2|\mathbf{x}) = 1 - p(C_1|\mathbf{x})$.

To use the logistic regression, its $\mathbf{w}$ coefficients must be estimated from training data. For this estimation, a maximum likelihood method is used to find such values of $\mathbf{w}$ that for each $\mathbf{x}$ from the training set, the predicted probability will be as close to its associated value 1 or 0 [1]. Formally, maximum likelihood method attempts to find optimal $\mathbf{w}$ to maximize the likelihood function:

$$p(\mathbf{t}|\mathbf{X}) = \prod_{i \in C_1} y(\mathbf{x}_i) \prod_{i \in C_2} \left(1 - y(\mathbf{x}_i)\right), \tag{3.4}$$

where elements of $\mathbf{t}$ indicate the class associated with a feature vector $\mathbf{x}_i$ from $\mathbf{X}$.

Let $t_i = 1$ for each $\mathbf{x}_i$ belonging to class C_1, and $t_i = 0$ for each $\mathbf{x}_i$ belonging to class C_2. Then we can simplify Equation 3.4 to:

$$p(\mathbf{t}|\mathbf{X}) = \prod_i y(\mathbf{x}_i)^{t_i} \left(1 - y(\mathbf{x}_i)\right)^{1-t_i}. \tag{3.5}$$

To find the maximum of the likelihood function, a negative logarithm of Eq. 3.5 called cross-entropy is used:

$$E(\mathbf{x}) = -\ln p(\mathbf{t}|\mathbf{x}) = -\sum_{i=1}^{N} \left\{ t_i \ln y(\mathbf{x}_i) + (1 - t_i) \ln \left(1 - y(\mathbf{x}_i)\right) \right\}. \tag{3.6}$$

Then such **w** coefficients have to be found that the error function would be minimal. To do so, a gradient of the error function in respect to **w** is used:

$$\nabla E(\mathbf{x}) = \sum_{i=1}^{N} (y(\mathbf{x}_i) - t_i). \tag{3.7}$$

It was shown that the error function is convex and has a unique minimum [1]. Therefore, as the gradient shows the direction in which the error function grows the most steeply, the minimum of the error function can be found by finding such **w** where the gradient would be zero.

These can be found using gradient descent method [1]. This is an iterative method where the gradient is calculated and a small step is made in the direction of highest decrease of the error function. These steps are repeated until a minimum is found or the values of the gradient get reasonably small. A single step of gradient descent can be described formally as follows:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}), \tag{3.8}$$

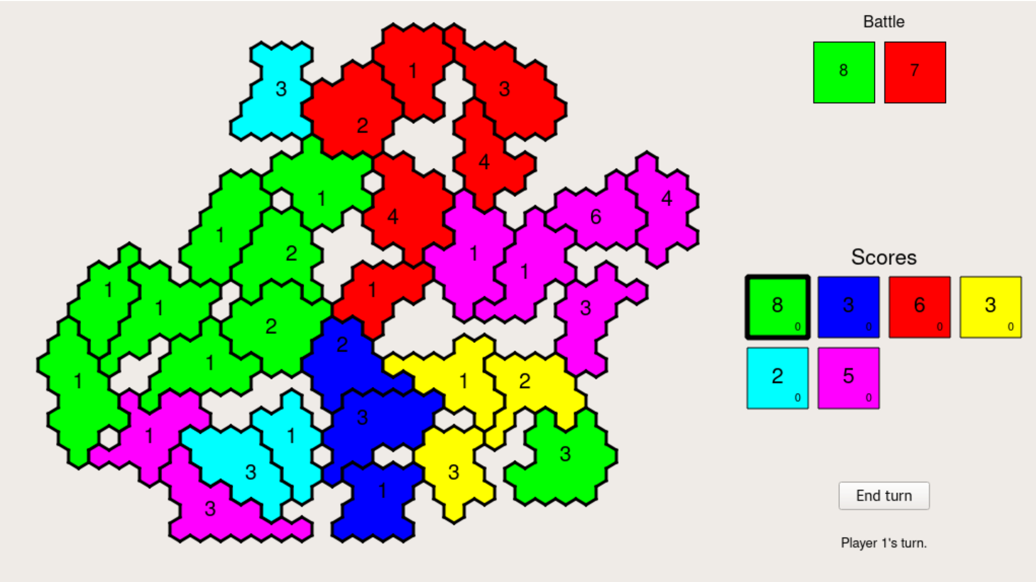
where $\tau > 0$ is the learning rate.

There is a risk of over-fitting using maximum likelihood when the two classes would be completely linearly separable. This would cause the logistic function to be infinitely steep. However, as shown on gathered data in Section 6.4, this would not be the case in this work as the data are impossible to separate linearly.

Chapter 4

Game Implementation

For a purpose of conducting experiments with AI strategies proposed in Chapter 5, the game of Dice Wars was implemented as a part of this work[1]. The resulting implementation is divided into two modules. The server provides game board generation, handles game logic, and communicates with players using their clients. The client then gathers player input, sends queries to the server with players' actions, and draws the game current state. This way, the server can be used for future experiments with possible AIs without the need of implementing the game's logic. The application is written in Python 3.6[2]. Figure 4.1 shows the GUI of the application.



Player 1's turn.

Figure 4.1: Graphical interface of Dice Wars. Most of the screen is taken by a game board divided into territories. Each territory has a strength represented by the number and an owner by the territory's color. On the top right, there is information about last battle, with the numbers shown representing dice rolls by both the attacker and the defender. Under the battle information are shown players' scores including the number of dice in each player's reserve.

---

[1] Source code can be downloaded from https://github.com/dturecek/dicewars.

[2] https://www.python.org/downloads/release/python-363/

4.1 Server

The server consists primarily of four classes representing the game state – `Game`, `Board`, `Area` and `Player`, and an instance of class `Generator` that generates the board at the start of the game. An instance of `Game` class holds information about the application state, including its IP address and port number, number of players and instances of the `Player` class that represents them, and an instance of the `Board` class. The `Board` class represents the game board and holds an instance of the `Area` class for each territory in the game. Simplified UML class diagram of the module is shown in Figure 4.2.

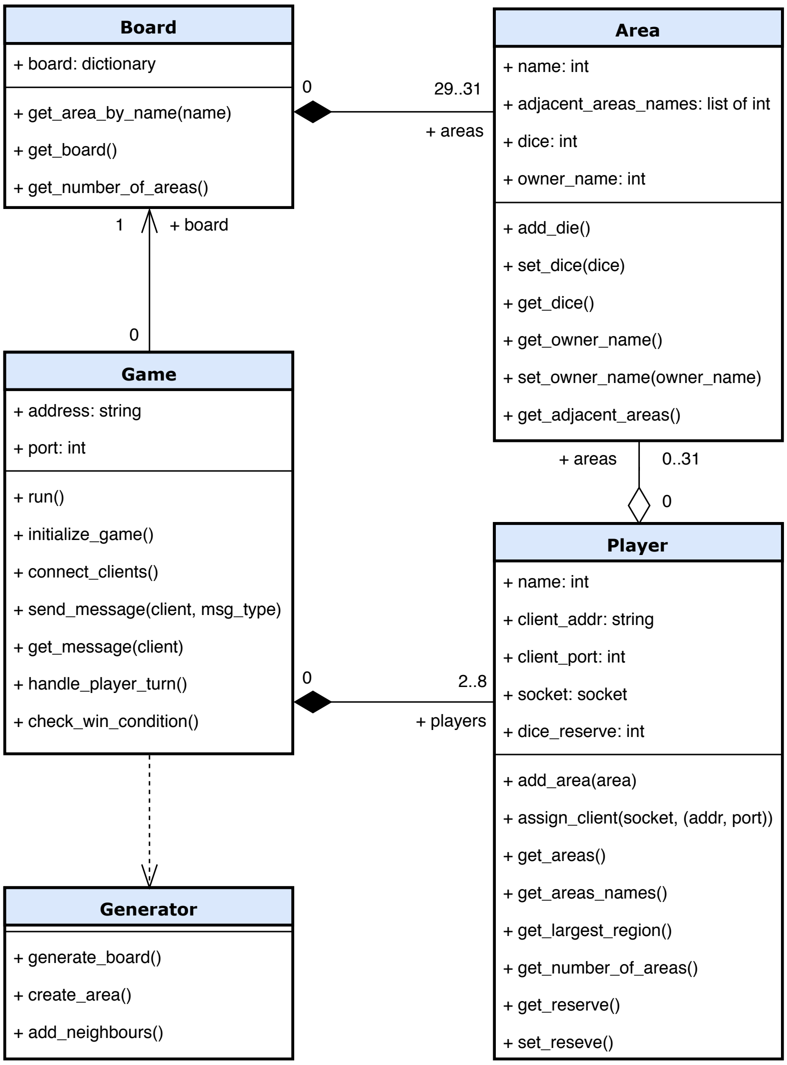


Figure 4.2: Simplified UML class diagram of the server module. Communication with clients and most of the game's logic is handled by the Game class. Game board is represented by the Board class and generated by the Generator class. Board territories are represented by the Area class. The Player class is used for holding the information about a player as well as the client controlling it.

When the server is started, an instance of the `Game` class is created with attributes of the number of players, IP address and port number passed from the command line. During the instantiation, an instance of the `Board` class is created as one of the Game's attributes, as well as a list of the `Player` class instances. Territories and dice are distributed to players, and the player order is determined. Then, the `Game.run()` method is called.

In the `Game.run()`, the server first waits for all players' clients to connect. Whenever a client connects, it is assigned to one instance of the `Player` class. After all clients are connected, the server periodically calls the `handle_players_turn()` method, where the player input from clients is processed, after which the end game condition is checked.

Whenever a client sends a query with a player's action, the server parses such query and performs the action. There are two possible actions: a battle and ending a turn. When the action is resolved, all clients are sent information about current game state. Client-server communication protocol is described in Section 4.3.

4.2 Client

The module mirrors the classes in the server module in terms of the represented objects. However, it lacks the `Generator` class and has additional classes which are used depending on the mode used – AI or human controlled.

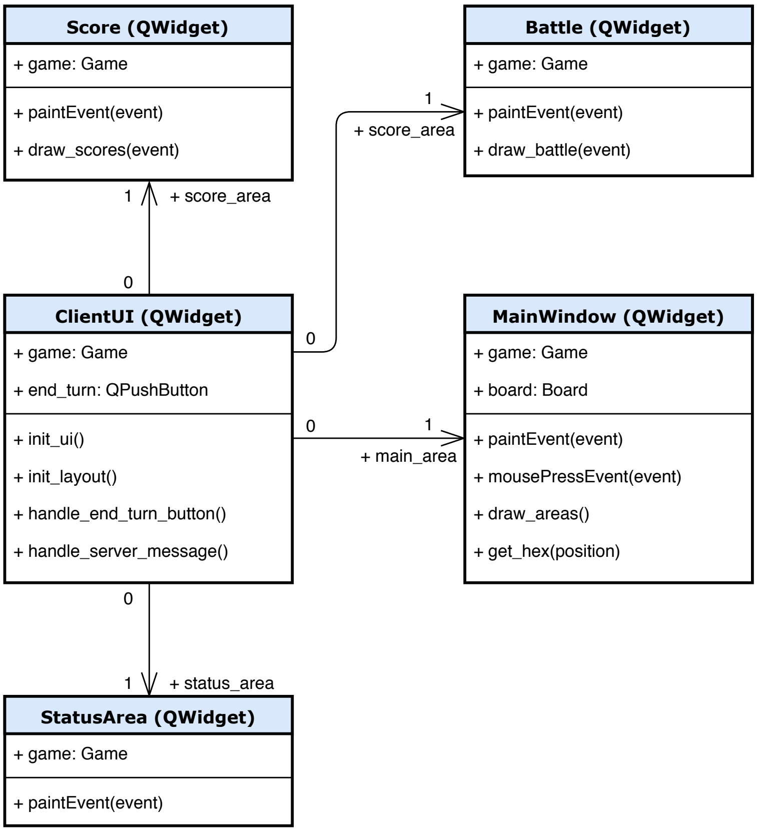


Figure 4.3: Simplified UML class diagram of client's user interface implementation. Each class inherits from `QtWidget`'s `QWidget` class. For the sake of simplicity, this is shown just as a note in parentheses instead of the conventional way of showing inheritance. `MainWindow` class shows the game board, `Battle` is used to display battle results, `Score` shows players' scores and reserves, and `StatusArea` shows current player.

When controlled by a human player, the client has a graphical user interface which uses PyQt[3], a Python binding for Qt[4] application framework. The GUI is implemented in class `ClientUI` using classes `MainWindow` for the game board, `Battle` for dice roll results of a battle, `Score` that shows scores as well as dice reserves of each player, and finally `StatusArea` that contains information about who is the current player. A simplified UML class diagram of GUI is shown in Figure 4.3.

A human controlled client uses a separate process to look for messages incoming from the server and gather them in a queue. This queue is checked periodically and game state is updated accordingly with changes described in the messages. After each such update, the application window is redrawn to show the current state.

On the other hand, a client controlled by an AI agent uses the `GenericAI` class. This class implements basic methods needed by all AI agents and serves as a basis for their creation.

4.3 Communication protocol

In Dice Wars, the server informs all clients about the state of the game after each action taken by one of the players. Clients send messages to the server only on their turn, informing it about actions they want to take.

There are the following five types of messages sent by the server:

- *Game start* – message sent at the start of the game to each client after it connects to the server. It contains information about each territory, such as the territories they are adjacent to, their owner and dice, as well as the positions of hexes they are composed of in the GUI. Next, the message specifies the number of players, player order, players' scores, the name of the current player and the player name assigned to the client the message is sent to.

- *Game state* – message sent after each client action. It contains information about the owner and dice for each territory, as well as the current player name.

- *Battle* – message sent after a battle occurs. The content of the message is the updated state of affected territories (number of dice, owner) and the results of dice rolled in the battle.

- *End turn* – message sent after a client ends its turn. It describes the territories that changed during the turn, as well as the name of the new current player and players' dice reserves.

- *Game end* – after the game ends, the server sends a game end message to inform the clients about the winner.

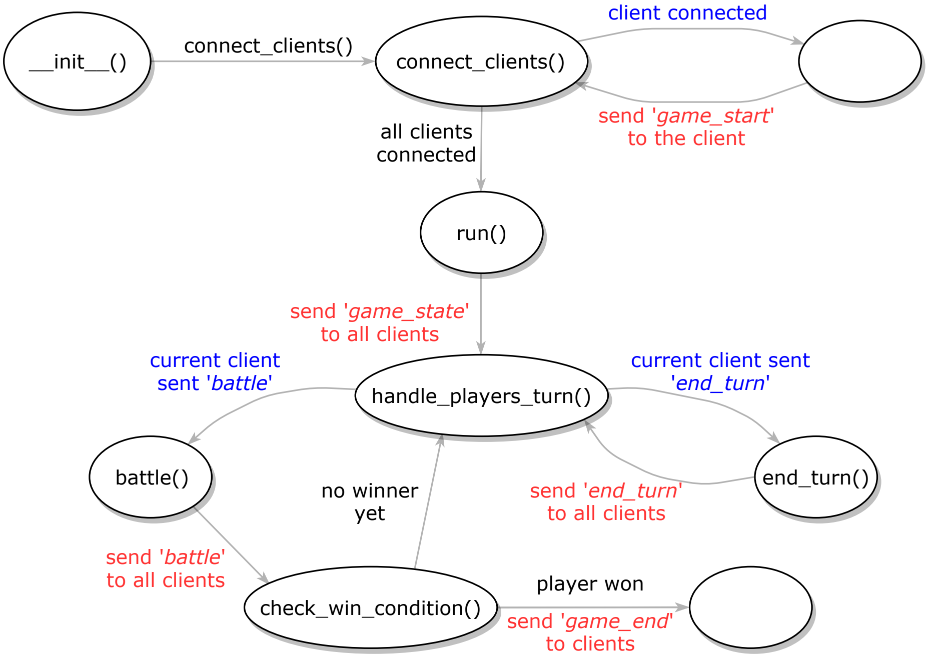
- *Invalid move* – message sent to a client when it attempts to make an invalid move.

In contrast to the server, the client only sends two types of messages. The first is *battle* which specifies the attacking and defending territories, and the second is *end turn* to declare that the client doesn't wish to make any more moves.

---

[3] https://www.riverbankcomputing.com/

[4] https://www.qt.io/



__init__()
connect_clients()
connect_clients()
client connected
send '*game_start*' to the client
all clients connected
run()
send '*game_state*' to all clients
current client sent '*battle*'
handle_players_turn()
current client sent '*end_turn*'
battle()
no winner yet
send '*end_turn*' to all clients
end_turn()
send '*battle*' to all clients
check_win_condition()
player won
send '*game_end*' to clients

Figure 4.4: Diagram of client-server communication from server's point of view. Server actions are labeled in red, whereas client actions are labeled in blue. With the exception of the `connect_clients()` method, server always sends the messages to all clients playing the game.

The messages sent by the server are sent to each client with two exceptions – each client is sent its own game start message due to the player name it is assigned, and invalid move message is sent only to the currently playing client. Clients send messages only to the server.

At the start of the game, after a client connects, the server sends the game start message to it. During the game itself, the server sends the game state message at the start of each player's turn. The client then makes attacks that are declared by its battle message. The server then resolves the combat and sends results in server's battle message. After each battle, server checks the win condition and if a player wins the game, it lets the clients know about the winner.

When a client doesn't want to make any more moves, it sends the end turn message. The server then informs all the clients about the new current player and the players' reserves. Diagram of the client-server communication protocol from the server's point of view is shown in Figure 4.4.

In case a client sends a message to the server during other client's turn, the message is ignored. If a client attempts to make an invalid move, it is sent the *invalid move* message.

Dice Wars uses JSON[5] for encoding its client-server communication.

---

[5] https://www.json.org/

Chapter 5

Strategies

In this chapter, game strategies adopted by developed AI players are described.

5.1 Naïve Player

This strategy is the simplest possible. Each turn, the naïve player iterates over all his territories in random order. For each one that has strength higher than one, he checks all adjacent territories and if any of them belongs to another player, it is picked as a target for an attack. After the attack, the player starts over. The player ends his turn if and only if no further attack can be made.

This strategy will serve as a baseline to assess the performance of the other AIs.

5.2 Strength Difference Checking

This strategy attempts to make reasonable attacks. It judges the strength difference (SD) of the attacker and the defender for every possible attack. Then it makes the one with the most favourable SD.

If there is no possible move with strength difference higher or equal to zero, the player using this strategy ends its turn.

5.3 Single Turn Expectiminimax

This variant uses a two-ply expectiminimax algorithm (Section 3.2 to estimate game state after a single opponent's turn to choose optimal move. As the utility function, it uses the number of held territories at a start of player's turn. When looking for a possible move, the player searches for such that has the highest chance to increase this number. To achieve this, moves with the highest probability of conquering a territory and holding it over a next player's turn are prioritized.

Let $P_{A \to D}$ be a probability that a player attacking from territory A can conquer a defending territory D as described in Eq. 3.1. Then the estimated probability of successfully attacking and holding a territory over next player's turn can be formally described as follows:

$$P = P_{A \to D} \times \prod_{i=1}^{k} \Big(1 - P_{N_i \to D}\Big), \tag{5.1}$$

where N_i are neighbours of territory D that are controlled by an opposing player.

However, this is just an approximation, as this doesn't take into account the possibility that D is taken from a territory held by the current player at the time of making the calculation.

In addition, this approximation is limited because it treats individual territories as independent and omits any relation between them. For example, in games played with more than two players, one of the neighbours of D might be taken over by an enemy with higher strength.

From all possible moves with a calculated probability higher than 20 %, the highest one is chosen. If no such move is found, the agent looks if there is a possible move from a territory containing eight dice. This is done to prevent a situation where no player is willing to make a move. Otherwise, the agent using this strategy ends its turn.

5.3.1 Improved STE

As the probability threshold of 20 % might not be ideal for each number of players in a game, a set of experiments were carried out to measure the effectivity of various values over multiple game configurations. The most advantageous values were then chosen for the final AI agent.

In addition, it might not be optimal to make attacks from territories not belonging to the player's largest region, as the number of dice distributed at the end of each turn depends only on the player's score. For this reason, attacks coming from the largest region are preferred over others. Each possible move is assigned a preference, where for the territories outside of the largest region it is simply the estimated probability from Eq. 5.1. For the territories inside the region, the preference is the probability multiplied by a preference constant that was chosen experimentally.

The experiments that lead to the choice of both the probability threshold of an attack and the preference constant are described in Section 6.3.

5.4 Win Probability Maximization

This strategy attempts to make such moves that maximize the estimated probability to win the game. It is based on logistic regression as described in Section 3.3. On each turn, an agent using this strategy estimates current win probability based on the state of the game using logistic function (Eq. 3.2). Then, for each possible move, a new probability is estimated, and the move with the highest improvement of win probability (if any) is chosen.

Three sets of features were chosen for this strategy, each of them implemented in a separate agent. The first set of features consists of players' scores, the second one is the total number of dice owned by individual players. The third combines both scores and dice, using their log values.

5.4.1 Player Scores as Features

This strategy uses players' scores as a feature to evaluate the probability of winning a game. A feature vector contains players' scores in the same order they play the game, with the first score being the player controlled by an agent using this strategy. An example of a feature vector in a four-player game might look like the following: $[7, 3, 5, 1]$.

An agent using this strategy starts each turn by evaluating the current probability of winning based on all players' scores. Then, for each move that would increase the agent's score, the improvement of the estimated probability to win is calculated as $I_p = \ln p_{i+1} - \ln p_i$. In addition to this, all territories with eight dice are considered as possible moves and their improvement is calculated as well. All the moves are then ordered with respect to the descending improvement. If the first move in the ordered list has a positive improvement or the attacking area has eight dice, it is chosen by the agent. Otherwise, the agent using this strategy ends its turn.

This strategy is formally described in Algorithm 1. In this description, following terms are being used – t is a territory, t_P is a set of all territories belonging to the current player, LR_P is current player's largest region and m_P is a list of possible moves for the current player in format $(t \to n, I)$, where t is the attacking territory, n is defending territory, and $I(t, n)$ is the improvement in win probability if the attack is successful. Next, $neighbours()$ method yields a list of all neighbouring territories, $dice()$ method returns the number of dice in a territory, and $order()$ method sorts the list of possible moves in descending order in respect to the improvement $I(t, n)$.

Algorithm 1 Choosing a move in logistic regression using players' scores.

```
Require: player's territories $t_P$, player's largest region $LR_P$, empty list of moves $m_P$
for $t \in t_P$ do
    for $n \in t.neighbours()$ do
        if $n \notin t_P$ then
            if $t \in LR_P$ then
                add move $(t \to n, I(t, n))$ to $m_P$
            else
                for $n' \in n.neighbours()$ do
                    if $n' \in LR_P$ then
                        add move $(t \to n, I(t, n))$ to $m_P$
                        break
$m_P.order()$
$(t \to n, I(t, n)) = m_P[0]$
if $I(t, n) > -0.05$ or $t.dice()$ is 8 then
    choose move $(t \to n, I(t, n))$
else
    end turn
```

5.4.2 Number of Dice as Features

This strategy uses players' total number of dice to estimate the probability of winning. A feature vector contains logarithms of individual players' dice in the same order they play the game, with the first score being the player controlled by an agent using this strategy. An example of a feature vector in a four-player game might look like the following: $[\ln(31+1)$, $\ln(96+1)$, $\ln(0+1)$, $\ln(19+1)]$[1]. The agent's player would in, this case, have 31 dice in total, and the players playing after him would have 96, 0 and 19 dice respectively. The

---

[1] One is added to each number of dice to avoid a situation of $\ln(0)$ in case a player was eliminated, thus having zero dice.

example shows that the strategy takes into a consideration even such situations where one or more players are eliminated.

The agent using this strategy starts each turn by evaluating the current probability of winning based on all players' dice. Then, ending the agent's turn is considered as a possible move that would increase the agent's dice by his score and the improvement for this is calculated. After this, for each possible move, the probability of winning p_w is estimated. If the probability of the attack success is p_a, it can be formally described as follows:

$$p_w = p_a p_{w+} + (1 - p_a) p_{w-}, \tag{5.2}$$

where p_{w+} and p_{w-} are the estimated probabilities of winning the game given the state after a successful and an unsuccessful attack respectively. The improvement over the estimated probability to win the game at the start of the turn is calculated. Both p_{w+} and p_{w-} take into consideration not only dice lost during the battle but also the dice that would be generated at the end of the turn.

The agent then chooses the move with the highest improvement with one exception – if the highest improvement would be associated with ending the turn and there is a possible move from a territory with eight dice, the move is chosen instead. This is again to prevent a situation where no player wants to make a move.

5.4.3 Combined Features of Player Scores and Dice

This strategy estimates the probability of winning a game using both player scores and dice. A feature vector contains logarithms of players' scores and dice in the order they play the game. A single feature vector in a two player game might look like this: $[ln(11+1)$, $ln(56+1)$, $ln(7+1)$, $ln(43+1)]$, where the current player would have a score of 11 and 56 dice and the second player would have a score of 7 and 43 dice.

Other than using this enhanced features, the way of choosing the move is the same as in logistic regression with the number of dice as features as described in the previous section.

Chapter 6

Experiments

This chapter describes results of experiments conducted using strategies proposed in Chapter 5. There are several metrics that we can measure when experimenting with games played by AI players. The obvious one is win rate.

However, the game gives an advantage for the player playing first, so win rate itself may be affected by the rate of which one is the first player – even though the advantage of playing first is lower in games played with a larger number of players. To demonstrate this, a set of ten thousand games for all numbers of players were played with naïve players only. Win rate of the first player from these games is shown in Figure 6.1. Because of this advantage, the rate at which a player wins despite playing second is used as a second metric to measure AI success in two-player games.

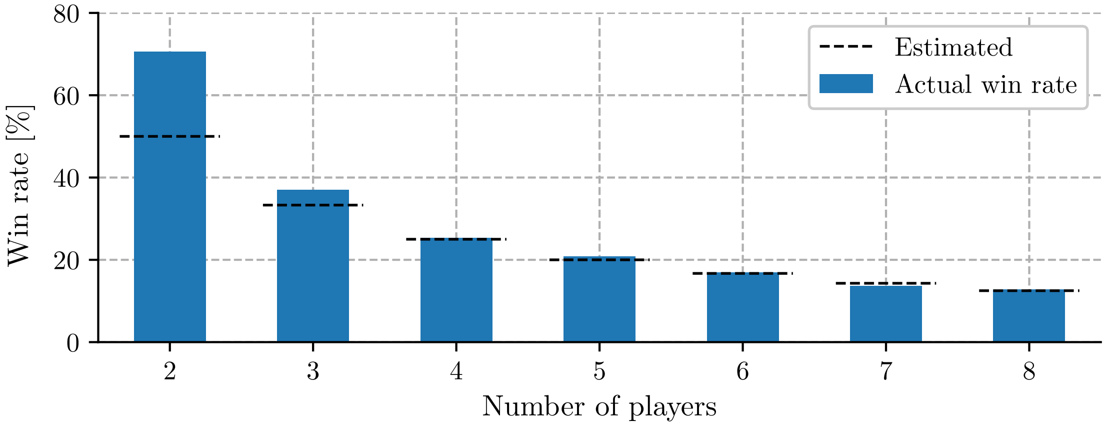


Figure 6.1: Win rate of the player starting first in games played by naïve players. The estimated win rate shown in the figure is the hypothetical win rate in case the starting order wouldn't matter. Whereas in two-player games the advantage leads to 20 % increase in win rate as opposed to the estimated one, in games played by four or more player the advantage disappears. Data shown were collected from 10 000 games for each number of players.

6.1 Naïve Player against Strength Difference Checking

Out of ten thousand two-player games played between the naïve player and the Strength Difference Checking player (SDC), a total of 6348 (or 63.4 %) were won by SDC. Table 6.1 shows data from these games. We can see that from all games that SDC started as a second player, it was able to win 40.4% of them, whereas the naïve player has won only 12.7 % of games in such situation.

Table 6.1: Data from 10 000 two-player games between naïve player and Strength Difference Checking (SDC).

	Naïve Player	**SDC**
Games won	36.5 %	63.4 %
Won starting second	12.7 %	40.4 %

For each possible combination of the two AI agents (eg. naïve-SDC-SDC and naïve-naïve-SDC for three-player games), a series of ten thousand multi-player games were played. From the collected data shown in Figure 6.2 can be seen that even with just the preference of highest strength difference, SDC is able to win significantly more games than the naïve player.

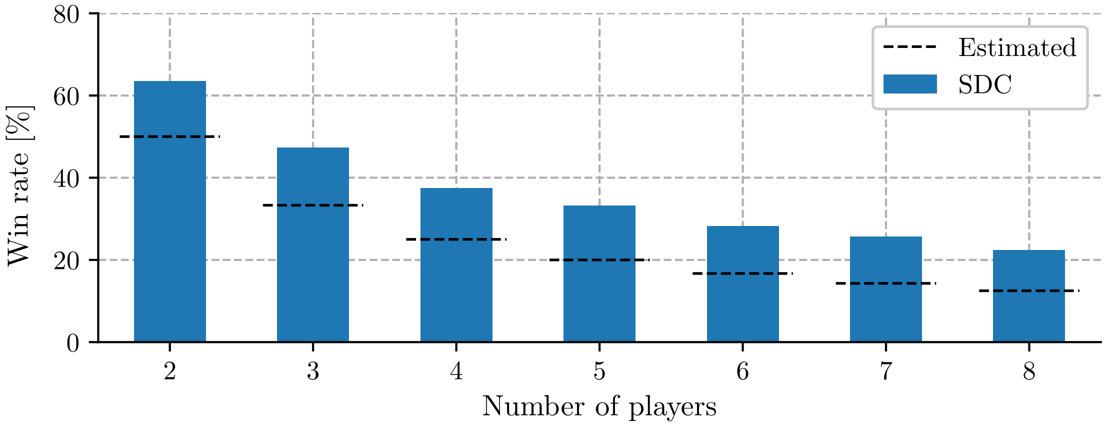


Figure 6.2: Win rate of SDC agent in games played against the naïve player. The data shown in the figure come from games where there was only one SDC and the rest were naïve ones. The estimated win rate shows the percentage of games that should be won in random games.

As the experiments described in this section took relatively long time to complete, as shown in Table 6.2, I examined how would the results differ if smaller sets of games were played in following experiments. The dependence of win rate with respect to the number of games played is shown in Figure 6.3 for two- and eight-player games with one SDC and one or seven naïve players respectively. Win rate in two-player games stabilizes fairly quickly. On the other hand, in eight-player games, win rate is still slightly rising when reaching 10 000 games. However, the difference between 1 000 and 10 000 games played is only 2.6 %. As a compromise between the time needed to run the experiments and the results accuracy, I chose a number of 1 000 games to be played for each configuration in following experiments.

Table 6.2: Average and total time needed to play 10 000 two-player games and the same number of eight-player games using the naïve player and SDC.

	2 players	**8 players**
Average time [s]	1.26	4.69
Total time [min]	209	781

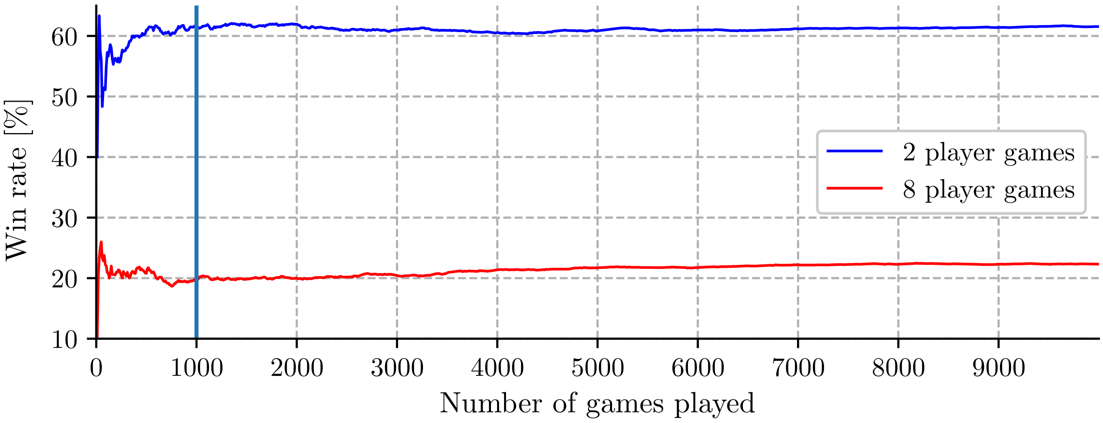


Figure 6.3: Win rate with respect to the number of games played. Games were played with one SDC and either one or seven naïve players. The vertical line shows the number of games chosen for future experiments. This number is not ideal for the eight-player games, as the win rate is still rising when reaching 10 000 games, however, it was chosen as a compromise between accuracy and the time needed to complete the experiments.

6.2 Single Turn Expectiminimax

A series of two-player games with random initialization were played against both naïve player and SDC. Table 6.3 shows data from these games. Single turn expectiminimax (STE) performs better against the naïve player than SDC does, winning 75.1 % of games and 60.7 % started from the second position.

However, in two-player games played against SDC, STE performs slightly worse, winning only 47.8 % of games. STE also wins only 24.4 % of games when starting second, whereas SDC wins 29.4 % of games in this situation.

However, as shown in Figure 6.4, STE was able to achieve higher win rates than SDC when playing games with three or more players.

For each move of the STE, the estimated probability of successfully attacking and holding a territory (Eq. 5.1) was collected alongside the actual outcome. Figure 6.5 shows

Table 6.3: Data from 1 000 two-player games between STE and both the naïve player or SDC. The table shows both the number of games won and the number of games won when starting as a second player (Won 2nd).

	Won	**Won 2nd**
STE	75.1%	60.7 %
Naïve	24.9%	10.9 %
STE	47.8%	24.4 %
SDC	52.2%	29.4 %

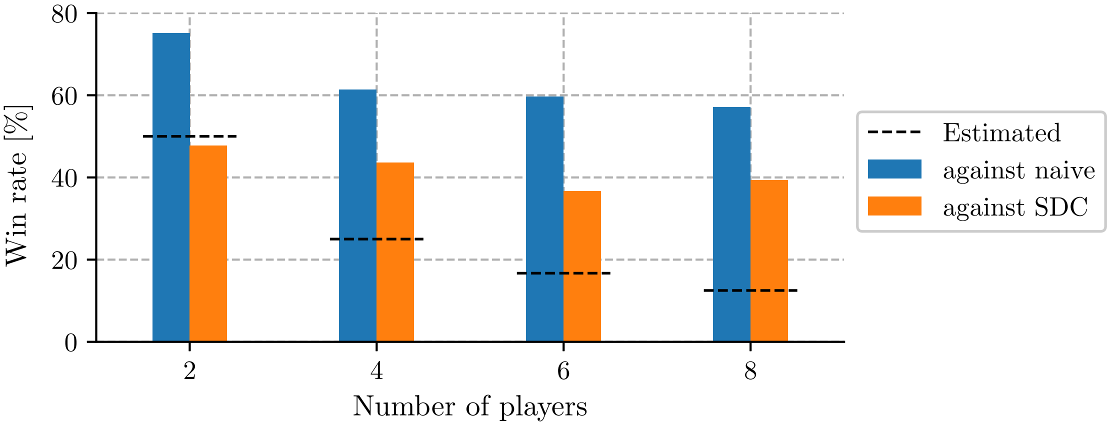


Figure 6.4: Win rate of STE from games against both naïve player and SDC. The data shown in the figure come from games where there was only one STE and the rest were either all naïve players or SDCs. The estimated win rate shows the percentage of games that should be won in random games.

the collected data. The probability estimation is quite accurate and very similar for both two-player and multi-player games.

Up to the probability of around 65 %, the actual results are better than the estimated probability. This is due to the fact that the estimation is calculated with the assumption that all enemies will attack, which is not the case because – with the exception of the naïve player – no strategy has 100 % aggressivity (aggressivity is discussed in more details in Section 6.5).

In addition, in some cases, a single enemy territory may be counted as an attacker for multiple territories when considering possible moves. On the other hand, for the probabilities higher than 80 %, the actual results are worse. This is because of the limits of used approximation as described in 5.3 – a territory might be targeted by a territory that doesn't belong to an opponent at the time of the estimation.

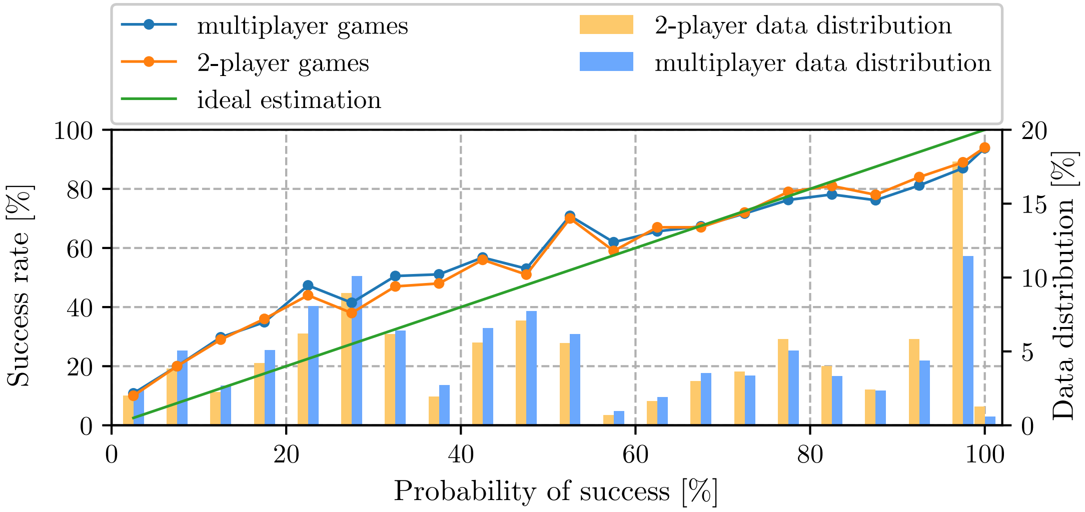


Figure 6.5: Success rate of attacking and holding a territory through opponent's turns in relation to estimated probabilities. The figure shows data collected from both two-player and multi-player games, along with the data distribution of both configurations. The ideal estimation line shows the hypothetical situation where the prediction would match the results exactly.

6.3 Improved STE

Improved STE agent (STEi) attempted to find the optimal value of success probability threshold for deciding whether to make a move. In addition, it tries to prefer attacks that would lead to player's score increase rather than taking any territory. A set of a thousand 2-, 4-, 6- and 8-player games were played with different configurations of both parameters against the naïve player, SDC, and STE.

Based on the data, I decided to choose one configuration for two-player games (threshold: 0.2, score preference: 3) and one for all mutiplayer games (threshold: 0.4, score preference: 2).

Series of thousand games were played with STEi in two-player and multi-player configurations against previous agents. Each multi-player game configuration consisted of one STEi agent, with all the rest being either naïve player, SDC or STE exclusively. Data from two-player games can be seen in Table 6.4. Even though STEi is quite efficient in games against the naïve player, it performs worse with both SDC and STE – although, in the case of STE, this might not be evident because of the higher win rate. However, STEi has won fewer games starting as a second player than STE. This is a more descriptive metric in this case because STEi played first in 52.4 % of these games.

Figure 6.6 shows STEi's win rates in these games. Despite mixed results in two-player games, STEi performs better than other agents in multi-player games.

Table 6.4: Data from 1 000 two-player games between STEi and the naïve, SDC and STE. The table shows both the number of games won and the number of games won when starting as a second player (Won 2nd).

	Won	**Won 2nd**
STEi	74.6 %	60.2 %
Naïve	25.4 %	11.2 %
STEi	48.2 %	27.0 %
SDC	51.8 %	30.2 %
STEi	50.6 %	30.9 %
STE	49.4 %	31.5 %

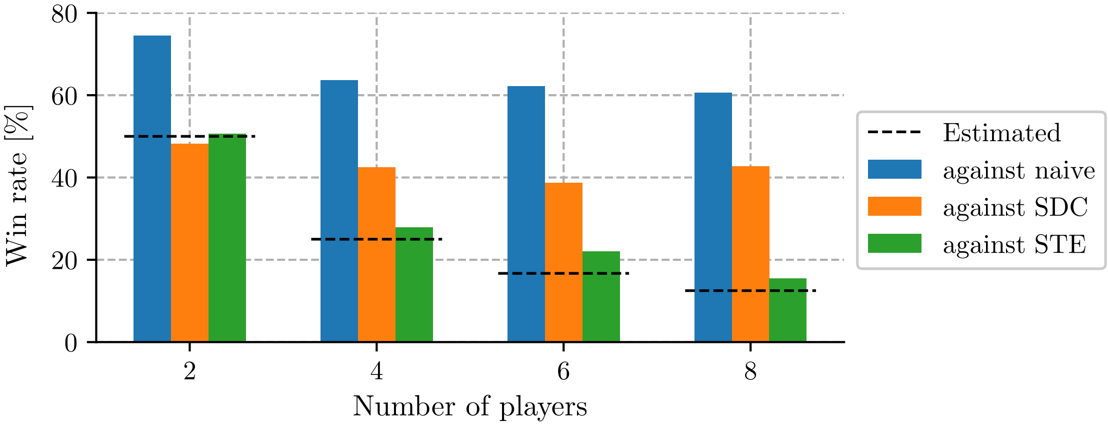


Figure 6.6: Win rate of STEi in games against the naïve player, SDC, and STE. The data shown in the figure come from games where there was only one STEi and the rest were either all naïve players, SDCs or STEs. A thousand games were played in each configuration. The estimated win rate shows the percentage of games that should be won in random games.

6.4 Win Probability Maximization

All win probability maximization (WPM) agents use weights trained on data from naïve player games. I tried training on other agents' data or a mix of multiple agents, however, the results were best when using the naïve player data.

6.4.1 Players' Scores as Features

This agent, called WPM-S, uses individual players' scores as a feature for the logistic function. Two-player data on which the parameters were trained are shown in Figure 6.7.

Data from two-player games played with WPM-S are shown in Table 6.5. WPM-S performs slightly worse than other agents against the naïve player, although it still manages to win 67 % of the games against it, with 46.1 % of won games that LR-S started as a second

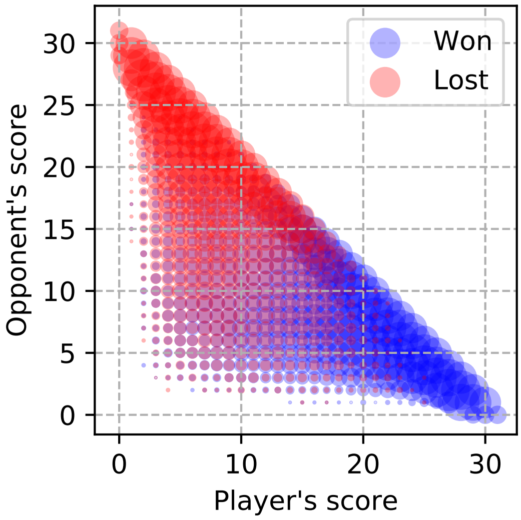


Figure 6.7: Data from the games played by naïve players used for training parameters of WPM-S agent. The size of each feature represents its frequency in the training data.

player. WPM-S has won only 38.4 % against SDC – one of the reason being WPM-Ss lower aggressivity (only 43.2 % in comparison to SDC's 69.8%). Finally, WPM-S performs slightly better against both STE and STEi.

Table 6.5: Data from 1 000 two-player game series played with WPM-S against previous agents. The table shows both the number of games won and the number of games won when starting as a second player (Won 2nd).

	Won	**Won 2nd**
WPM-S	67.1 %	46.1 %
Naïve	32.9 %	12.6 %
WPM-S	38.4 %	16.0 %
SDC	61.6 %	39.7 %
WPM-S	53.3 %	34.7 %
STE	46.7 %	25.5 %
WPM-S	52.9 %	34.2 %
STEi	47.1 %	29.6 %

Figure 6.8 shows WPM-S's win rate from a series of games played against all previous agents. In all games, there was a single WPM-S agent, with all opponents being either all naïve players, SDC, STEs or STEis. WPM-S performs better than SDC with the exception of two-player games. On the other hand, against STE and STEi, WPM-S's performance drops with increasing number of players in the game.

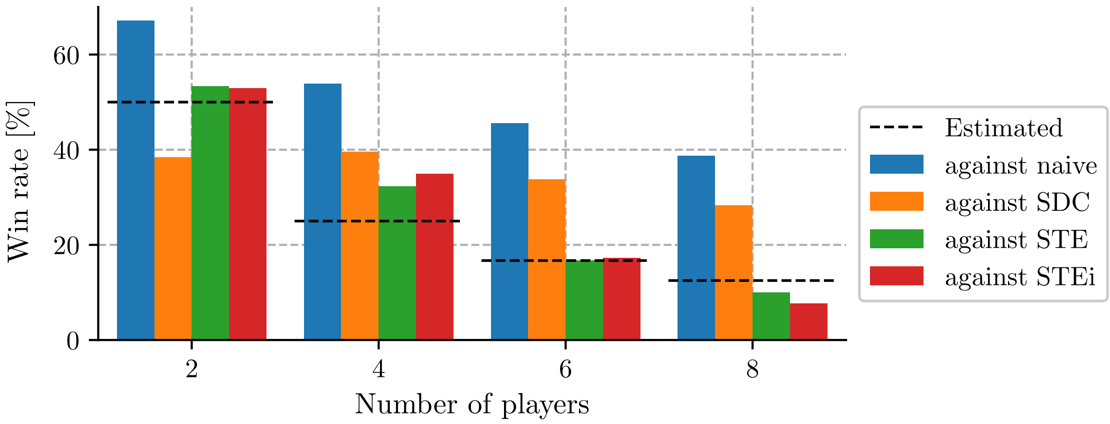


Figure 6.8: Win rate of WPM-S in against previous agents. The data shown in the figure come from games where there was only one STEi and the rest were either all naïve players, SDCs, STEs or STEis. A thousand games were played in each configuration. The estimated win rate shows the percentage of games that should be won in random games.

6.4.2 Numbers of Dice as Features

This agent, called WPM-D, uses logarithms of numbers of dice as a feature to estimate win probability. Parameters were trained on naïve players data.

Data from two-player games against all previous agents are shown in Table 6.6. WPM-D has higher win rates against all of them, being the first agent using one of the more complex strategies that is able to defeat SDC.

Table 6.6: Data from 1 000 two-player game series played with WPM-D against previous agents. The table shows both the number of games won and the number of games won when starting as a second player (Won 2nd).

	Won	**Won 2nd**
WPM-D	65.9 %	41.5 %
Naïve	34.1 %	9.6 %
WPM-D	55.6 %	26.9 %
SDC	44.4 %	18.0 %
WPM-D	55.3 %	31.1 %
STE	44.7 %	21.8 %
WPM-D	55.4 %	32.6 %
STEi	44.6 %	22.1 %
WPM-D	68.3 %	47.9 %
WPM-S	31.7 %	11.5 %

Despite its domination in two-player games, WPM-D's performance rapidly declines with increasing of players. Even though it has higher win rate than SDC in all configurations, it is substantially worse than both versions of STE agents and WPM-S.

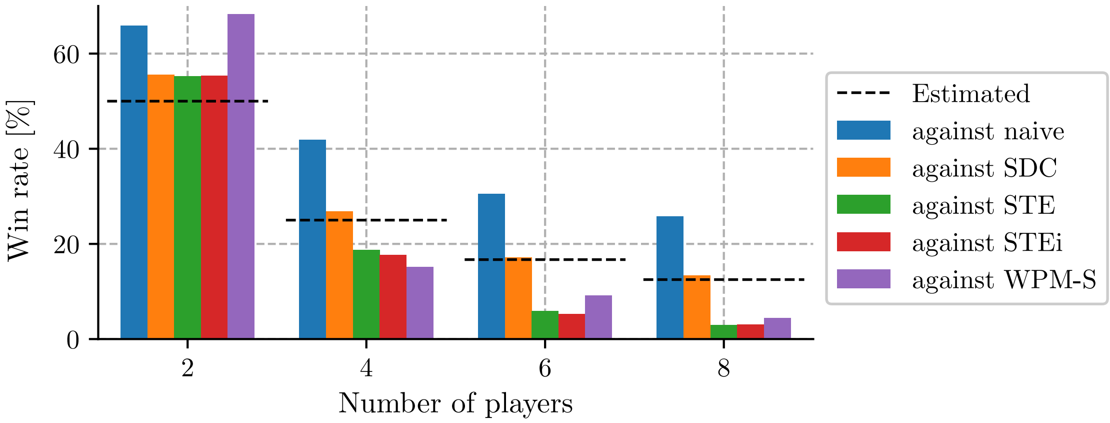


Figure 6.9: Win rate of WPM-D in against previous agents. The data shown in the figure come from games where there was only one STEi and the rest were exclusively all naïve players, SDCs, STEs, STEis or WPM-Ss. A thousand games were played in each configuration. The estimated win rate shows the percentage of games that should be won in random games.

6.4.3 Combined Features of Player Scores and Dice

This agent (WPM-C) uses combined features of logarithms of players' scores and dice to estimate win probability.

Table 6.7 shows data collected from two-player games. WPM-C's performance in these games is slightly better than when using only player's scores or dice. However, this difference is only minor.

Similarly, WPM-C's performance is close to WPM-D in multi-player games, as shown in Figure 6.10.

In conclusion, from the strategies proposed in this work, win probability maximization strategy is the best in two-player games, with best results achieved when using either players' scores or combined scores and dice as a feature to estimate win probability for each move. However, in multi-player games, WPM fails to compete with STE agents.

Table 6.7: Data from 1 000 two-player game series played with WPM-C against previous agents. The table shows both the number of games won and the number of games won when starting as a second player (Won 2nd).

	Won	**Won 2nd**
WPM-C	71.0 %	50.5 %
Naïve	29.0 %	11.6 %
WPM-C	56.4 %	27.4 %
SDC	43.6 %	16.1 %
WPM-C	53.4 %	31.3 %
STE	46.6 %	22.8 %
WPM-C	54.8 %	32.3 %
STEi	45.2 %	22.9 %
WPM-C	69.0 %	48.7 %
WPM-S	31.0 %	10.6 %
WPM-C	50.1 %	19.1 %
WPM-D	49.9 %	19.3 %

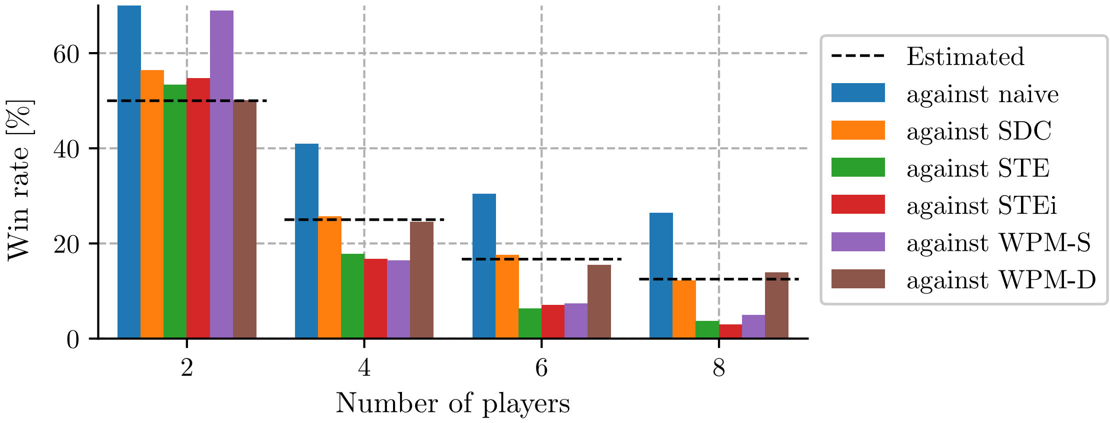


Figure 6.10: Win rate of WPM-C in against previous agents. The data shown in the figure come from games where there was only one STEi and the rest were exclusively all naïve players, SDCs, STEs, STEis, WPM-Ss or WPM-Ds. A thousand games were played in each configuration. The estimated win rate shows the percentage of games that should be won in random games.

6.5 Aggressivity

Throughout the experiments described in this chapter, aggressivity of all agents was measured as the ratio of carried out moves to the number of all possible moves in a turn. Naïve player, performing all moves, thus has 100 % aggressivity.

Aggressivity of all agents used in this work is shown in Table 6.8. In general, agents with higher aggressivity can – to a certain degree – compensate for lack of good decision-making in two-player games.

Table 6.8: Aggressivity of each agent used in this work. Data were collected from all games played in experiments described in this chapter.

	Naïve	**SDC**	**STE**	**STEi**	**WPM-S**	**WPM-D**	**WPM-C**
Aggressivity [%]	100	78.3	45.7	39.8	48.4	72.0	70.0

Because of this, a series of thousand games against each agent were played using a modified naïve player. This modified agent behaved in the same way as the one described in Section 5.1 with the addition of artificially lowered aggressivity to ∼78 %[1]. To achieve the lowered aggressivity, for each turn, the agent rembers all territories from which a move can be made as well number of moves already taken. If the ratio of the taken moves and number of territories that can make a move would exceed a certain value[2], the agent would end its turn.

This modified naïve player performed worse against the original naïve player, winning only 44.9 % of games. However, it has a slightly better win rate than the original when playing against both STE versions. Win rates against all agents are shown in 6.9.

Table 6.9: Win rate of the modified naïve player with lower aggressivity against all agents. A set of a thousand games were played against each agent.

	Naïve	**SDC**	**STE**	**STEi**	**WPM-S**	**WPM-D**	**WPM-C**
Win rate	44.9	24.9	27.3	29.6	26.5	21.9	21.4

---

[1] The reason for using this value is that SDC has the same aggressivity and can perform quite well in two-player games while not using a particularly complex way to choose its moves.

[2] This ratio was chosen experimentally and has a value of 0.65 in the case of 78 % aggressivity.

Chapter 7

Conclusion

In this bachelor project, I have proposed several strategies for the game Dice Wars using both game state search approach and logistic regression.

To evaluate individual strategies, I implemented Dice Wars as a client-server application in Python under the GPL license[1]. In addition, more AI agents can be created on top of this implementation without the need to handle the game's logic.

Experiments carried out in this work have shown that in two-player games, it is often useful to be aggressive instead of making only the most favourable moves. For this reason, the agent using win probability maximization with combined scores and dice (WPM-C) – with the aggressivity of ~70 % – proved to be the best of the proposed agents in two-player games. WPM-C has win rate higher than 50 % against each of the other agents and has an average win rate of 59.1 % in all two-player games it played.

In multi-player games, on the other hand, it is generally better to make not make moves with low probability of success. For this reason, the improved single turn expectiminimax strategy (STEi) performs the best. In 8-player games, it has over 60 % win rate against naïve players. In 8-player games played with all agents, STEi has the highest win rate of 21.4 %, with STE being close behind with 20.4 % win rate.

Results of this work's experiments were presented at the Excel@FIT student conference [14] where it also received an award from the expert committee.

Future work will focus on combining both approaches used in this work. In addition, as none of the developed agents utilize players' dice reserves, adding this into account will potentially lead to higher performance. Furthermore, the influence of the map topology on individual agents' performance can be analyzed.

---

[1] Source codes can be accessed from `https://github.com/dturecek/dicewars`

Bibliography

[1] Bishop, C. M.: *Pattern Recognition and Machine Learning*. Springer Science Business Media. 2006. ISBN 0-387-31073-8.

[2] Feng-Hsiung Hsu; Campbell, M. S.; Hoane, A. J.: Deep Blue System Overview. In *Proceedings of the 9th international conference on Supercomputing*. 1995. ISBN 0-89791-728-6. pp. 240–244. doi:10.1145/224538.224567.

[3] Goodman, D.; Keene, R.: *Man Versus Machine: Kasparov Versus Deep Blue*. H3 Inc. 1997. ISBN 1-888281-06-5.

[4] Hendel, S.; Hoffman, C.; Manack, C.; et al.: Taking the risk out of RISK: Conquer Odds in the Board Game RISK. 2015. [online; seen 2018-05-04]. Retrieved from: https://arxiv.org/abs/1512.04333

[5] James, G.; Witten, D.; Hastie, T.; et al.: *An Introduction to Statistical Learning with Applications in R*. Springer. 2013. ISBN 978-1-4614-7137-0.

[6] Russel, S. J.; Norvig, P.: *Artificial Intelligence: A Modern Approach, 3rd ed.* Prentice Hall. 2010. ISBN 978-0-13-207148-2.

[7] Samuel, A. L.: Some Studies in Machine Learning Using the Game of Checkers. In *IBM Journal of Research and Development*. 7 1959. pp. 210–229. doi:10.1147/rd.33.0210.

[8] Schaeffer, J.; Burch, N.; Björnsson, Y.; et al.: Checkers Is Solved. In *Science*. 2007. pp. 1518–1522. doi:10.1126/science.1144079.

[9] Silver, D.; Huang, A.; Maddison, C. J.; et al.: Mastering the Game of Go with Deep Neural Networks and Tree Search. In *Nature*. 2016. pp. 484–489. doi:10.1038/nature16961.

[10] Stratilová, L.: *The Hnefatafl Board Game Artificial Intelligence*. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. 2012. Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=13210

[11] Tan, B.: Markov Chains and the RISK Board Game. In *Mathematics Magazine*, vol. 70. 12 1997. pp. 349–357. doi:10.2307/2691171.

[12] Tesauro, G.: Temporal Difference Learning and TD-Gammon. In *Communications of the ACM*. 1995. pp. 56–68. doi:10.1145/203330.203343.

[13] Tesauro, G.: Programming Backgammon Using Self-teaching Neural Nets. In *Artificial Intelligence*. 2002. pp. 181–199. doi:10.1016/S0004-3702(01)00110-2.

[14] Tureček, D.: Artificial Intelligence for a Board Game. In *Excel@FIT, Student Conference*. 2018.

[15] Yannakakis, G. N.; Togelius, J.: *Artificial Intelligence and Games*. Springer. 2018. ISBN 978-3319635187.

Appendix A

Contents of the CD

`/`

`thesis.pdf`	Text of the thesis.
`tex`	LaTeX sources.
`dicewars`	Implementation of Dice Wars.
`client`	Client implementation.
`server`	Server implementation.
`dicewars.py`	Script to run the game.
`README`	Contents description.