Testing Opponent Modeling Strategies

Melanie Mayrhofer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2015

© Copyright 2015 Melanie Mayrhofer

This work is published under the conditions of the *Creative Commons License Attribution–NonCommercial–NoDerivatives* (CC BY-NC-ND)—see http://creativecommons.org/licenses/by-nc-nd/3.0/.

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 25, 2015

Melanie Mayrhofer

Contents

Declaration iii

Kurzfassung vi

Abstract vii

1 Introduction 1
- 1.1 What is Opponent Modeling? 1
- 1.2 Goals 4
- 1.3 Outline 4

2 Related Work 6
- 2.1 Board games 6
- 2.2 Card games 7
- 2.3 Video games 7

3 The Game 10
- 3.1 Rules of the Game 10
  - 3.1.1 Units 11
  - 3.1.2 Types of Platforms 12
- 3.2 Implementation 13
  - 3.2.1 Architecture 14
  - 3.2.2 The Static AIs 16
- 3.3 Game Balance 20

4 The Opponent Modeling AI 22
- 4.1 Naive Bayes' Classifier 24
- 4.2 CART Classifier 26
- 4.3 kNN Classifier 29
- 4.4 Implementation Details 31

5 Tests and Evaluation 35
- 5.1 Naive Bayes' Classifier 36
- 5.2 CART Classifier 39

5.3 kNN Classifier . . . 43
5.4 Combined Results . . . 45

6 Conclusion and Outlook **47**
6.1 Lessons Learned . . . 47
6.2 Outlook . . . 49

A Content CD-ROM **51**
A.1 PDF Files . . . 51
A.2 Executable JAR Files and Dependencies . . . 52
A.3 Test Results . . . 52

References **53**
Literature . . . 53
Online sources . . . 55

Kurzfassung

Eine künstliche Intelligenz (KI), die es in einem Spiel schafft die gegnerische Strategie herauszufinden und diese Information nutzt, bietet eine größere Herausforderung und ein spannenderes Spielerlebnis. Dieser Ablauf ist unter dem Begriff *Opponent Modeling* bekannt. Um die Strategie des Gegners zu identifizieren, können Klassifizierungsansätze verwendet werden. *Opponent Modeling* zählt daher zu den Klassifizierungsproblemen. In dieser Arbeit werden drei verschiedene Ansätze getestet und die Ergebnisse verglichen.

Ein Ziel dieser Arbeit ist es, die wichtigsten Schritte, die bei der Erstellung einer solchen KI nötig sind, zu veranschaulichen. Zusätzlich soll die gezeigte Implenentierung anderen Spieleentwicklern helfen, diese Methoden in ähnlichen Spielen zu verwenden. Getestet wurden folgende Klassifizierungsmethoden: Naive Bayes', der CART Algorithmus zur Erstellung eines Entscheidungsbaumes und die Nächste-Nachbarn-Klassifikation. Ein rundenbasiertes Strategiespiel wurde als Testumgebung für die Erkennung der gegnerischen Strategien im Zuge der Master Lehrveranstaltungen Thesis-Project I und II in der Programmiersprache Java implementiert. Das Spiel ist ein Nullsummenspiel für zwei Spieler mit perfekter Information. Die Aufgabe der Spieler ist es spezielle Spielfelder mit ihren Spielfiguren einzunehmen, um sich einen Vorteil gegenüber ihren Gegner zu verschaffen. In dem implementierten Spiel gibt es drei verschiedene Strategien. Testspiele, in denen statische KIs gegeneinander angetreten sind, wurden verwendet, um die verschiedenen Klassifizierungsmethoden zu trainieren.

Um gute Ergebnisse zu erzielen, müssen mehrere Faktoren berücksichtigt werden. Das Bestimmen der richtigen Features ist einer der wichtigsten Schritte bei der Klassifizierung. In dem implementierten Spiel wurde der Gegner nach jeder Runde klassifiziert, um möglichst schnell auf die erkannte Strategie des Gegners reagieren zu können. Die KI gewann mit Hilfe der verschiedenen Klassifizierungsansätze 44 von 45 Spielen. Ein Spiel verlor die KI aufgrund von Fehlklassifizierung in den ersten Runden. Während der Evaluierung konnte dieser Fehler behoben werden, indem die Anfangsstrategie der KI für die ersten fünf Runden fix gesetzt wurde. Die Ergebnisse zeigen, dass mit dieser Maßnahme die Genauigkeit der Methoden verbessert werden konnte. Die besten Ergebnisse lieferte CART mit einer Genauigkeit von fast 95 %.

Abstract

An artificial intelligence (AI) in a game which is able to create a model of the opponent player and uses this information to compete against this opponent makes the game AI more challenging and more exiting to play against. Opponent modeling is considered a classification problem, as the methods try to classify the strategy of the opponent. This research is about testing different opponent modeling strategies in a turn-based strategy game in order to evaluate their performance.

This thesis should provide the reader with information about the most important tasks when implementing a simple opponent modeling AI in a game. Additionally the results should give other game developers guidance about how to include the opponent modeling methods in a similar game setup. The tested classifiers are: Naive Bayes', the decision tree classifier CART and k-nearest-neighbor. The testbed for the opponent modeling is a simple turn-based strategy game developed in Java during the master programs ThesisProject I and II. It is a two-player, zero-sum game where the player moves units to special platforms in order to capture them and receive rewards. The players in the game do have perfect information about the whole setup and their opponent's moves. There are three different strategies in the game. Test-games, where static AIs played games against each other, are used to train the opponent modeling AI.

In order to achieve good results, there are many tasks to be considered. Feature extraction is one of the most important ones. As classification was performed every round, the opponent modeling AI could quickly react upon the opponent's strategy. With the help of the classifiers, the opponent modeling AI did win almost every game against the static AIs. In one out of 45 games, misclassification in the early game caused the opponent modeling AI to lose the game. During the evaluation process, this error could be eliminated by fixing the strategy of the opponent modeling AI during the first few rounds. By doing so, the accuracy of all classifiers was increased. All in all, the CART classifier performed most accurately. In almost 95% of all rounds, classification was correct. The other classifiers did just slightly worse.

Chapter 1

Introduction

Entertainment is the most important goal of almost any game. In order to reach this goal, the game designers and developers try to produce games which capture the players' complete attention and make them forget about everything else while playing. This condition is better known as the game *flow*. In *The Art of Game Design: A Book of Lenses* [21] flow is described as "a feeling of complete and energized focus in an activity, with a high level of enjoyment and fulfillment." Besides clear goals, no distractions and direct feedback, providing the players with continuous challenges is a crucial part of keeping them in the state of flow as long as possible. Of course, if the challenges in the game are too difficult, the players are frustrated quickly. On the contrary, players become bored if the game is too easy. Figure 1.1 visualizes the flow channel between boredom and frustration.

When including an artificial intelligence (AI) in the game, it is difficult to achieve constant flow for the player. When the AI is over powered, or obviously cheating the player will not enjoy the game. On the other hand, if the AI is static, or non-adaptive, meaning it is not able to analyze the player's actions and react upon them, the player can easily exploit the AI's weaknesses. Beating the same AI is easy then and again, the game will lose its charm.

An approach of creating a non-cheating but challenging AI is to include the ability of opponent modeling.

1.1 What is Opponent Modeling?

In a game, where all different kinds of strategies are known, the enemy's strategy is exploited by classifying the current game state or the actions of the opposing player. This classification and the process of identifying the strategy of opponents with the help of the observed information of their behavior is called *Opponent Modeling*. In Bakkes et al. [2] it is defined as creating "an abstracted description of a player or of a player's behavior

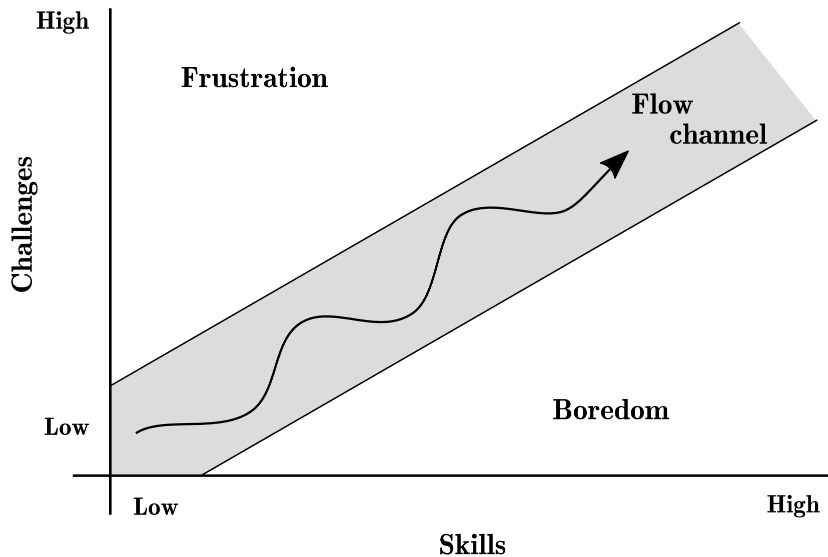


Figure 1.1: Game flow: When the challenges are too high, but the skill level of the player is low, frustration is inevitable. If the player's skills are high and the challenges are not, the player becomes bored. Inside the flow channel, skills and challenges are well-matched and the player experiences excitement. The waved arrow should visualize the player's experience growth, which makes the game a bit easier, followed by rewards and new challenges. The original picture and a more detailed description can be found in [21].

in a game." Hence, the task of an opponent modeling AI is to identify its opponents' strengths and weaknesses and react to them. Of course, in order to keep the game fair and the players inside the flow channel, the opponent modeling AI should only use information which could be observed by a human player too (no cheating).

Opponent modeling is used to create game AIs that are fair, more challenging and more exiting to play against. An opponent modeling AI which is able to use the observations to identify the opponent's strategy can achieve better results than playing a known optimal solution a.k.a. Nash equilibrium solution concept [31]. For example, in the game "Roshambo" (rock-paper-scissors) the Nash equilibrium strategy is to play randomly. When the AI is doing so, it will win 1/3 of all games, 1/3 of the games will be lost and the rest are ties. When the opponent modeling AI finds out, that its enemy sticks to playing rock, it can switch its strategy and play paper as a counter. As a result, it will win more than 1/3 of the games, or force the opponent to switch to another strategy too. Hence, the AI is able to perform better than playing the Nash equilibrium solution when it is able to identify the

opponent's strategy.

Here, the term *strategy* refers to the plan of the player how the long-term goal can be reached in the game. For example, there are different strategies in the game series of Sid Meier's Civilization. These strategy games are turn-based and the main goals there are: building a civilization, capturing land, harvesting resources and money in order to invest in new technology or a better army to gain an advantage over the opponent civilizations. On the one hand, the game can be won aggressively, by destroying the other civilizations. On the other hand, when putting effort into research and the technology improvements in order to build a rocket and explore the space, the game can be won defensively too. The overall plan, which actions to make in order to win the game, is considered a game strategy. On the contrary, when talking about tactics, the duration is short as a tactical move includes only one or a small set of actions. For instance, a tactical move in chess would be if a player decides to sacrifice the rook in order to save the queen from being taken. All in all, the strategy encloses all tactical moves during the play.

A game AI which is able to create a model of the opponent player and uses this information to compete against this opponent makes the AI more challenging and more exiting to play against. Opponent modeling is considered a classification problem, as the methods try to classify the strategy of the opponent. The different classification methods are part of the field of *Machine Learning*. Bowling et al. [4] state that opponent modeling, as well as partner and team modeling, can be found in the sub-domain "learning about players". Here, the pattern classification methods are used in order to identify the strategy of the opponent in the game.

At first, when creating an opponent modeling AI, it is important that the AI is aware of all the possible strategies in the game. In order to become familiar with the game, the AI needs to be trained. This is similar for all human players too. However, for a human being, classification is done automatically. Certainly, knowing the rules of the game and some training is also necessary to become familiar with the possibilities in the game. Furthermore, being aware of the strategies and their counter is important to have a chance against a more experienced player. Nonetheless, for a computer program the classification task is not trivial and requires more steps than training. Some of those steps are discussed in detail in the subsequent chapters.

After the opponent modeling AI has been trained, it is able to use the observations during the game in order to classify its opponent with the help of pattern classification algorithms.

1.2 Goals

In this research one of the main goals is to test different opponent modeling methods in a small game environment in order to evaluate their performance. The three methods which were tested are: *Naive Bayes'*, the decision tree classifier *CART* and the nearest-neighbor approach *kNN*. Besides the testings the results should give other game developers guidance about how to include these methods in a similar game setup. The test environment of the opponent modeling methods is a small turn-based strategy game, which is written in the object-oriented computer programming language Java and was developed during the master programs: ThesisProject I and II. The classifiers were implemented in Java as well and no software tools were used for the testings.

In previous work, most of the opponent modeling methods were tested in well-known games, like Civilization or StarCraft® (developed and published by Blizzard Entertainment™), with replay data from expert plays. As a result, the classifiers could be trained effectively. As the implemented game here is not available online or somewhere else, and as it was just created for testing the different opponent modeling methods, there is no data from game logs available. Hence, another goal of this research is also to encourage game developers to include opponent modeling methods before having their games released.

As mentioned before, the pattern classification methods were implemented by hand. This approach is advisable if the classification methods are used for the first time in order to fully understand the different methods and the mechanisms beyond. One big advantage is that the reasons for misclassification can be identified easier if one is familiar with the workflow of the used classification method.

The following list provides an overview of the main goals of this thesis:

- Test and evaluate different pattern classification methods for opponent modeling.
- Provide an idea of how to implement the tested methods for other researchers in this field.
- Show how the methods can be used without replay data or game logs from expert play.

1.3 Outline

In chapter 2, related work in the field of opponent modeling in games is presented. Studies for the different types of games, like board games, or video games, are structured and summarized in the subsections of this chapter.

The turn-based strategy game and the rules of the game are described in detail in chapter 3. Additionally, some information about the implemented

game architecture and the used algorithms can be found in section 3.2. The identified strategies in the game are described in section 3.2.2. At the end of this chapter, some information about the game balance is provided.

In chapter 4, the opponent modeling AI is presented. The main parts here are: the AI's workflow, feature extraction and the implemented classifiers. Section 4.1 covers the basic information about the Naive Bayes' classifier. A classifier using a decision tree for the identification of the opponent's strategy is presented in section 4.2. The basics of the kNN classifier are described in section 4.3. Implementation details of the AI as well as the classifiers are given in the last section of this chapter.

Chapter 5 covers the test results of the opponent modeling AI when playing against static AIs. The performance as well as the simplicity of each classifier was evaluated individually. The results are compared in section 5.4.

The last chapter summarizes the results of this research and provides an outlook for further improvement of the proposed opponent modeling strategies.

In appendix A the content of the enclosed CD-ROM is listed.

Chapter 2

Related Work

By analyzing the tactical moves of a player and trying to classify them, opponent modeling can be considered as a classification problem. In the book *Pattern Classification* by Duda et al. [9] detailed information about classification methods and algorithms can be found. Besides these descriptions, mathematical foundations are enclosed at the end of this book in order to better understand the described methods for pattern classification.

Bakkes et al. [2] describe in the article "Opponent modelling for case-based adaptive game AI" the two different roles of player modeling. There are: (1) modeling the companion and (2) modeling the opponent player. When trying to identify the strategy of a team member, the opponent modeling AI is used to help and adjust its moves to the player. On the other hand, when opponent modeling is used to compete with the player, the task of the opponent modeling AI is to identify the player's strategy and try to challenge it with the appropriate counter.

The following sections provide an overview about related work in the field of opponent modeling considering different types of games.

2.1 Board games

In zero-sum board games with perfect information, like chess, opponent modeling can be done with adversarial search. Certainly, the Minimax algorithm is infeasible in most games, as the search tree of the possible moves is too large. Adding alpha-beta pruning helps to reduce the search space a little. However, Minimax tries to compute the Nash equilibrium for both players, meaning the algorithm assumes that the opponent does not make mistakes and tries to maximize the payoff [12].

In board games with imperfect information Stankiewicz and Schadd [24] used a Bayesian approach to model the opponent's behavior in Stratego. With the help of probability distributions they tried to determine the unknown pieces of the game. A similar approach was provided by Zhang [28]

who built an opponent model for a variation of Stratego. For easier classification they created models of the arrangement tendency of opponents out of history data of the game. A probability table for each different strategy was built to generate decision methods. As a result, four different opponent models (general, aggressive, balance and defensive model), on the base of the players' arrangement tendencies, could be established.

2.2 Card games

When considering opponent modeling in card games, Poker is the one which has been studied most. While Hoehn et al. [14] analyzed Khun Poker, a simplified two-player version of Poker, Boudewijn and other researchers examined the most popular variation of Poker, Texas Hold'em [3, 22]. In order to build reliable opponent models, historical data from games against human players was used and stored in an artificial neural network. In the end, the system was able to predict the next action (fold, call, raise) of the opponent with an accuracy up to 60%. The most popular Poker bot which includes an opponent modeling system is called Poki [5, 6]. Although there has been some progress in creating an opponent model for Poker, there are still some factors which were not taken into account. Davidson et al. [6] state that there can be improvement of the opponent model when it comes to a showdown by making inference about the players' actions after the cards are known.

Schauenberg [20] mentions in "Opponent Modelling and Search in Poker" that Poker is an excellent testbed for opponent modeling as it is possible to include knowledge about the opponent players from previous games. That is because when playing Poker, all players usually play more than one session against the same opponents. Hence, information about the behavior of the players from the previous games can be easily included in the classification process for the following games.

2.3 Video games

In Weber et al. [27], opponent strategies for the Real-time strategy (RTS) game StarCraft[1] were identified with a data mining approach. The main goal of this research was to create a general model of expert StarCraft gameplay. The domain knowledge was extracted from a large amount of replay data which is available for StarCraft. When analyzing expert play, rules for certain strategies can be extracted and used for classifying the opponents' strategy. Algorithms from an open-source data mining software called "Weka" [33] were used for the classification.

Synnaeve et al. [25] introduced a Bayesian model in order to infer the opponents' opening strategy in RTS games. The testings were also done in

[1] developed by Blizzard Entertainment™

the RTS game StarCraft. The idea here is to predict the opponents strategy during the game phase where the players do not see the actions of each other as they are covered beyond the fog of war[2]. The opening timespan includes the first 5 to 15 minutes of the game. For the training of the Bayesian model Synnaeve et al. used the labeled replay data from Weber et al. [27]. For clustering the data, k-means was implemented. In the end, the implemented bots by Synnaeve et al. were able to predict the opening strategy of the opponent with an accuracy of about 65% after the first five minutes. After ten minutes, this prediction even reached an accuracy up to 94% in some cases. Furthermore, Synnaeve et al. state, that their approach is robust against noise if there are some observations missing.

Avontuur [1] compared the performance of different classification methods in the RTS game Wargus. The testing was done with the help of the previously mentioned Weka toolkit. Among the tested classification methods was the Naive Bayes' classifier, the Ibk algorithm, which uses the nearest-neighbor approach and also an implementation of the C4.5 algorithm which uses a decision tree. All possible actions in the game were used for the classification. When doing so, also unnecessary or overfitting information is included. Hence, it is clear that the results of the testings were not particularly outstanding.

Another approach for predicting opponent behavior can be found in Rashad [18] where the classification is done with the help of rough sets and neural networks. After identifying the main features of the game, rough sets are used to extract the relevant data to reduce the complexity to a minimum. This is done in order to overcome the disadvantages of neural networks, like the slow learning rate, which occurs if the dimension of the input data is large.

In the book *AI Game Programming Wisdom 3* in the chapter "Preference-based player modeling" a description of how opponent models are built with the help of preference functions can be found [8]. Preference-based player models were created for the turn-based strategy game Civilization IV (CIV4) [19, 23]. As in CIV4 games can be won in peace or war, the main goal of these researches was to determine if the players prefer to fight or not. The training data upon which the opponent models were built was gathered from 1 vs. 1 battles where AI bots played against each other. Unfortunately, the results of the player model were not accurate enough as CIV4 is a large game with many different features. Furthermore, preferences of human players could be predicted only if the player stuck to a certain play style.

Van Der Heijden et al. [26] used the information about the opponent to change the formation of the units dynamically in the Open Real-Time

[2]The fog of war refers to an area in the game which is unknown until the player moves units to it.

Strategy (ORTS) game environment. In this research they built different opponent models and counter strategies for each model. By classifying the opponents' strategy the most suitable model was determined considering the given game observations. With this technique also new and unknown opponents could be classified. The Bayes' theorem was used for the classification. Hence, the model with the highest likelihood given the observed feature values was determined.

Besides the opponent modeling in RTS games, it can also be found in first person shooters or multi-agent systems. Hladky and Bulitko [13] introduce in their research a possibility to predict the opponents position in the first person shooter Counter-Strike: Source (developed by Valve Software). For this prediction, they used a modified hidden Markov model approach and particle filters. Ledezma et al. demonstrate in their research that opponent modeling can be even added to a soccer game of robots (RoboSoccer) [15].

To sum up, opponent modeling is possible, however not always easy, for different types of games. An AI which is able to identify the strategy of its opponent and uses this information to adapt its play style can make a game more attractive, provided the modeling is done correctly. In some of the previous work toolkits with the implemented data mining algorithms were used for the testings. Furthermore, opponent modeling was added to games with online logs or available replay data where models were built according to large amount of training data. Hence, there has already been some research on this topic, however, there is still much room for improvement.

Chapter 3

The Game

The environment for testing different opponent modeling strategies is a small turn-based strategy game. This test environment was developed during the master program subjects: ThesisProject I and II. Basically, it is a two-player zero-sum game with a fully observable environment. Each player controls units, which can be moved between the platforms. The goal of the game is to send these units to special platforms in the game world to gain a competitive advantage over the opponent. Figure 3.1 displays a mock-up of the game world. The main idea of capturing platforms originates from the game Biotix (BIOTIX:PHAGE GENESIS developed by TEN PERCENT RED) which is a fast-paced real-time strategy (RTS) game for mobile devices. Further influences to the gameplay come from turn-based games like Sid Meier's Civilization V(published by Take-Two Interactive Software and its subsidiaries) and the board game Risk(© 2000 Hasbro International Inc.).

The following sections cover the overall game setup, like the rules of the game and some implementation details. In order to be able to test different opponent modeling strategies, static AIs were implemented and take the role of stable opponents. These AIs as well as their implemented strategies will be explained in section 3.2.2. At the end of this chapter, some information about the game balance is provided.

3.1 Rules of the Game

One crucial factor of being able to play a game well is to know and understand the rules of the game. For the small test environment there are just a few of them. As mentioned before, it is a two-player game. In order to differentiate between the players, different colors are assigned to them. The game world consists of platforms. Paths between these platforms allow the players to move units along the paths to the platforms. Each path is divided into steps. At the beginning of the game, each player owns a base platform and two units. These units are placed on the base. Additionally, the player

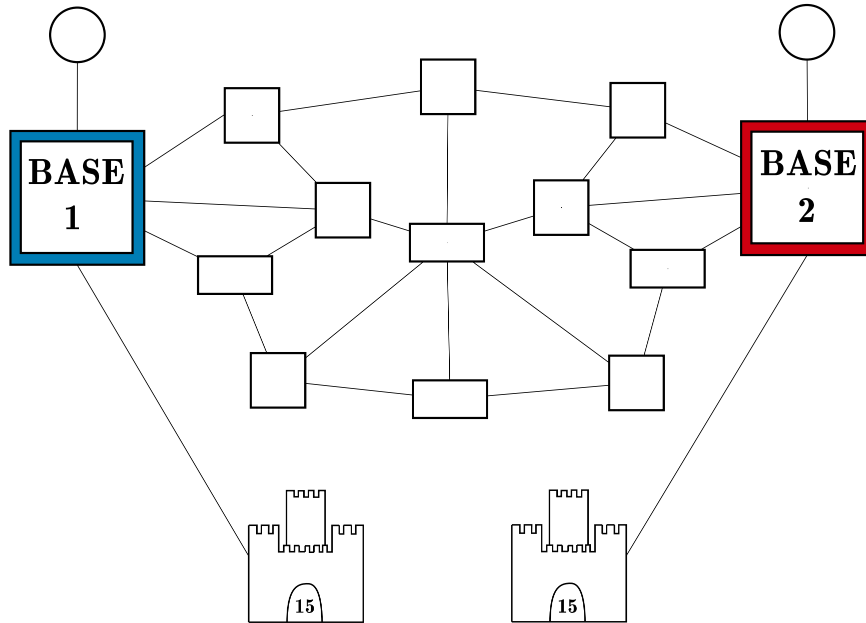


Figure 3.1: Game mock-up: The game is turn-based, two player and zero-sum. Different platform types allow the players to make strategic decisions.

may move units five steps per turn. The core rules of the game are as follows:

1. Turn-based: players do not play simultaneously but take turns.
2. Each turn a player is allowed to perform a certain amount of moves.
3. Units belonging to the player can be moved from one platform to another along the given paths.
4. Platforms are obtained when moving units to them.
5. Owning a platform: the player receives benefits for obtaining a platform at the player's next turn.
6. Win the Game by destroying the enemy base, or
7. win the Game by taking over the special platform called Castle (detailed description see section 3.1.2).

3.1.1 Units

In the game, the player owns units. These can be moved around. For the sake of convenience, the units in this game are visualized as circles colored in their owners' color. The strength of a unit is equal to its hit points (HP). Meaning, when there is a fight between two units with equal HP they cancel each other out. If a unit attacks an enemy unit with less HP, the enemy

unit gets destroyed, however the attacking unit also loses HP. Units can be upgraded when owning the platform of type C (see section 3.1.2)

In addition to the players' units, there are also neutral units in the game. These units are static during the whole game—they do not move because they do not belong to a player. When there are neutral units on a platform, they have to be destroyed first, before a player can take over the platform.

3.1.2 Types of Platforms

Beside the player's base there are four other types of platforms in the game. Figure 3.2 displays how these different types are visualized. These platforms do not just grant their owner advantages during the play, but also give the players the possibility to make different strategic decisions. In the following, these four types are described in detail.

Platform A

When owning a platform of type A, a new unit spawns directly on this platform in the next round. New units are created as long as the platform belongs to a player (number of units on the platform is greater than zero). Owning a platform of type C will prevent new units from spawning. If there are neutral units on a platform of type A, no units are spawned on it.

Platform B

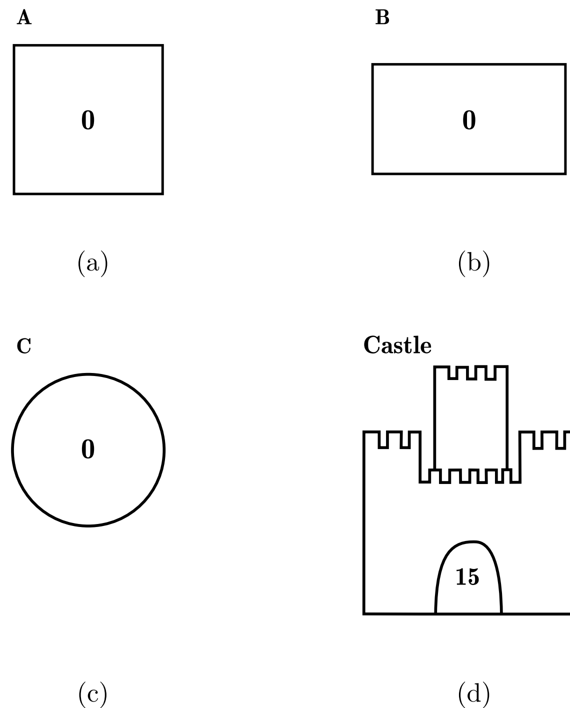
Positioning units on a platform of type B grants the owner of the units extra moves in the next round. For example, if a player sends five units to a platform of type B, the player is allowed to do five extra moves at the next turn (provided the units on this platform stay alive).

Platform C

When a player owns a platform of type C, all platforms of type A belonging to the same player will stop spawning new units. After ten rounds, all units of the player receive extra HP. In addition to that, the player's moves per turn are permanently increased by three.

Castle

The special platform called Castle is occupied by 15 neutral units. A player has to defeat these units first, before the platform can be taken over. However, when a player has taken over the Castle, the game is over and that player wins.



(a) (b) (c) (d)

Figure 3.2: This figure displays the different types of platforms in the game. Most of the platforms are visualized as basic shapes. The square in (a) represents platforms of type A, the rectangle in (b) platforms of type B, the circle in (c) platforms type C and the shape in (d) visualizes the special platforms Castle. The number inside the platform indicates the current number of units on it. When a player owns a platform, the outline is colored with the player's color.

3.2 Implementation

The game was implemented in the computer programming language Java during the master program subjects: ThesisProject I and II. The open source game development framework libGDX[1] was used for the visualization. In order to build a proper architecture with lose coupling, a messaging system in connection with an entity-component system was implemented. These systems are explained in detail in the section 3.2.1.

The game loop[2] saves a list of tasks which are executed once per frame. Everything in the game, which needs to be updated during the game loop, implements the interface *Task*. The following code displays how this interface is written in Java and the three methods, *initialize*, *execute* and *cleanup*,

[1] http://libgdx.badlogicgames.com

[2] The game loop is sometimes called the heartbeat of a game, as it is responsible for keeping the game alive.

which are known by the game loop:

```
1 public interface Task {
2   public void initialize();
3
4   // return true if this task has finished, false otherwise
5   public boolean execute();
6
7   public void cleanup();
8 }
```

The initialize method is called when a new task is added to the game loop. Inside this method general jobs for initialization are done. For example, memory is allocated for lists and hash maps, the class subscribes to explicit messages from the messenger, etc. As mentioned before, the game loop calls the execute method of the stored tasks every frame. The return value of this method indicates if this task has finished its work or if it needs to be executed again. When a task is not needed any more, e.g., an entity in the game got destroyed, the task is removed from the game loop's list of tasks and its cleanup method is called. The task then frees any resources used. With this generic interface, any task can be executed by the game loop[3].

3.2.1 Architecture

In a game, everything with which the programmer can interact is called an entity. In the book *Introduction to Game Development* by Rabin [17] an entity is defined as "a self-contained piece of logical interactive content". For this game the entities are units, platforms and players. An entity consists of several components and dependencies. Figure 3.3 shows how the entity for units is composed. Small components are easily exchangeable and can be reused for different entities. For example, there is a component in the game which stores the information about the owner of an entity. Another component is responsible for the movement along a given path, a third one saves HP, and the last one saves the position in a coordinate system. All these components together can be used to create a unit entity. Additionally, the component with the information about the owner, as well as the knowledge about the position can be used for the platform entity too.

A class called *EntityManager* takes care of all the entities. It provides methods to add and remove entities from the game. Additionally, there are other managers inside the game logic. The *GameManager* adds the AIs to the game and handles the turns. The *UnitManager* creates and destroys the unit entities and passes the information to the *EntityManager*, etc.

Inside the update method of the components continuous computations, like the movement of a unit, take place. The results are wrapped in messages and sent to the messaging system (messenger). Each message is labeled with

[3]See a similar approach in [17].

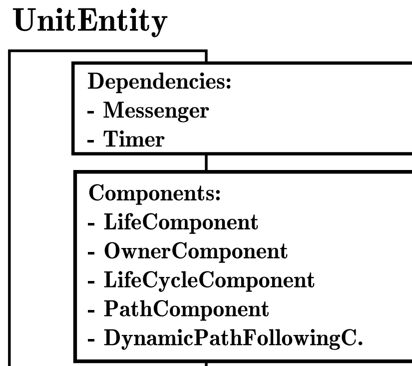


Figure 3.3: UnitEntity: Dependencies: For the position update a dependency to the *Timer*, a class which stores the delta time (= the timespan between two updates), is needed. The *Messenger* is used to inform the view about changes of the position. Components: The *PathComponent* stores the information of the path the unit should follow (a list of points). It also saves the current position of the unit. The *DynamicPathFollowingComponent* uses this information to move the unit along the path. The *LifeCycleComponent* is responsible for sending messages when the unit is initialized, updated and destroyed.

a public type to which any class can subscribe. Furthermore, the class needs to implement an interface *Receiver*. By doing this, the class implements a method which handles the subscribed message. The messenger stores the information about all the classes (receivers) which subscribed to a certain message. Incoming messages are then passed to the receivers either instantly, or during the next update of the game loop. Figure 3.4 visualizes which messages are sent when a new unit is created. The messaging system, helps to minimize the coupling between the logic and the view and also between the entities. The messenger is an example of a dependency which can be added to an entity. With the help of dependency injection there is no need to pass a reference of the messenger, other components of the entity can just use it. In this way, position updates of a unit, or changes of the ownership of a platform are directly managed by their components. Managers, components and entities are part of the game logic.

The view is responsible for displaying the entities. In other games, the view also takes care of user input. This part is not needed for this game as it is played by AI agents and runs automatically. Furthermore, the visualization was just needed during the programming phase to test if the implementation was correct. For testing the opponent modeling methods, a debug log is sufficient.

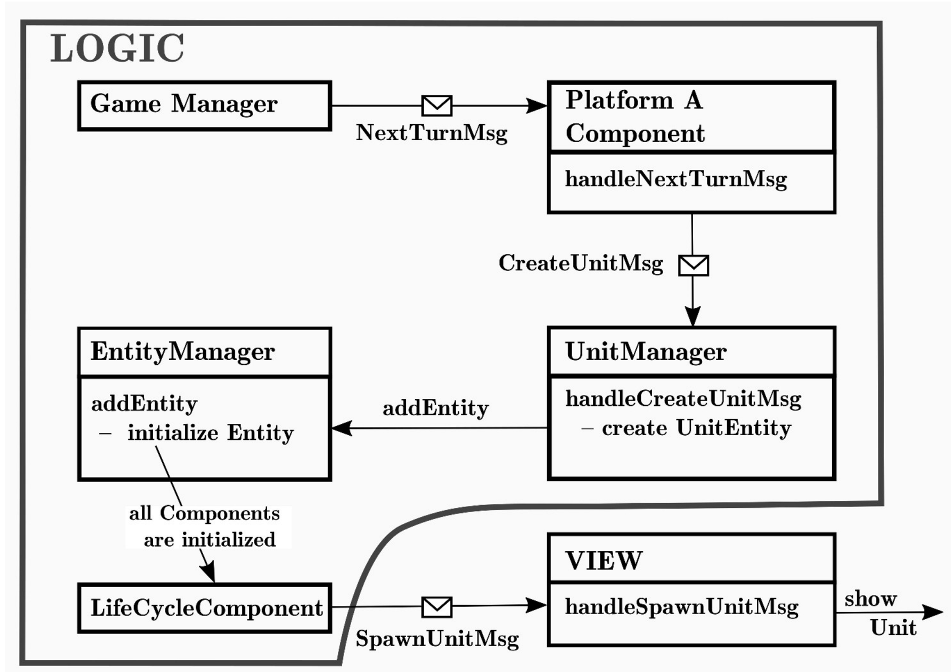


Figure 3.4: Each new turn (*NextTurnMessage* sent by the GameManager), platforms of type A which belong to the current player send a *CreateUnitMessage*. The UnitManager handles this message and creates a new unit entity. When adding it to the EntityManager, it gets initialized. Hence, all components of the entity are initialized too. The LifeCycleComponent sends a *SpawnUnitMessage* on initialization. The view handles this message and shows the unit on the platform. Inside the game logic there are some direct references, here visible between the UnitManager an the EntityManager. However, between the logic and the view, only messages are passed.

3.2.2 The Static AIs

In this game, the static AIs are goal-oriented, meaning they have several goals they strive to reach. The priority of a goal is determined by the AI itself. During the game, the AIs use a utility-function to choose the goal which is currently the most feasible. This function is defined for each goal extra. The idea is to use the information about the current game state in addition to the priority of the goal, in order to decide which action to do next. For example, the utility function of the goal to destroy the enemy base includes the path costs, as well as the number of enemies along the path. If there are too many enemy units in the way, this goal is infeasible. The goal to take over a platform of type A additionally calculates the ratio between the owned platforms of this type after the next move and those which are still available. This value tells the AI if the planned move increases the number of

owned platforms of type A. With this knowledge, the AI can decide whether to execute the move, or if an other one is more profitable.

As paths connect the platforms, the game world can be described as a graph with nodes (= the platforms) and edges (= the paths). Hence, the algorithms Dijkstra and A* (pronounced as “A star”) were implemented for pathfinding[4]. The AIs use these algorithms either to find the nearest platforms of a special type, or to find the shortest path to the castle or the enemy base.

In a game, where opponent modeling should be implemented, it is important that there is no dominant strategy. Considering, there is one strategy which ensures the player to win the game, there is no need to find out the opponent’s strategy. Therefore, the game needs balanced strategies, where a strategy is good against one, but defeated by an other. The game “Roshambo”, a.k.a. rock-paper-scissors, is an excellent example in this case. It does not matter if the player picks rock, paper or scissors, the chances of winning or losing the game are equal. For this game, three major strategies were identified and implemented for different AIs. Each strategy faces a counter.

Strategy 1: Rush

The first AI uses a strategy called *Rush*. The plan is to produce many units with little effort to charge the enemy early in the game. In RTS games this strategy is better known as “Zerg Rush”. This name comes from the popular game StarCraft® (developed and published by Blizzard Entertainment™), as the cheapest unit in this game, which is available at the very beginning of the game, is called *Zerg*. This strategy is good against an enemy who saves resources for better units. Here, this strategy defeats the enemy who tries to upgrade the units (see **Strategy 3**). The main goal of strategy 1 is to destroy the enemy’s base. To accomplish this goal as soon as possible, the shortest path from each unit to the enemy is calculated. This is done with the pathfinding algorithm A*. For this calculation also the number of enemies along this path are taken into account, as the path with the lowest movement costs is infeasible if there are too many enemies to fight against. Another goal of this strategy is to take over platforms of type A and B along the path. In this way, the AI gathers units and extra moves while moving its units towards the enemy. Figure 3.5 displays the static AI when playing strategy 1.

Strategy 2: Defend

A counter to the first strategy is to defend all the possible paths around the own base. This goal is reached by taking over platforms of type A next

[4] A detailed description of these algorithms as well as pseudo-code can be found in [16].

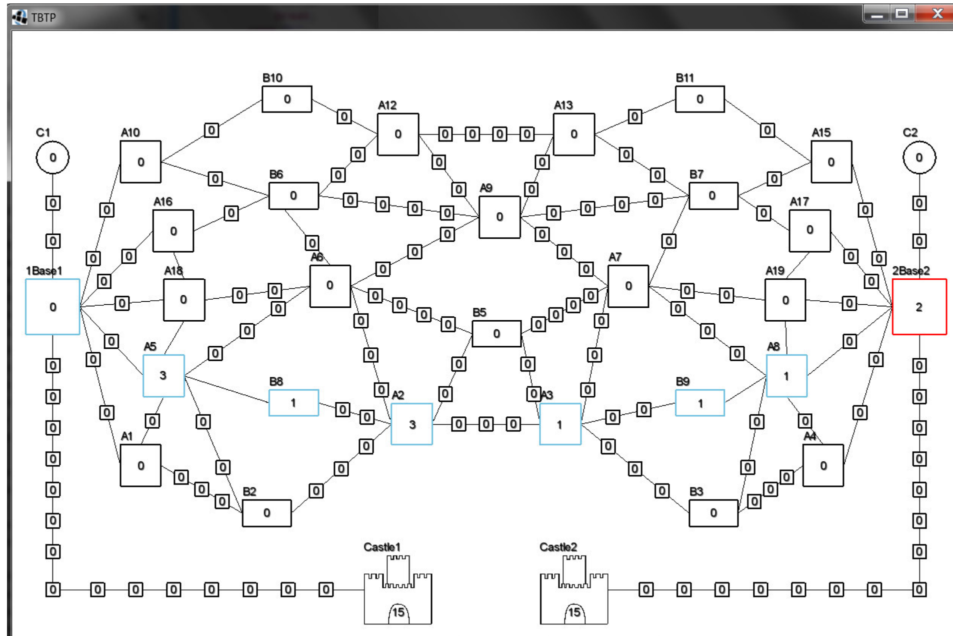


Figure 3.5: This figure displays a screenshot of the implemented game, where a static AI, owning the base on the left side of the game, is playing strategy 1. When doing so, the static AI sends its units along the shortest path to the enemy's base. The small platforms between the special platforms indicate the path costs along the path. The number of units of the static AI increase each turn for each platform of type A the AI has captured. Furthermore, the static AI's number of moves are increased by two, as it has positioned two units on platforms of type B. In order to visualize the strategy best, the player on the right side, who owns the second base, is not performing moves.

to the base. By doing so, new units are spawned on these platforms and a defensive border or wall is created. The second goal of this strategy is to win the game by taking over the platform Castle. In the game world there exists a path from the player's base to this platform. It was designed this way to ensure that the enemy player can not reach the same platform of type Castle. Otherwise it would be possible to "steal" the victory, by waiting for the enemy to destroy the neutral units. In the beginning of the game the AI starts building the wall of units around the base. When there are enough[5] units on a platform, further units are sent to the AI's base and in the next step to the Castle. Figure 3.6 displays the static AI when playing strategy 2.

[5]A fixed threshold is defined.

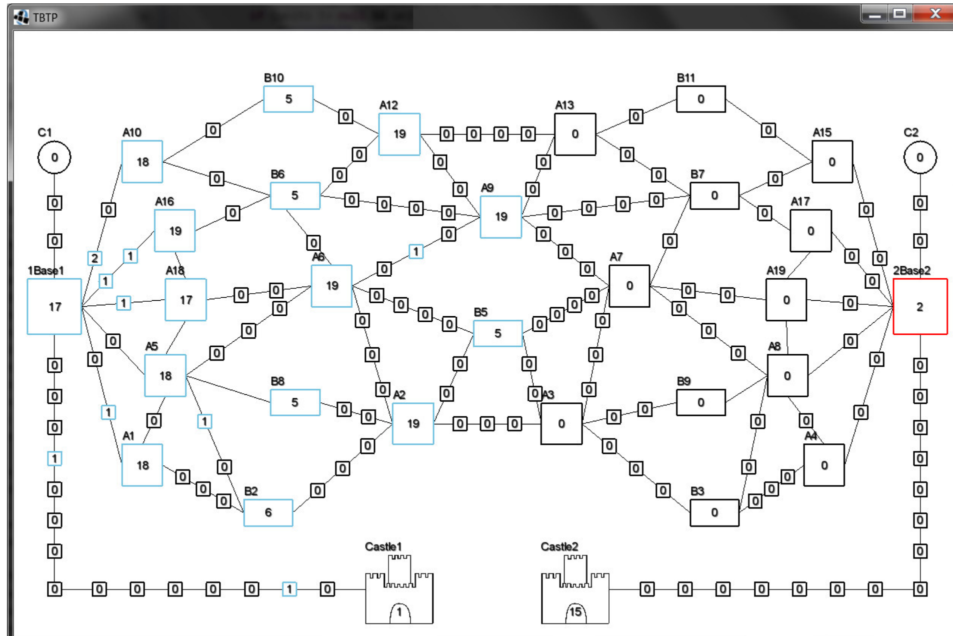


Figure 3.6: This figure displays a screenshot of the implemented game, where a static AI, owning the base on the left side of the game, is playing strategy 2. When doing so, the static AI builds a defensive wall around its base first. Afterwards, it sends units to the special platform Castle which is occupied by neutral units at the beginning of the game. The small platforms between the special platforms indicate the path costs along the path. The number of units of the static AI increase each turn for each platform of type A the AI has captured. Furthermore, the static AI's number of moves are increased by 26, as it has positioned 26 units on platforms of type B. When the static AI succeeds in taking over the special platform Castle, the AI wins the game. In order to visualize the strategy best, the player on the right side, who owns the second base, is not performing moves.

Strategy 3: Tech Rush

The third AI implements a strategy called *Tech Rush*. At first, a defensive wall is built like during the strategy of the second AI. Next, one unit is sent to a platform of type C where it stays for ten rounds. During this time no units are created on the AI's platforms of type A and the AI is vulnerable to attacks. Hence this strategy is weak against the *Rush*. However, after the ten rounds, all remaining units experience a technical upgrade. Additionally, the AI may perform extra moves per turn. In the next step, the AI starts to attack the enemy base. As the units are stronger than the enemy units with no upgrade, they are able to break through the wall of units of the pure defensive strategy. Figure 3.7 displays the static AI when playing strategy 3.

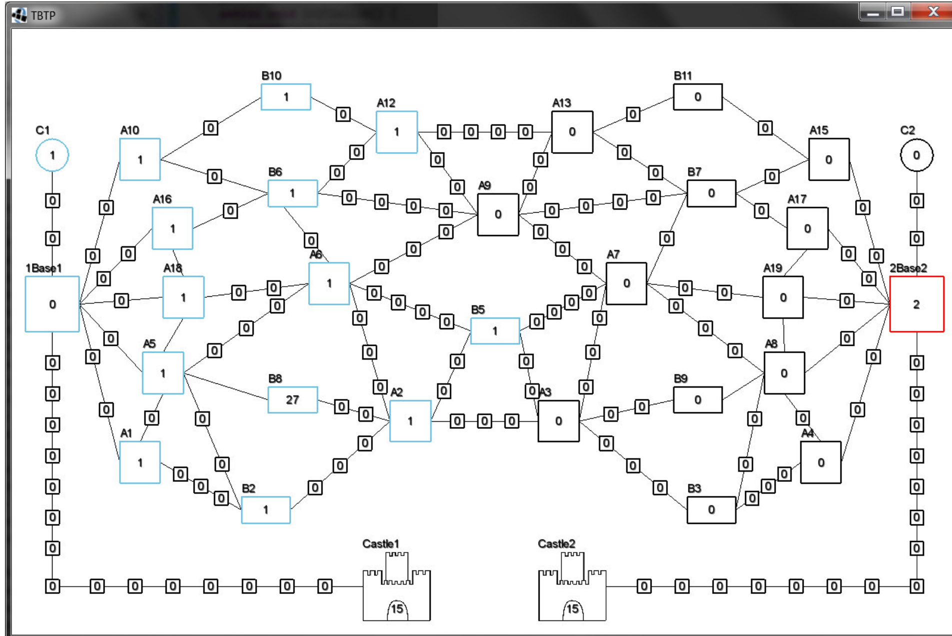


Figure 3.7: This figure displays a screenshot of the implemented game, where a static AI, owning the base on the left side of the game, is playing strategy 3. When doing so, the static AI first builds a defensive wall around its base. Afterwards, it sends one unit to the special platform C. After ten rounds, the AI and its units are upgraded and ready to attack the enemy's base. The small platforms between the special platforms indicate the path costs along the path. The number of units of the static AI increase each turn for each platform of type A the AI has captured. Furthermore, the static AI's number of moves are increased by 31, as it has positioned 31 units on platforms of type B. In order to visualize the strategy best, the player on the right side, who owns the second base, is not performing moves.

3.3 Game Balance

Balance is a term of game design and defined as "the concept and the practice of tuning a game's rules" [29]. For this game, the goal of balancing was to prevent one strategy from being dominant. In order to reach this goal, the following parts of the game were tuned:

- game world,
- initial number of player moves per turn,
- time for next upgrade on special platform C,
- type of upgrade on special platform C,
- number of neutral units on special platform Castle.

The game world was designed symmetrically. The paths from one base to the other are equal for both players. Also the number of special platforms and the paths to those platforms are equal on both sides.

At the beginning of the game, each player may perform five moves per turn. The shortest path from one base to the other is 16. Hence, at least four turns are necessary for one player to rush its enemy's base when no platforms of type B are captured. This time is considered sufficient for a player playing strategy 2 in order to build a defensive wall.

When playing strategy 3, one goal is to upgrade all units as well as the player's number of moves by capturing a platform of type C for a certain time span. The duration for this upgrade was set to ten rounds. On the one hand, this timespan time is necessary in order to give strategy 1 the chance to win the game by rushing the enemy's base. On the other hand, the defensive strategy needs more time to win and can still be defeated. Nevertheless, ten rounds is a long time. Consequently, the upgrade afterwards increases the number of moves of the player and the strength of the player's units.

Finally, there are 15 neutral units on the special platform Castle. This number was chosen in order to allow an enemy playing strategy 3 to break the defensive wall of a player playing strategy 2. When competing against strategy 1 this number is not relevant. The enemy is not able to reach the base with the normal units, no matter how long the game takes.

As explained earlier in this section, balanced strategies are important when opponent modeling should be included. Tuning the game is a difficult but necessary task for every game and requires a lot of testing.

Chapter 4

The Opponent Modeling AI

Besides knowing the rules of the game, the goal of an Opponent Modeling AI (OM AI) is to analyze its enemies' actions and identify the opposing strategies. The identification or classification can be done with pattern recognition. While this seems to be a simple task for a human being, as it is done automatically, it is not a trivial task for a computer program. This is because training is not easy as even simple things are sometimes difficult to describe, or many different descriptions match for the same object. For example, when describing a chair simply as an object where someone can sit on. Then it is clear that many other objects will be misclassified as a chair—e.g. the floor. Consequently, a more detailed description is needed. However, when describing an object with too many details, classification may be incorrect as well. For example, when describing the chair by having four legs, a barstool with only one leg or a laboratory chair with springs and wheels are not considered as chairs anymore. Describing an object with too many details is called *overfitting*. It is a known problem of pattern classification. All in all, adjusting the complexity of classifying is a difficult, but major part of this task.

In this game, the OM AI is playing in a turn-based setup against static AIs. For the identification of the opponent's strategy, different pattern classification methods were implemented. To become familiar with the particular strategies in the game, the classifiers use labeled training data[1]. This data was collected while letting the static AIs play 30 games against each other (each AI played 10 games). For the game setup explained in chapter 3, three different classifiers were tested. How each classifier uses the training data to identify the opponent's strategy is explained in detail in the following sections.

Figure 4.1 visualizes the general workflow of the OM AI. Once the static AI has finished its turn, the OM AI observes the values of prior defined

[1] Using already classified data which is labeled with the corresponding class/strategy is a.k.a. learning from example.

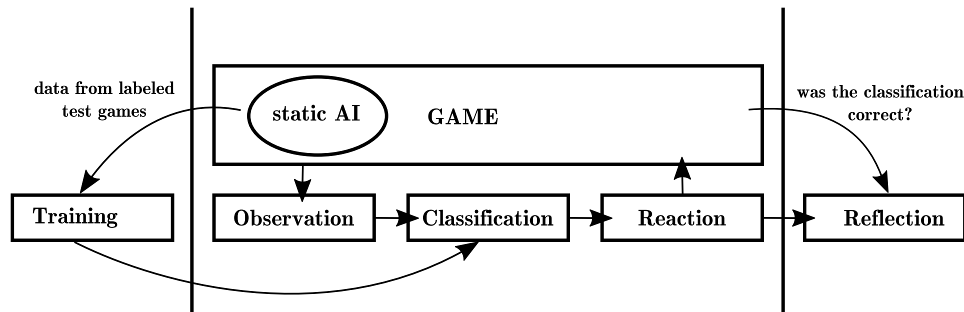


Figure 4.1: Opponent Modeling AI - Workflow: Before strategy classification can take place, the AI needs to be trained with labeled data from test games (for this setup 30 test games were used). This data includes all known strategies. During the game, the OM AI passes observation data to the classifier. The training data, as well as the observations are used for the classification. If a strategy could be identified, the AI reacts by playing the counter strategy. At the end of the game, the results are evaluated.

game features. Then this information, as well as the labeled data from the test games, is passed to the classifiers. As a result, the classifiers return for each strategy the probability of facing it. In the end, the OM AI chooses to play the counter strategy to the one with the highest probability.

Knowledge about the domain is necessary in order to choose the appropriate features for the classification. This process is also called *feature extraction*. Certainly, not all the information of the game is needed for the identification of the opponents' strategy. Hence, the following four game-features were selected:

1. number of special platforms,
2. number of attacks,
3. shortest distance between a unit and the special platform C,
4. shortest distance from a unit to the special platform Castle.

These features were chosen, as the values of them differ for the different strategies. For example, when playing strategy 1 (Rush) the number of attacks is high, compared to the defensive strategy. Furthermore, the distance to the special platform C will only decrease, when the opponent tries to upgrade its units and hence follows the third strategy. The values of each feature are observed after each player's turn. There were more features in the game which could have been used for the identification of the strategies, like the number of units, or the distance to the enemy's base. However, these features turned out to have just a small impact on the correct classification.

The implemented classifiers are: Naive Bayes', CART and k-nearest-neighbor (kNN). The basics of these classifiers are described in the following

sections. Further details on the implementation of the classifiers can be found at the end of this chapter. The performance of the tested classifiers is evaluated in chapter 5. The main ideas as well as the mathematical background for the implemented classifiers were taken from Duda et al. [9]. The OM AI and the classifiers were implemented in Java.

4.1 Naive Bayes' Classifier

The first classifier uses the Bayes' rule,

$$P(Y \mid X) = \frac{P(X \mid Y) \cdot P(Y)}{P(X)},$$

to calculate the probability of a certain *state of nature* Y (in this case strategy 1, 2 or 3), given observed values X (features). This probability $P(Y \mid X)$ is also called the *a posteriori probability* (or *posterior*). It can be determined by multiplying the *class-conditional probability density* function (or *likelihood*) $P(X \mid Y)$ with the *a priori* (or *prior*) probability $P(Y)$ and dividing the result by the *evidence factor* $P(X)$,

$$posterior = \frac{likelihood \cdot prior}{evidence}.$$

When playing against an opponent for the first time, there is no prior knowledge about the opponent's tendency for choosing a specific strategy. Therefore the *a priori probability* of each *state of nature* $P(s_i)$ is considered equally distributed and hence 1/3 for each strategy. Adding up $P(s_1)$, $P(s_2)$ and $P(s_3)$ results to one[2]. This prior knowledge about the strategies could be updated after a few games, playing against the same opponent.

The *likelihood* $P(f_j \mid s_i)$ describes the probability of observing a certain feature value f_j, given the strategy s_i played. For example: $P(f_1 \mid s_1)$ and $P(f_1 \mid s_2)$ determine the difference between the number of special platforms when playing the first strategy and the second one. For calculating the likelihood of a feature given a certain strategy, the labeled data from test games has to be prepared. This is done by using **binning**—a data clustering mechanism, where the given data is sorted in ranges, so-called bins[11]. Once a new value is added to the test data, the counter of the bin with the corresponding range is increased. With these bins, the likelihood of each feature given a certain strategy can be evaluated. Figure 4.2 shows how the data of round 15 and feature one (number of special platforms) is prepared and the likelihood is calculated. Defining an appropriate size of the bins is important. If the range is too small, there is the danger of *overfitting* the bins to the test data. If the range is defined too big, the results may be not

[2] Assuming that the identified strategies are the only ones in this game and further ones are just similar to them.

Feature 1: # of special platforms

	s_1	s_2	s_3
15–18	0	1	0
12–15	0	8	7
9–12	0	1	3
6–9	0	0	0
3–6	7	0	0
0–3	3	0	0
$\sum$	10	10	10

Round 15:

insert values:
s_1: 2, 3, 3, 3, 3, 5, 2, 2, 3, 3
s_2: 13, 14, 13, 14, 12, 11, 14, 15, 13, 14
s_3: 13, 13, 12, 12, 13, 11, 11, 13, 12, 11

$P(f_1 = 11 \mid s_1) = 0/10$
$P(f_1 = 11 \mid s_2) = 1/10$
$P(f_1 = 11 \mid s_3) = 3/10$

Figure 4.2: Binning feature one, in round 15: The values from the test games are sent to ranged bins. There are bins for each strategy. The ranges of each bin are fixed ([0-2], [3-5], ...). The *class-conditional probability density* function is determined by reading the value from the bin with the required range and dividing it by the total count of values of the corresponding strategy. For example: $P(f_1 = 11 \mid s_1) = \frac{0}{10}$, $P(f_1 = 11 \mid s_2) = \frac{1}{10}$ and $P(f_1 = 11 \mid s_3) = \frac{3}{10}$.

accurately enough. After examining the test data properly, the ranges of the bins were set to three values each.

As the features are considered to be independent, the likelihood for all feature values given a certain strategy can be easily calculated. The strategy's prior probability is multiplied by the individual likelihoods of the features given the strategy:

$$P(f_1, f_2, f_3, f_4 \mid s_i) = P(f_1 \mid s_i) \cdot P(f_2 \mid s_i) \cdot P(f_3 \mid s_i) \cdot P(f_4 \mid s_i).$$

The so-called *evidence factor* is the probability for the combined occurrence of the feature values $P(f_1, f_2, f_3, f_4)$. It is calculated by summing over the product of the likelihood of the features, given a certain strategy and its prior probability. N represents the number of strategies in the game[3]:

$$\sum_{i=1}^{n} P(f_1, f_2, f_3, f_4 \mid s_i) \cdot P(s_i).$$

The evidence factor is responsible for scaling the results to make the posterior probabilities sum to one. To decide which strategy is more likely when certain feature values are observed, the evidence factor is not needed.

[3]For this setup $n = 3$.

Putting all together, it is possible to calculate the probability of each strategy after observing certain independent feature values, the *posterior* probability $P(s_i \mid f_1, f_2, f_3, f_4)$, by inserting into Bayes' rule,

$$P(s_i \mid f_1, f_2, f_3, f_4) = \frac{P(f_1, f_2, f_3, f_4 \mid s_i) \cdot P(s_i)}{P(f_1, f_2, f_3, f_4)}.$$

Each round the OM AI observes the current feature values of the opponent. This information is passed to the Bayes' classifier. Once the *posterior probability* for each strategy is calculated, the OM AI can decide which counter strategy to play. For simplicity let $F = \{f_1, f_2, f_3, f_4\}$. If $P(s_1 \mid F) > P(s_2 \mid F) \wedge P(s_1 \mid F) > P(s_3 \mid F)$ the opponent is probably playing the first strategy. If $P(s_2 \mid F) > P(s_1 \mid F) \wedge P(s_2 \mid F) > P(s_3 \mid F)$, the OM AI decides that its enemy is playing strategy 2. Otherwise, the opponent is most likely to play strategy 3.

4.2 CART Classifier

The second classifier builds a decision tree in order to find out which strategy the OM AI is facing. As the name might already tell, the tree consists of query or decision nodes. At each node, the decision about which branch to follow is made. Branches or links lead from one node to its descendent nodes. Decisions are made directional from the root node (at the top) to a leaf node[4] (at the bottom). This also implies that decisions that are made earlier in the tree do have an effect on the subsequent decisions. The simplest form of a decision node is a binary node with just two branches.

As stated in Duda et al. [9] decision trees have some advantages compared to other classification mechanisms. On the one hand, classifications, performed by traversing a decision tree, are easy to interpret. This is because each decision for a certain strategy represents a conjunction of all decisions made from the root to the leaf node. On the other hand, classification can be done very quickly, as no difficult calculations have to be performed (especially when the tree consists of binary nodes only). Furthermore, expert knowledge about the environment can be included when building the decision tree. This often leads to quick and reliable classification.

The Classification and Regression Trees (CART) algorithm is used to build the tree by finding the best split at each decision node. Again the four features, specified at the beginning of this chapter, are used for the classification. In this case, they are used as query nodes. Figure 4.3 displays how the tree can look. The subsequent nodes of a feature are determined by hand. Here, expert knowledge about the domain is included. The first split at level one in the query tree is the feature one (number of platforms). If the resulting subset of the first level contains values of strategy 1, the

[4] A node which does not have any subsequent nodes.

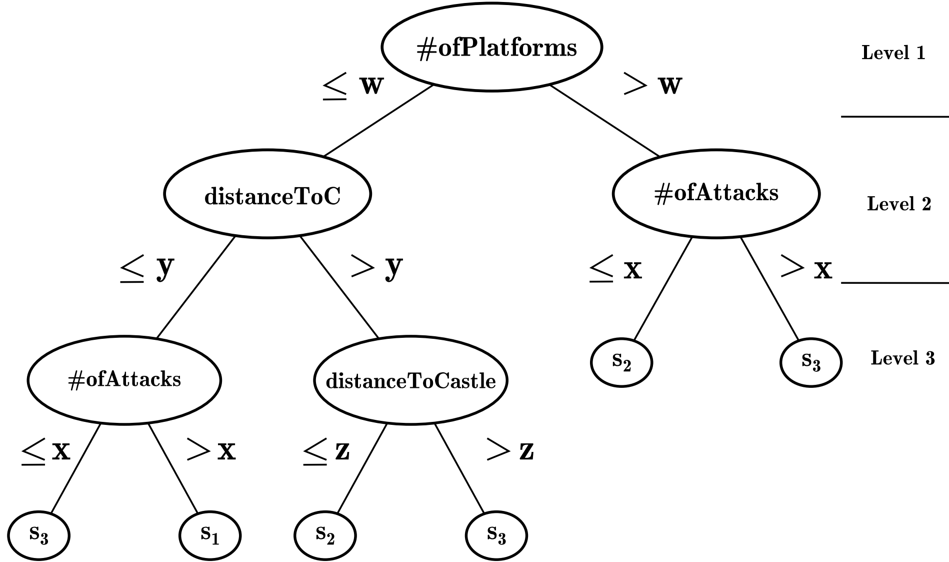


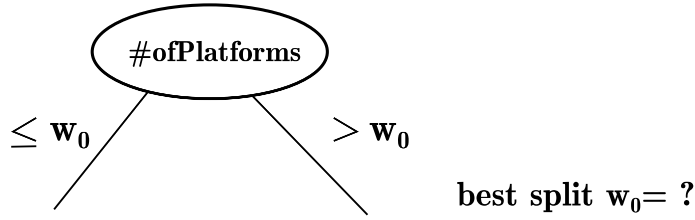
Figure 4.3: This is an example of a **decision tree**, built with the four features as query nodes. The variables **w**, **x**, **y** and **z** indicate the split values at the nodes. After the observation of the opponents feature values, the tree is traversed by answering the queries. At the leaf nodes, the decision for a particular strategy is made. This is done with the help of the labeled test data, which is reduced after each split. For example, when observing a small number of platforms ($f_1 \leq \mathbf{w}$), the decision is made to follow the left branch of the tree. Hence, the test data with a value of the feature one greater than **w** can be dismissed.

query node of the second level is the feature three (distance to platform C). Otherwise, the query node is the feature two (number of attacks). At the third level of the decision tree, the query node is chosen similarly. When there are still values of the first strategy in the subset after splitting the test data twice, the query node is the feature two. Else, the split at level three is done for the fourth feature (distance to platform Castle).

In order to determine the best split at each query node and to decide which strategy to choose at a leaf node, the labeled test data is needed. The best split value of a feature is evaluated by calculating the *sum-of-squared errors* (SSE) for each possible split between two values of the test data (visualized in Figure 4.4). This has to be done, as the values of the features are numerical[5].

As the split should be binary, the data $\mathcal{D}$ is partitioned at each possible split into two subparts $\mathcal{D}_1$ and $\mathcal{D}_2$ with counting n_i values. Then the mean

[5]The calculation would be different for categorical values.



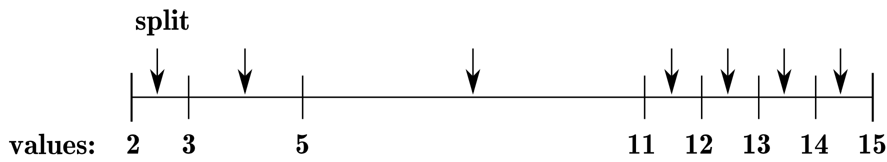


Figure 4.4: Determine the best split $\mathbf{w}_0$ for the feature one (number of platforms), by calculating the *sum-of-squared errors* (SSE) at each possible split. The above values are taken from the test data in round 15.

$\mathbf{m}_i$ of both parts is determined by

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{\mathbf{x} \in D_i} \mathbf{x}. \tag{4.1}$$

In a next step, the SSE can be calculated by

$$J_e = \sum_{i=1}^{c} \sum_{\mathbf{x} \in \mathcal{D}_i} \parallel \mathbf{x} - \mathbf{m}_i \parallel^2, \tag{4.2}$$

where $c = 2$ as there are only two subparts. When determining the SSE, the total squared error of the n samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$ is calculated by summing the squared lengths between $\mathbf{x}$ and the mean vector $\mathbf{m}_i$ in $\mathcal{D}_i$. As stated before, the SSE is calculated at each possible split. At the end, the one with the smallest SSE is selected as the best split of the corresponding feature. The best splits for all features can be determined before the game starts. Hence, when the OM AI observes the values of the features, the decision tree can be traversed immediately. Afterwards, the remaining values in the test data are counted and the strategy with the most values is determined. The tree can be pruned earlier, if there is just one strategy remaining in the resulting subset after splitting.

Classifying with CART is known to be a *greedy method*, as the decisions are optimized locally at the nodes and not globally for the whole tree. This can result in big trees. However, that is just an issue if the branches are determined automatically, which is not the case for this testing (as expert knowledge was included).

4.3 kNN Classifier

The last tested classifier uses the k-nearest-neighbor (kNN) rule for determining the strategy of the opponent. In this process, the values of the test data, as well as the observed values are mapped to a four-dimensional space. The distances to all neighbors of the observed values are evaluated next. The variable **k** denotes the number of nearest neighbors which are filtered from the test data. At the end, the strategy with the most values in the subset of nearest neighbors is identified as the strategy of the OM AI's opponent. Figure 4.5 visualizes this process for a specific round. Classifying with the kNN rule is simple and explained in the tutorial of Teknomo [32] by the following five steps:

1. Set the number of **k** neighbors for the classification.
2. Calculate the distance between the observed values and those from the test data.
3. Determine the **k** neighbors with the smallest distance to the input values.
4. Count the number of instances in the subset for each strategy.
5. Label the observed values with the strategy with the most instances in the subset.

When determining the variable **k** it is important to choose a value which avoids ties. Therefore a value, which is not a multiple of the available classes (in this case: $\mathbf{k} \notin \{3, 6, 9, \dots\}$), needs to be selected.

Before calculating the distance between the test data and the input values, the test data needs to be prepared again. If the pure values of the test data would be used, some features would have more impact on the distance than others. This is because the values of each feature do have different ranges, e.g. the distance to platform C is usually between 0 and 7, whereas the distance to the Castle is at the beginning 33. Hence, the test data, as well as the observed values, are standardized before the distances are calculated. For the standardization, the mean for each round and feature $\mathbf{m}_i$ is determined by

$$\mathbf{m}_i = \frac{1}{n} \sum_{\mathbf{x} \in f_i} \mathbf{x}.$$

Next, the standard variance σ_i is calculated with the help of $\mathbf{m}_i$ by

$$\sigma_i = \frac{1}{n} \sum_{\mathbf{x} \in f_i} \parallel \mathbf{x} - \mathbf{m}_i \parallel^2,$$

for each feature and round. Finally, each value of the features the standardized value

$$\frac{\mathbf{x} - \mathbf{m}_i}{\sigma_i}, \tag{4.3}$$

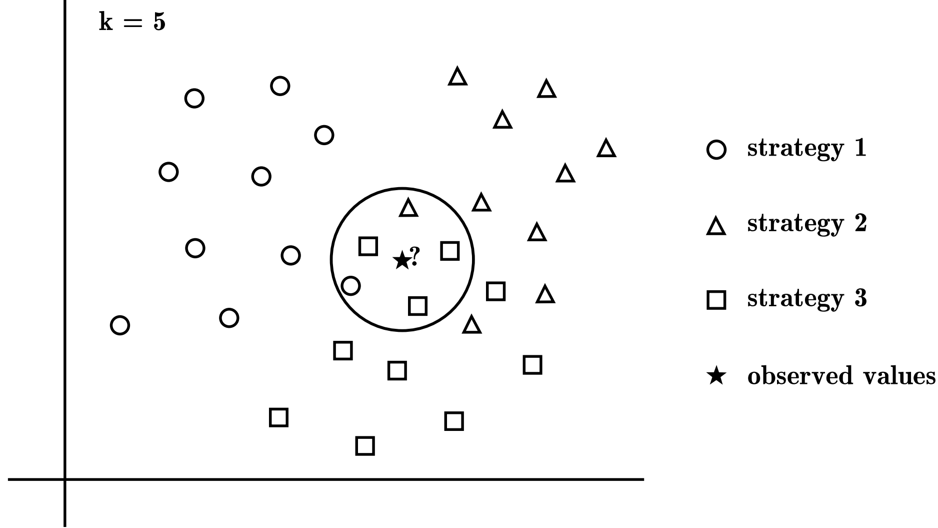


Figure 4.5: The symbols ◯, △ and □ represent all values of the test data belonging to a certain st[illegible]h symbol denotes values of the four features observed in a test game in the same round. When the OM AI observes values of its opponent in the specified round, here visualized with the symbol ★, the k-nearest-neighbors (here: five) are determined. The incoming values are then labeled with the strategy of the majority of its nearest-neighbors. In this case, ★ will be labeled with strategy 3 (□).

is determined. Here, $\mathbf{x}$ denotes a value of feature i. The mean $\mathbf{m}_i$ is subtracted and the resulting value is then divided by the feature's standard variance σ_i. After the standardization, all feature values do have their mean at zero and a standard derivation of one.

In order to evaluate the distance between the test data and the new values, the *Euclidean distance* is calculated [30]. For an easy and fast calculation, the squared distance is used. Let $\mathbf{F} = (f_1, f_2, f_3, f_4)$ be a point of the test data in the *Euclidean n-space* and $\mathbf{O} = (o_1, o_2, o_3, o_4)$ another point representing the observed values. Then the squared distance d can be calculated by

$$
\begin{aligned}
d(\mathbf{F}, \mathbf{O}) = d(\mathbf{O}, \mathbf{F}) &= (f_1 - o_1)^2 + \cdots + (f_n - o_n)^2 \\
&= \sum_{i=1}^{n} (f_i - o_i)^2,
\end{aligned}
$$

where $n = 4$ as there are four features used for the classification.

After determining the distances between all values, the subset of the k-nearest-neighbors is built. In the last step, the strategy with the most instances in the subset is selected as the strategy of the OM AI's opponent. Hence, the AI is then able to adopt its own strategy.

4.4 Implementation Details

Each classifier needs some training data before any classification can be done. While the Naive Bayes' classifier uses the data for calculating the posterior probability for each strategy, the CART classifier determines the best split at each node of the decision tree. Finally, the kNN classifier computes the Euclidean distance between the incoming values and the labeled ones from the training data. As mentioned before, the test data is gathered by letting the static AIs play games against each other[6].

During training, the AIs send a message (*StatisticsMessage*) with the values of the features and their strategyID to the class *StrategyStatistics*. This is done for each turn. The values of a feature are stored in an instance of the class named *FeatureValues*. The class *Round* stores four instances of this class. Furthermore, the identifier of the strategy (strategyID), played when observing the certain values, is stored in a list in *Round* as well. The class *StrategyStatistics* saves a list of *Round* instances, one instance for each game round. The following code lines show the three classes which store the information about the features:

```
public class StrategyStatistics implements Serializable, StatisticsReceiver {
  private List<Round> dataPerRound;

  @Override
  public void handleStatisticsMessage(StatisticsMessage message) {
    // create new Round or store values for existing Round
    // call insertValues method of Round with the values and
    // the corresponding strategyID as parameters
  }
}

public class Round implements Serializable {
  private List<FeatureValues> features;
  private List<Integer> strategyIDs;
  public static int featureCnt = 4;

  // expects an double[ ] of size featureCnt
  public void insertValues(double[] values, int strategyID) {
    // call insertValue method for each FeatureValue instance
    // save strategyID for current inserted values
  }
}

public class FeatureValues implements Serializable {
  private List<Double> values;
  public void insertValue(double value) { ... }
}
```

To be able to load the data efficiently when it is needed, it is serialized and saved in a file. Hence, the classes *StrategyStatistics*, *Round* and *FeatureValues*

[6]For this setup, each static AI played ten games in total.

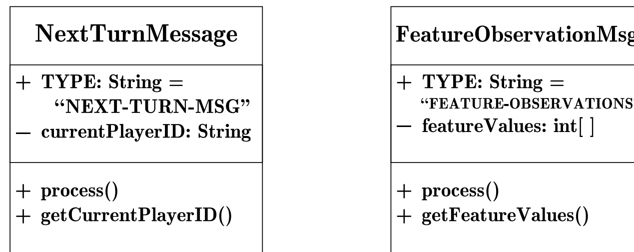


Figure 4.6: The *NextTurnMessage* stores the information about the current player. The *FeatureObservationMessage* contains the player's current values of the four features (number of special platforms, number of attacks, shortest distance to special platform C and shortest distance to platform Castle). Both messages are sent after each turn. The public property *TYPE* is used by the *Receivers* to subscribe to messenger in order to receive these methods.

implement the Java interface *Serializable*.

The communication between the OM AI and its environment (the game) is done via messages. The *NextTurnMessage* and the *FeatureObservationMessage* are the most important messages the AI listens to. Figure 4.6 displays the two classes. The first one contains the information about the current player. It is sent whenever a player is done with moving units around. The *FeatureObservationMessage* is also sent after a player's turn. It provides information about the four feature values. The OM AI uses the observations of its opponent to identify the strategy. Both messages do have a public property *TYPE*. Classes, which implement the corresponding interface (like the OM AI), use this property to subscribe to the messenger in order to receive the messages.

As described in section 4.1, the test data is prepared for the Naive Bayes' classifier by putting the values into bins with ranges. The range of a bin is fixed to three[7]. During the game, the classifier receives the current feature values from the OM AI. The likelihoods of each feature value given a certain strategy can be looked up in the prepared bins. These likelihoods are multiplied with the a priori probability of the strategy ($P(s_i) = 1/3$). Doing this for all three strategies and summing up the results yields the evidence factor. The last step is to insert the parts into the Bayes' formula and determine the strategy with the greatest posterior probability. Plain Java Lists and Arrays were used for the data structure of the bins.

For the implementation of the CART classifier, the decision tree for each round is built in advance. For the first level of the tree, the best split for the first feature is determined. This is done by calculating the SSE for each possible split[8] between the distinct values of the sorted test data. For

[7]This range was chosen after gaining experience with the test values.
[8]split = value1 + (value2 - value1) / 2

determining the SSE, the values of the test data are split and stored in two separate lists. The values to the left side of the split as well as the values to the right side of the split are summed and counted in order to determine the mean (see equation 4.1) of each list. Next, for each value in the two lists, the squared errors are determined with the corresponding mean and summed at the end (equation 4.2). The result is the SSE of the test data for feature one of a split. Doing this for all possible splits, the best split (with the smallest SSE) can be determined.

As mentioned before, the subsequent query nodes are influenced by the decisions made earlier. Hence the second level is built by using the resulting lists after splitting the test data. If there are only values of one strategy left in a list, the node at this branch becomes a leaf node. Otherwise, the feature for the query node of the second level is determined. If a list still contains values of strategy 1, the next feature used as query node is the third one (distance to platform C). Else, the query node at level two will be the feature "number of attacks". Again, the best split is determined for the selected feature. Afterwards, the lists of the new branches are built.

The third level of the tree is built similarly to the second level. If only values of one strategy are left in a list, leaf nodes are created. If there are more strategies left, and one of those is strategy 1, the query node of the third level is the second feature (number of attacks). Otherwise, the fourth feature (distance to platform Castle) becomes the new query node. Once more, the best split is calculated and the new branches are established. As the tree structure is determined by hand, there are no more levels after the third split.

During the game, the OM AI passes the observed features to the CART classifier. The predefined tree of the current round is then traversed by comparing the values with the best splits of the features at the specific levels. At the leaf nodes, the strategy with the greatest amount of values remaining in the list is determined as the current strategy of the AI's opponent. Hence, the OM AI starts to play the counter to this strategy.

As mentioned in section 4.3, the test data needs to be prepared differently for the kNN classifier. First, for each feature and round, the mean $\mathbf{m}_i$ and standard derivation σ_i is calculated. These values are used to standardize the test data (see equation 4.3). When the game begins, the OM AI passes the observed values to the kNN classifier. These values are then standardized with the feature's corresponding mean and standard derivation, like the test data before. In a next step, the *Euclidean distances* between all values from the test data and the observed ones are calculated. The distance and the corresponding label of the test data (the strategy) are stored in a class called *FeatureDistance*. Afterwards, the list of *FeatureDistance* instances is sorted by the static method *Collections.sort(featureDistances)*. This is possible, as the class *FeatureDistance* implements the Java interface *Comparable* and its function *compareTo*. Figure 4.7 shows the class in the Unified Modeling

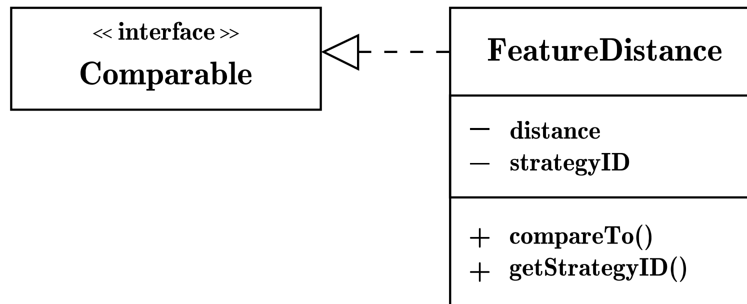


Figure 4.7: The class *FigureDistance* holds the Euclidean distance, calculated between the feature values of the current round of a test game and the observed values. Furthermore, the strategyID of the strategy which produced the test values is stored as well. The class implements the interface *Comparable* in order to be able to sort instances of *FigureDistance*.

Language. In the end, the first **k** elements are taken from the list. As they are labeled, the strategy with the majority can be determined. In Duda et al. [9] it is mentioned, that a large value of **k** effectively can reduce the error rate, however only if there is a lot of training data available. If the value of **k** is chosen too small, there might be misclassification due to noise[9] in the data. Hence, determining the appropriate value for **k** requires at least some testing.

All three classifiers do have an abstract superclass which is responsible for the AI's initialization, and cleanup tasks. Furthermore, *NextTurnMessages* are handled and the current strategy is executed. The subclasses, which implement a specific classifier, listen to the previously mentioned *FeatureObservationMessage*. Once the subclasses identify the strategy of the AI's opponent, the information is passed to the superclass. As a result, the corresponding counter strategy is played.

[9]Noise in this game appears due to some randomness of the specific strategies during actual game play.

Chapter 5

Tests and Evaluation

The classifiers, which were explained in chapter 4, were tested by playing games against static AIs. The subsequent sections deal with the results of the testings. Each classifier is evaluated individually first. At the end of this chapter, their performance is compared. The scope of the testing consists of the following points:

1. Simplicity:

   Was it easy to understand the classifier?

   Where there difficulties in the implementation?

2. Performance:

   Did the OM AI win the games?

   How accurate was the classification per round?

   What are the reasons for misclassification?

Whereas the performance measures can be taken from the results of the games, the simplicity of the classifiers is a subjective perception of the author. Furthermore, this point is also dependent on mathematical background knowledge and programming skills.

The OM AI in this game is used to challenge its opponents and to encourage them to alter their strategy. Otherwise, if they stick to one play-style, the OM AI is able to identify the strategy and win the game. In fact, the implemented OM AI will never outperform a human player, or a dynamic enemy (somebody who switches strategies). However, that is not the goal here. The requirements of the classification methods for this game are clear: the OM AI should be able to win any game against the static AIs. Furthermore, the decision for playing a specific counter strategy has to be clear and misclassification minimal. Nevertheless, as classification is not a trivial task it might not always be correct or accurate enough.

For testing the performance of the classifiers, the OM AI used them to classify its opponents in games against static AIs. As mentioned in chapter 4, the classifiers were trained with data from 30 games, labeled with the correct

strategies. The goal of the classifier was to find out which of the three possible strategies (explained in detail in chapter 3) the OM AI is most likely facing. In order to measure the performances individually, only one classifier was used to identify the opponent's strategy per game. In total, a classifier was tested in 15 games, five against each strategy. The results were written to a simple *Comma-separated values* (CSV) file for further evaluation. Of course, the information about the winners of the games is stored. Additionally, the intermediate results per round were written to the CSV file as well. Such a result consists of:

- round number,
- observed feature values,
- calculated probability for each strategy,
- identified strategy and
- actual strategy.

The results are now evaluated in detail for each classifier separately.

5.1 Naive Bayes' Classifier

For a classifier using the Bayes' Rule it is important that the required probabilities are known. If that is not the case, the probabilities have to be estimated from test data. This is the first challenge in order to fully understand how the classifier works. In this setup, the test data was gathered from games of all different classes/strategies. The *a priori probability* of each strategy is more or less easy to determine. If all possible strategies are known and equally dominant and the opponent is unknown, the probability for each strategy is equally distributed too. If there is data from previous games against the same opponent available, the a priori probability can be updated by including information about the favored strategy of the opponent.

For calculating the likelihood of the observed feature values, given a certain strategy, the test data needs to be structured. In this setup, the data was organized in ranges/bins. The challenge here was to determine the "best" size of the ranges in order to prevent overfitting or inaccurate results. Defining appropriate ranges requires knowledge about the feature values. For this game the sizes of the ranges were set to three after gaining experience with the data and some testings with other ranges. For more accuracy, the best size could also be determined by calculating the SSE (see equation 4.2) for different ranges and choosing the one with the smallest result. However, for a small game like the one presented here, estimating the "best" size is perfectly fine.

Determining the right data structure for the structured feature values was the only challenge when implementing the Naive Bayes' classifier. As

mentioned before, the values were organized in bins. This was done for each feature individually. A bin for a specific range and strategy became a simple *Integer* value. This value represents the number of values which were observed inside the range and for one strategy. Hence, the bins for a specific range are organized in an *Array* of *Integer* values, where the size of the *Array* is three, as there are three strategies. All ranges are then stored in a Java *List*. The first entry in the list represents the observed values zero, one and two. The second one, the values three, four and five, and so on. Additionally, the total number of observed values for each strategy is stored too. This value is used to determine the likelihood of observing a value by dividing the value from the bin of a strategy by the total number of observed values of the same strategy.

The rest of the implementation was rather simple. The probabilities were inserted into the formula and the strategy with the greatest posterior probability was determined.

When using the Naive Bayes' classifier for identifying the strategy of the opponent, the OM AI did win all 15 games. Besides this excellent result, the classification was not correct for every round. Table 5.1 visualizes the intermediate classification results. The classifier was able to identify the first strategy in 90% of all rounds. It never misclassified the strategy for an other one. However, in 10% of the rounds it was not able to identify any strategy. When playing against strategy 2, the classifier performed just slightly worse. In almost 83% of all rounds the classification was correct. There were some misclassification for strategy 1 and 3. Again, the classifier was sometimes not able to classify any strategy. Facing the third strategy seemed most challenging for the classifier, as it was only able to correctly identify the opponents' strategy in about 62% of all rounds. In the most cases of misclassification, the classifier identified strategy 1 (ca. 24%).

Having a closer look at the test data it is clear that the values of the features of all strategies do not differ much during the first few rounds. Hence, the misclassification for an other strategy mainly occurred at the beginning of the game. Table 5.2 visualizes the intermediate results without the first five rounds. Except for some misclassification for strategy 2 when playing against the third one, the output shows that the classifier performed better after the first five rounds.

The fact, that the classifier is sometimes not able to identify any strategy may have different reasons, as the number of misclassification did not decrease after the first rounds. When considering the example in figure 5.1, the source of error in the displayed round seems to be the feature one. In this game, the OM AI played against strategy 3. In round 10 the classifier was not able to identify any strategy. On the one hand, the feature itself could turn out to be less informative than the other ones. In that case, the process of feature extraction should be reviewed. On the other hand, the size of the ranges of the bins could have been not fitting. As visible in the

Table 5.1: Average intermediate classification results of the Naive Bayes' classifier per game. The games against strategy 1 lasted about 38.2 rounds, those against strategy 2 28.2 rounds and playing against strategy 3 took about 15.2 rounds. The rows indicate the actual strategy and the columns represent the classified strategies. Values in the last column (X) indicate that the classifier was not able to identify any strategy. The bold values in the diagonal display the correct classifications.

Average intermediate results per game:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**34.6**	0.0	0.0	3.6
Strategy 2	1.4	**23.2**	0.2	3.6
Strategy 3	3.6	0.6	**9.4**	1.6

Average intermediate results per game in percentage:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**90.58**	0.00	0.00	9.42
Strategy 2	4.93	**81.69**	0.70	12.68
Strategy 3	23.68	3.95	**61.84**	10.53

Table 5.2: Average intermediate classification results of the Naive Bayes' classifier per game without the first five rounds. The rows indicate the actual strategy and the columns represent the classified strategies. Values in the last column (X) indicate that the classifier was not able to identify any strategy. The bold values in the diagonal display the correct classifications.

Average intermediate results per game:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**29.6**	0.0	0.0	3.6
Strategy 2	0.0	**19.8**	0.0	3.6
Strategy 3	0.0	0.6	**8.0**	1.6

Average intermediate results per game in percentage:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**89.16**	0.00	0.00	10.84
Strategy 2	0.00	**84.62**	0.00	15,38
Strategy 3	0.00	5.88	**78.43**	15.69

Round 10:
observed values: $f_1 = 11$, $f_2 = 2$, $f_3 = 5$, $f_4 = 33$

Bins f_1:

Bin	s_1	s_2	s_3
> 15	0	0	1
12–15	0	5	9
9–12	0	3	0
6–9	0	2	0
3–6	10	0	0
0–3	0	0	0

Bins f_2:

Bin	s_1	s_2	s_3
> 15			
12–15			
9–12			
6–9	2	0	0
3–6	8	1	0
0–3	0	9	10

Bins f_3:

Bin	s_1	s_2	s_3
> 15			
12–15			
9–12			
6–9			
3–6	10	10	3
0–3	0	0	7

Bins f_4:

Bin	s_1	s_2	s_3
> 33	10	0	10
30–33	0	10	0
.			
.			
.			
0			

Figure 5.1: Here the misclassification in a game in round 10 playing against strategy 3 is visualized. The marked bins are the ones where the likelihoods for calculating the probabilities are taken from, as the observed feature values are inside the bins' ranges. It is clear, that the error for identifying the correct strategy occurs at feature one as the likelihood of observing value 11 given strategy 3 is zero. As a result, the posterior probability of strategy 3, given all the observed values is zero too. Additionally, the posterior probabilities of the other strategies are zero as well. That is because the likelihood for strategy 1 and feature one with value 11 and the likelihood of strategy 2 and feature four with value 33 result in zero.

example in the figure, the observed values just slightly drop out from the bins where classification would have been possible. The observed value for feature one is 11. The next bin with values for strategy 3 starts at value 12.

As the OM AI wins the game in the end, some misclassification is of little relevance here. However, if the OM AI would lose according to this error, it needs to be resolved. Another cause of this error might be that the test data for training the classifier is to one-sided or too little. This error could be fixed by including more test games in the training.

5.2 CART Classifier

Identifying the opponent with a decision tree seems an easy task once the tree has been built. However, building the tree and determining the best split at each node was a bit more difficult than gathering probabilities from test data. Therefore the main challenge of this classifier was rather to understand the creation of the decision tree, than the implementation of the formulas.

As mentioned in the previous chapter, the decision which feature should be used for a node was done by hand. Besides using expert knowledge about the domain, several arrangements were tested in order to produce clear results. That there can be a relevant difference in choosing to split at one feature before an other is visualized in figure 5.2. For better understanding, just two features were selected for this example. The values of round 15 from the test data of the first (number of special platforms) and the second

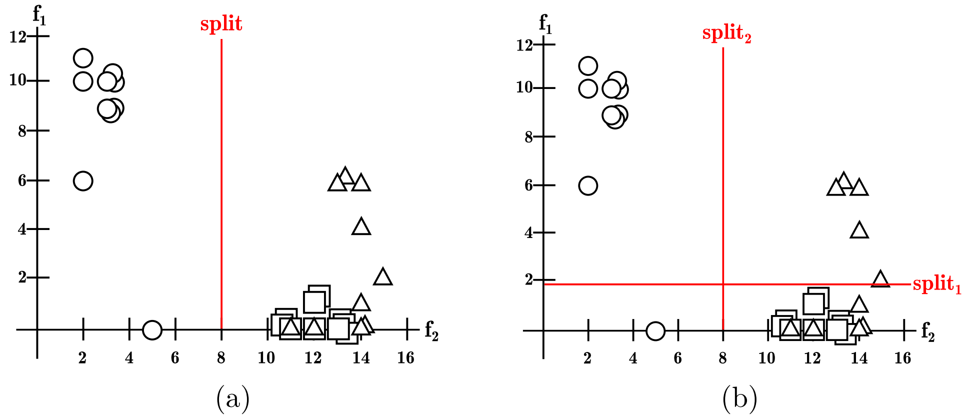


Figure 5.2: Determining at which node/feature to split early in the tree requires some testing when this is done by hand. The coordinate systems in (a) and (b) hold the combined values of feature one (number of special platforms) and two (number of attacks) from the test data in round 15. The shapes denote the corresponding strategies which played when these values were observed. The symbol ◯ denotes the combined values of strategy 1, △ values of strategy 2 and □ indicates values of strategy 3. The x-axes holds values of feature one, the y-axes those of feature two. When splitting first at feature one (a), a pure subset of values of strategy 1 can be filtered. The second subset does not contain values of strategy 1. Splitting first at feature two (b), more than one split is necessary in order to receive a similar result. **Note:** Jitter added to improve graphical representation of discrete values.

feature (number of attacks) were mapped to a two dimensional coordinate system. If splitting the values by feature one first, the resulting subset on the left side contains values of strategy 1 only. Furthermore, the second resulting subset contains only values of strategy 2 and 3. If deciding to split the values first by feature two, further splits are necessary to receive the same "pure" result.

For the implementation of the CART classifier a tree data structure was built for holding the best split at each node. As this was done in advance for each round, the structures were stored in a Java *List*. Besides defining the appropriate structure for traversing the tree quickly, there were no major difficulties at the implementation. For the best split the formula of calculating the mean of a subset (see equation 4.1) and the one for the SSE (see equation 4.2)were implemented. Hence, determining the best split programmatically was not difficult either.

When having a look at the average performance of the classifier during the games, there is just little misclassification. Table 5.3 displays the average classification results during a game in numbers and percentage. Some misclassification can be reduced if the first five rounds are not considered in the results (see table 5.4).

Table 5.3: Average intermediate classification results of the CART classifier per game. The games against strategy 1 lasted about 38.8 rounds, those against strategy 2 28.4 rounds and playing against strategy 3 took about 16.0 rounds. The rows indicate the actual strategy and the columns represent the classified strategies. Values in the last column (X) indicate that the classifier was not able to identify any strategy. The bold values in the diagonal display the correct classifications.

Average intermediate results per game:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**38.2**	0.6	0.0	0.0
Strategy 2	2.0	**25.2**	1.2	0.0
Strategy 3	1.8	0.2	**13.8**	0.2

Average intermediate results per game in percentage:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**98.45**	1.55	0.00	0.00
Strategy 2	7.04	**88.73**	4.23	0.00
Strategy 3	11.25	1.25	**86.25**	1.25

Table 5.4: Average intermediate classification results of the CART classifier per game without the first five rounds. The rows indicate the actual strategy and the columns represent the classified strategies. Values in the last column (X) indicate that the classifier was not able to identify any strategy. The bold values in the diagonal display the correct classifications.

Average intermediate results per game:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**33.2**	0.6	0.0	0.0
Strategy 2	1.0	**22.2**	0.4	0.0
Strategy 3	0.0	0.2	**11.0**	0.2

Average intermediate results per game in percentage:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**98.22**	1.78	0.00	0.00
Strategy 2	4.24	**94.07**	1.69	0.00
Strategy 3	0.00	1.75	**96.49**	1.75

Table 5.5: The values here represent the average number of splits performed by the CART classifier during the games against the three different strategies. The values for one and two splits indicate that the decision for a specific strategy is made early in the tree and pruning was possible. As the tree was built by hand, traversing the whole decision tree caused three splits at a maximum.

Average results per game:

	Strategy 1	*Strategy 2*	*Strategy 3*
One split:	18.8	0.2	0.4
Two splits:	10.0	13.4	0.4
Three splits:	10.0	14.8	15.2
Total number of splits:	68.8	71.4	46.8
Avg number of rounds:	38.8	28.4	16.0
Avg splits per round:	1.8	2.5	2.9

As visualized in the results, in only one round out of all rounds in the 15 games the classifier was not able to identify any strategy. In total, the classifier succeeds to correctly classify the strategy of its enemy in almost 95% of the cases after the first few rounds. This result implies, that the arrangement of the features in the tree is properly done. If the results would be significantly worse, a rearrangement of the nodes should be considered. Furthermore, mistakes during the feature extraction process can also cause poor results if the features are not expressive enough.

Also interesting are the results of the traversed tree visualized in table 5.5. On average, 1.8 splits are performed per round when playing against strategy 1. In almost 50% of the rounds (18.8 out of 38.8), the decision tree could be pruned after the first split. In another 24% of the cases, pruning was possible after the second split. When playing against strategy 2 or 3, there was less pruning than for the first one. The decision tree for this setup is with its three levels a rather small tree. Hence, there is not much difference in time performances with or without pruning. However, the fact that there is pruning shows that the right order of the features can increase the performance of the classifier significantly, even for a simple tree.

All in all, the CART classifier did perform very well. The identification of the opponents strategy for each round was mainly correct and the decision tree kept minimal. Hence, the OM AI did win any game played against the static AIs when using the CART classifier for classification.

5.3 kNN Classifier

The kNN classifier was, out of the three tested classifiers, the easiest one to understand and implement. The idea here is to put all values in an n-dimensional space, where n denotes the number of features used for the classification. When there are new values observed, the ones in the space that are closest to the new values are considered to be most likely of the same class. Hence, the only challenge for this classifier is to determine the distance between the input values and those of the labeled test data. The *Euclidean distance* is a reasonable measurement for distances in space and therefore matches perfectly for this task. For quicker calculations, the squared distance was determined.

Some features may have more impact on the result than others, as the ranges of the features are not equal. Therefore, it is not advisable to use the pure values of the features. For this setup, all values were transformed by standardizing them before the distances are calculated.

There were no difficulties in implementing the formulas of this classifier. For the data structure simple Java *Lists* were used. In this implementation the lists were sorted in order to determine the k-nearest-neighbors. Another possibility would be to use a *PriorityQueue* where the neighbors are automatically sorted ascending according to the implemented *Comparator* or the natural ordering of the used data type of the distance.

When testing the kNN classifier the OM AI succeeded to defeat its enemy in 14 out of 15 games. The OM AI lost one game against a static AI playing strategy 1 due to misclassification early in the game. As mentioned in chapter 3, the goal of the first strategy is to rush the enemy's base in the beginning of the game. Hence, if the classifier fails to identify this strategy during the first few rounds, a loss is inevitable.

The results in table 5.6 display the average performance of the classifier during the game. Table 5.7 shows the average results of the same games without the first five rounds. Comparing the values, it is clear that some misclassification results from the similarity of the data in the beginning of the game. When playing strategy 2 as a counter to strategy 1, one game lasted longer than all the test games. Hence, in a few rounds the classifier was not able to detect any neighbors and as a consequence it was not able to identify any strategy.

Misclassification when using kNN mainly occurs when the value of **k** is not appropriately chosen. Determining **k** for a setup where just two categories are possible is quite simple. Ties can be easily avoided by choosing an uneven value for **k**. In this game there are three categories/strategies present in the test data. Hence, not every uneven value **k** is the appropriate choice. For example, multiples of the number of the categories may also result in ties. Hence, defining the appropriate value for **k** is not a trivial task and requires at least some testing.

Table 5.6: Average intermediate classification results of the kNN classifier per game. The games against strategy 1 lasted about 35.0 rounds, those against strategy 2 28.8 rounds and playing against strategy 3 took about 16.2 rounds. The rows indicate the actual strategy and the columns represent the classified strategies. Values in the last column (X) indicate that the classifier was not able to identify any strategy. The bold values in the diagonal display the correct classifications.

Average intermediate results per game:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**30.8**	1.8	0.4	2.0
Strategy 2	0.4	**28.0**	0.4	0.0
Strategy 3	2.0	0.8	**13.4**	0.0

Average intermediate results per game in percentage:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**88.00**	5.14	1.14	5.71
Strategy 2	1.39	**97.22**	1.39	0.00
Strategy 3	12.35	4.94	**82.72**	0.00

Table 5.7: Average intermediate classification results of the kNN classifier per game without the first five rounds. The rows indicate the actual strategy and the columns represent the classified strategies. Values in the last column (X) indicate that the classifier was not able to identify any strategy. The bold values in the diagonal display the correct classifications.

Average intermediate results per game:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**27.2**	0.8	0.0	2.0
Strategy 2	0.0	**23.8**	0.0	0.0
Strategy 3	0.8	0.2	**10.2**	0.0

Average intermediate results per game in percentage:

	Strategy 1	*Strategy 2*	*Strategy 3*	X
Strategy 1	**90.67**	2.67	0.00	6.66
Strategy 2	0.00	**100.00**	0.00	0.00
Strategy 3	7.14	1.79	**91.07**	0.00

5.4 Combined Results

The duration of the games were quite different for the particular strategies. On average, a game lasted about 38 rounds when playing against strategy 1. When facing strategy 2, each player made about 28 turns. Games against the third strategy were won after about 16 rounds. These values are easily explained. When correctly identifying strategy 1, the OM AI starts to play the counter, which is strategy 2. As this is the defensive one, the OM AI first captures platforms to protect its base platform. In the next phase, the OM AI needs to send 16 units to the special platform Castle in order to win the game[1]. Hence, it is quite obvious that games against an enemy playing the first strategy can take a while.

When correctly identifying that the opponent is playing strategy 2, the OM AI can send a unit to the special platform C. After ten rounds the OM AI receives a speed boost which allows the AI to move its units more steps per turn. Additionally, all units of the OM AI are upgraded. With this advantage the OM AI is able to attack the enemy's base and win the game. As the upgrade lasts ten rounds and usually two or more upgrades are necessary for breaking the defensive wall of the enemy playing strategy 2, the average of 28 rounds is quite reasonable.

Finally, when facing strategy 3, the game needs to be won quickly. The OM AI therefore decides to play strategy 1 and "rush" the enemy's base. If the classifier fails to identify that the opponent is playing strategy 3, or if it identifies it too late in the game, the opponent can easily destroy the OM AI's base platform with the upgraded units.

In fact, the OM AI always plays the counter to its opponent's strategy if the classification was correct. Hence, it is no surprise that these games sometimes last longer than the test games. As a result, the classifier is not able to identify any strategy, or there is misclassification for a different one. The first case can be solved by relying on the last classification result. For example, when there is no test data available for a round, the OM AI sticks to the previously classified strategy and its currently executed counter strategy. Another approach would be to choose the one which was identified the most during the past rounds.

Solving the error of misclassification for a different strategy due to missing test data is a bit more difficult. One solution might be to check if the last classified strategy is available in the data set of the current round. If that is not the case, the OM AI could proceed similarly to the solutions proposed for the case when there is no data available for a round.

Misclassification in the first few rounds is reasonable but can have bad consequences, in the worst case: losing the game. This case occurred for the

[1]There are 15 neutral units on the platform Castle, which need to be defeated before the platform can be taken.

kNN classifier when playing against strategy 1 and the enemy rushes the base as the strategy was not classified correctly. A possible way to avoid a loss due to misclassification at the beginning of the game is to protect the base platform first. Hence, playing strategy 2 in the first few rounds in order to be ready for early attacks. Classification can take place in the background, but should not have an impact on the played strategy. The test results for all classifiers show that misclassification at the beginning of the game happens mainly during the first five rounds. Therefore, it is reasonable to use the classification results after this five rounds.

Comparing the performance of the three classifiers, all of them were able to fulfill the given task (not mentioning the one exception which could have been avoided by implementing the proposed solution above). The CART classifier was able to identify the opponents strategy more accurately than the other two. The kNN classifier was the easiest to understand and implement. The performance of the Naive Bayes' classifier was not as accurate as the other classifiers. However, the classifier would perform better, if the probabilities are given and need not be determined from a small data set.

Chapter 6

Conclusion and Outlook

6.1 Lessons Learned

In this research, three different pattern classification methods were described and tested in detail. Furthermore, ideas how to implement those methods are provided. It is shown, that the methods can be used without replay data or game logs from expert play.

A small two-player, turn-based strategy game was developed in order to test the different opponent modeling methods. The goal of the game is to capture special platforms in order to gain an advantage over the enemy player. There are paths between the platforms. The players can move their units from one platform to an other along the given paths. Each player is able to move units five steps per turn. The game can be won if the base platform of a player is destroyed by the opponent. Another possibility is to use all resources (units and moves) to capture a special platform called Castle. When one player succeeds in doing so, the game is over and the player with units on this special platform wins the game.

Three different strategies were identified in the game. Each strategy is strong against one, but weak against the other. Hence, there is a counter to each strategy. Static AIs were implemented to use the strategies (one for each strategy). The goal of strategy 1 is to destroy the enemies base as quickly as possible. When playing strategy 2, the AI is acting defensively and tries to win the game by capturing the special platform Castle. This strategy is strong against the first one, as the AI gathers units around the base in order to protect it (the AI builds a defensive wall). The third strategy sends one unit to another special platform which causes a technical upgrade for the AI and its units after a few rounds. Strategy 3 can beat a player playing the second strategy as the upgraded units can break the defensive wall. However, when facing strategy 1, the AI playing the third one will lose the game as the technical upgrade takes some time and makes the AI vulnerable to attacks.

For testing the opponent modeling methods, another AI was created. This AI is trained with data from test games of the static AIs playing against each other. In total, there was data from 30 test games available for the training of the opponent modeling AI. The tested classification methods were: Naive Bayes', the decision tree implementation CART, and the k-nearest-neighbors algorithm. These methods were tested in 45 games, 15 games per classifier. The classifiers played five games against each strategy.

The Naive Bayes' uses the test data in order to calculate the posterior probability of the opponent to play a certain strategy given the observations in the game. When using the Naive Bayes' for classification, the opponent modeling AI did win any game against the static AIs. During the games, the classifier performed best when facing strategy 1. However, in more than 20% of all rounds it was not able to identify strategy 3. Hence, the Naive Bayes' classifier was sometimes not able to produce accurate results.

The second classifier which was tested, was the CART classifier. It uses a decision tree in order to identify the opponent's strategy. The most difficult part here was to build the decision tree. First, the order of the nodes in the tree was defined. For this small setup, the order was determined by hand. Hence, some testing was necessary before the order was fixed. Second, the best split at each node needed to be determined. This was done by calculating the SSE for each possible split and for all nodes. When testing the performance of the CART classifier, the opponent modeling AI did win any game against the static AIs. The accuracy of the classifier was above 94% when playing against strategy 2 and even higher when facing strategy 1 or 3.

The kNN classifier was the last one which was tested in this research. It was the easiest to understand and implement. For this classifier, the data was put into an n-dimensional space and the Euclidean distance was calculated between the training data and the observed one. After identifying the k-nearest-neighbors, the strategy, which appears most often in the selected subset, is the one the opponent modeling AI is most likely facing. When using the kNN classifier the opponent modeling AI did win 14 out of 15 games. Due to misclassification at the beginning of the game, the AI lost one game against a static AI playing strategy 1.

The results of the testing show, that the opponent modeling AI can be improved by fixing its strategy at the beginning of the game to the defensive strategy. The advantage in doing so is clear. During the first few rounds, the test data does not differ much between the three strategies. Hence, the classifiers are prone to misclassification in this phase. When fixing the strategy to protect the base of the AI at the beginning, and starting to classify the opponent afterwards prevents the AI from losing the game due to misclassification in the early game.

The approach of fixing the strategy in the first few rounds is also advisable when having a game with imperfect information. In this case, the AI

can start playing defensively and change its strategy after gathering enough information about its opponent. Another possibility would be to play the Nash equilibrium strategy first, which means, choosing randomly which of the three strategies to play.

The most important lesson learned is that including opponent modeling in a game is not easy and requires expert knowledge about the game in order to select appropriate features for the classification process. Opponent modeling is used in order to improve the game AI and make it more challenging and more exciting to play against. An AI which is able to identify its opponent's strategy and react upon it should be fair and not cheating. Furthermore, a well-balanced game is necessary if opponent modeling should be included.

6.2 Outlook

With the help of the implemented game, further pattern classification methods can be tested and evaluated. For example, the classification with *hidden Markov models* (HMMs). As in a game, all actions in time t are dependent on moves the player performed in the state $t-1$, HMMs can be used for classification. Inferences can be made from previous states as they do have a direct influence on the current one. Certainly, the second player's actions also have an influence on the first player's moves. Hence, the difficult part here is to determine the conditional probabilities of a transition between the strategies. Implementations of the expectation maximization (EM) algorithm, like the *Baum-Welch* algorithm, are used for the learning of these probabilities from sample sequences. A detailed description of HMMs and the pseudocode of the *Baum-Welch* algorithm can be found in Duda et al. [9]. An implementation of the HMM approach can be found in [7]. The model was implemented for the RTS game StarCraft. In this research Dereszynski et al. used games from expert play in order to train the HMMs.

The available strategies in the implemented turn-based strategy game were identified by the designer. In large games it is sometimes necessary to automatically identify those strategies. However, game logs or replay data is needed for this identification. Etheredge et al. [10] use cluster analysis in order to determine the different player types in their own game. Clustering algorithms are used, when the given sample data for training is not labeled with the available classes or strategies in the game. This automated process is also called *unsupervised learning*. Applying automatic identification of the different strategies in a game can save much time and effort of collecting labeled data, or labeling data by hand. Furthermore, unsupervised learning can also be used for finding appropriate features for the classification. The greatest risk, when including clustering algorithms, is that the result may be not able to determine all classes, if the number of clusters is not known in

advance. The clustering may also be not accurate, when different classes are very similar. For example, for the presented game the features of strategy 2 and 3 do not differ much during the early and the mid game.

As stated in section 4.2, the decision tree for the classification with CART was built by hand. For games with more features this solution may be infeasible as it requires a lot of testing in order to find an appropriate order of the nodes. A solution for automating this process is to calculate the information gain for each node and select the one which leads to "pure" descendent nodes. A node is "pure" if the classification for one class is clear, meaning the subset of this node does not contain data of other classes and no further splits are needed. For the calculations of the information gain, a common technique is to determine the impurity of the nodes. The most common mathematical measures of impurity are *entropy* and *gini impurity*. When building decision trees automatically, the most important factor is to define when to stop splitting. This is because there is the danger of overfitting, if the tree is built until only pure leaf nodes are left. One approach of deciding when to stop splitting is *cross-validation*. By using this technique, the tree is trained with 90% of the training data and the rest is used for validation. Another possibility would be to set a certain threshold value for the information gain, or for the number of samples left in the resulting subset (for example 5% of the total training set). Instead of deciding when to stop splitting, the tree can be built completely, and afterwards be pruned. This technique is recommendable, if the decision trees are small, as the computational costs are not too high.

To sum everything up, the implemented game environment was used to test three different classification methods and can be used to test further ones. Additionally, the presented methods can be extended for efficient usage in bigger environments.

Appendix A

Content CD-ROM

A.1 PDF Files

Path: /

Mayrhofer_Melanie_2015.pdf Master Thesis

Path: /images

A.pdf	Vector graphic of platform of type A
B.pdf	Vector graphic of platform of type B
binning.pdf	Vector graphic displaying the binning mechanism for data preparation
C.pdf	Vector graphic of platform of type C
cartSplit1.pdf	Vector graphic displaying splitting data at one feature
cartSplit2.pdf	Vector graphic displaying splitting data at two features.
Castle.pdf	Vector graphic of platform of type Castle
CreateUnitWorkFlow.pdf	Vector graphic of the workflow when creating a new unit
decisionTree1.pdf	Vector graphic of a sample decision tree
decisionTree2.pdf	Vector graphic of splitting at a node of the decision tree
featureDistance.pdf	Vector graphic of the class *FeatureDistance*
flow.pdf	Vector graphic displaying the game flow
MessageClass.pdf	Vector graphic of the class *NextTurnMessage*
MessageClass2.pdf	Vector graphic of the class *FeatureObservationMessage*
mockup.pdf	Vector graphic of the game mockup

nearestNeighbors . . . . Vector graphic displaying the nearest neighbor approach

OMAI.pdf . . . . Vector graphic displaying the workflow of the opponent modeling AI

strategy_1.pdf . . . . . Graphic displaying a screenshot of the game when the AI is playing strategy 1

strategy_2.pdf . . . . . Graphic displaying a screenshot of the game when the AI is playing strategy 2

strategy_3.pdf . . . . . Graphic displaying a screenshot of the game when the AI is playing strategy 3

testingBayes.pdf . . . . Vector graphic of a test result while using the Naive Bayes' classifier

UnitEntity.pdf . . . . . Vector graphic of the class *UnitEntity*

A.2 Executable JAR Files and Dependencies

Path: /build

BAYES.jar . . . . Executable JAR file where the opponent modeling AI uses the Naive Bayes' classifier for the identification of the static opponents' strategies.

CART.jar . . . . Executable JAR file where the opponent modeling AI uses the CART classifier for the identification of the static opponents' strategies.

KNN.jar . . . . Executable JAR file where the opponent modeling AI uses the kNN classifier for the identification of the static opponents' strategies.

strategyStatistics.ser . . Serialized data for training the classifiers.

A.3 Test Results

Path: /statistics

LOG_BAYES.csv . . . . Test results of the Naive Bayes' classifier playing 15 games against static AIs

LOG_CART.csv . . . . Test results of the CART classifier playing 15 games against static AIs

LOG_KNN.csv . . . . . Test results of the kNN classifier playing 15 games against static AIs

References

Literature

[1] Tetske Avontuur. “Opponent Modelling in Wargus”. Bachelor Thesis. Tilburg University, Dec. 2010 (cit. on p. 8).

[2] Sander CJ Bakkes, Pieter HM Spronck, and H Jaap Van Den Herik. “Opponent modelling for case-based adaptive game AI”. In: *Entertainment Computing* 1.1 (2009), pp. 27–37 (cit. on pp. 1, 6).

[3] Nadia Boudewijn. “Opponent Modeling in Texas Hold’em”. Bachelor Thesis. Utrecht University, Jan. 2014 (cit. on p. 7).

[4] Michael Bowling et al. “Machine learning and games”. In: *Machine Learning* 63.3 (May 2006), pp. 211–215 (cit. on p. 3).

[5] Aaron Davidson. “Opponent Modeling in Poker: Learning and Acting in a Hostile and Uncertain Environment”. Master Thesis. University of Alberta, 2002 (cit. on p. 7).

[6] Aaron Davidson et al. “Improved Opponent Modeling in Poker”. In: *International Conference on Artificial Intelligence, ICAI’00*. 2000, pp. 1467–1473 (cit. on p. 7).

[7] Ethan Dereszynski et al. “Learning Probabilistic Behavior Models in Real-Time Strategy Games”. In: *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2011, pp. 20–25 (cit. on p. 49).

[8] H.H.L.M. Donkers and P.H.M. Spronck. “Preference-based player modeling”. In: *S. Rabin (Ed.), AI Game Programming Wisdom 3*. Charles River Media, 2006, pp. 647–659 (cit. on p. 8).

[9] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. Wiley, 2012 (cit. on pp. 6, 24, 26, 34, 49).

[10] Marlon Etheredge, Ricardo Lopes, and Rafael Bidarra. “A Generic Method for Classification of Player Behavior”. In: *Proceedings of the Second Workshop on Artificial Intelligence in the Game Design Process*. 2013, pp. 1–7 (cit. on p. 49).

[11] S. García, J. Luengo, and F. Herrera. *Data Preprocessing in Data Mining*. Springer International Publishing, 2014 (cit. on p. 24).

[12] H Jaap van den Herik, HHLM Donkers, and Pieter HM Spronck. “Opponent Modelling and Commercial Games”. In: *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG’05)*. 2005, pp. 15–25 (cit. on p. 6).

[13] Stephen Hladky and Vadim Bulitko. “An Evaluation of Models for Predicting Opponent Locations in First-Person Shooter Video Games”. In: *Proceedings of the IEEE 2008 Symposium on Computational Intelligence and Games (CIG’08)* (2008), pp. 39–46 (cit. on p. 9).

[14] Bret Hoehn et al. “Effective Short-Term Opponent Exploitation in Simplified Poker”. In: *AAAI*. Vol. 5. 2005, pp. 783–788 (cit. on p. 7).

[15] Agapito Ledezma et al. “OMBO: An opponent modeling approach”. In: *AI Communications* (2009), pp. 21–35 (cit. on p. 9).

[16] I. Millington and J. Funge. *Artificial Intelligence for Games*. Morgan Kaufmann. Taylor & Francis, 2009 (cit. on p. 17).

[17] S. Rabin. *Introduction to Game Development: Second Edition*. Game development series. Cengage Learning, 2010 (cit. on p. 14).

[18] MZ Rashad. “A Rough–Neuro Model for Classifying Opponent Behavior in Real Time Strategy Games”. In: *International Journal of Computer Science & Information Technology* 4.5 (2012), pp. 185–196 (cit. on p. 8).

[19] Matthijs Rohs. “Preference-based Player Modelling for Civilization IV”. Bachelor Thesis. Maastricht University, July 2007 (cit. on p. 8).

[20] Terence Conrad Schauenberg. “Opponent Modelling and Search in Poker”. Master Thesis. Alberta, Faculty of Graduate Studies and Research, 2006 (cit. on p. 7).

[21] J. Schell. *The Art of Game Design: A Book of Lenses, Second Edition*. Morgan Kaufmann. Taylor & Francis, 2014 (cit. on pp. 1, 2).

[22] Finnegan Southey et al. “Bayes’ Bluff: Opponent Modelling in Poker”. In: *Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI*. 2005, pp. 550–558 (cit. on p. 7).

[23] Pieter Spronck and Freek den Teuling. “Player Modeling in Civilization IV.” In: *AIIDE*. 2010, pp. 180–185 (cit. on p. 8).

[24] Jan A Stankiewicz and Maarten PD Schadd. “Opponent Modeling in Stratego”. In: *Proceedings of the 21st BeNeLux Conference on Artificial Intelligence (BNAIC’09)(eds. T. Calders, K. Tuyls, and M. Pechenizkiy)*. 2009, pp. 233–240 (cit. on p. 6).

[25] Gabriel Synnaeve and Pierre Bessiere. "A Bayesian Model for Opening Prediction in RTS Games with Application to StarCraft". In: *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE. 2011, pp. 281–288 (cit. on pp. 7, 8).

[26] Marcel Van Der Heijden, Sander Bakkes, and Pieter Spronck. "Dynamic Formations in Real-Time Strategy Games". In: *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*. IEEE. 2008, pp. 47–54 (cit. on p. 8).

[27] Ben George Weber and Michael Mateas. "A Data Mining Approach to Strategy Prediction". In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE. 2009, pp. 140–147 (cit. on pp. 7, 8).

[28] Jiajia Zhang. "Building Opponent Model in Imperfect Information Board games". In: *TELKOMNIKA Indonesian Journal of Electrical Engineering* 12.3 (2014), pp. 1975–1986 (cit. on p. 6).

Online sources

[29] *Balance (game design)*. URL: http://en.wikipedia.org/wiki/Balance_(game_design) (visited on 06/24/2015) (cit. on p. 20).

[30] *Euclidean distance*. URL: http://en.wikipedia.org/wiki/Euclidean_distance (visited on 06/24/2015) (cit. on p. 30).

[31] *Nash equilibrium*. URL: http://en.wikipedia.org/wiki/Nash_equilibrium (visited on 06/24/2015) (cit. on p. 2).

[32] Kardi Teknomo. *K Nearest Neighbors Tutorial*. URL: http://people.revoledu.com/kardi/tutorial/KNN/index.html (visited on 06/24/2015) (cit. on p. 29).

[33] *Weka: Data Mining Software in Java*. URL: http://www.cs.waikato.ac.nz/ml/weka/ (visited on 06/24/2015) (cit. on p. 7).