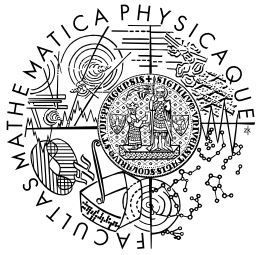

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Eliška Suchardová

Artificial Intelligence for a Castle Conquest Simulation Game

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Most of all, I would like to thank my supervisor for always being helpful, giving valuable advice and never losing his patience. A special thank you also goes to my family and friends, who were always there for me when I needed them.

Title: Artificial Intelligence for a Castle Conquest Simulation Game

Author: Eliška Suchardová

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Artificial intelligence presents a highly researched field with significant contribution to the progress of modern technologies. As there exist numerous distinct approaches to AI implementation, games can be conveniently utilized as environments for their testing. This thesis aims to create a simple strategy game with an AI framework designed for such testing. Furthermore, to illustrate its usage, several of those approaches, mostly evolutionary, are realized together with assessment of their performance.

Keywords: artificial intelligence, strategy games, evolutionary algorithms, coevolution, neuroevolution

Contents

Introduction

Artificial intelligence (AI) has been heavily studied for several decades and gained significant popularity over the years. Advancements in the field are crucial in development of modern technology and contribute to the improvement of quality of everyday life. With the rapid progress of computers, AI is being employed in more and more diverse settings such as medicine, transportation but also in games. There are many distinct approaches used in AI development and games provide a great environment for their testing.

Similarly to AI, computer games have been getting increasingly popular in recent decades. As mentioned, the game industry is amongst the many fields that try to utilize AI to its advantage. Certain genres heavily depend on usage of AI usually in the form of an opponent for the human player. Notably, strategy games are amongst those that would be drastically different without it. They often portray battles where enemy armies are controlled by AI.

With this in mind the main goals of this thesis are as follows:

- Implement a simple castle conquest strategy game together with a framework for AI implementation and consequent training.
- Utilize the framework to implement and train several AIs for the game using different, mostly evolutionary, approaches and evaluate their performance.

Additionally, a simple interface for a human player will be also implemented.

The thesis is divided into five separate chapters. The first one shortly introduces the genre of strategy games along with their early history. In the second chapter the game design of the castle conquest will be discussed together with all the elements present in the game. The third chapter describes the idea behind evolutionary algorithms and special evolutionary approaches such as coevolution and neuroevolution. Fourth chapter focuses on detailed explanation of the AI framework, its usage and presents all the implemented AIs. Finally, in the last chapter several experiments that were conducted to properly balance the game and settings of the AI players are presented along with performance assessment of the trained AIs.

1. Strategy games

The genre of strategy games is nothing new and has been around for quite some time. The oldest strategy games, such as *Senet*, can be dated all the way back to the times of ancient Egypt [11].

Naturally, with the the development of modern computers, many known board games were being made into their computerized versions [16]. In 1940, a variation of *Nim*, an ancient strategy game, was presented at the New York World's Fair and is considered to be amongst the first computer games ever created. Other board games such as checkers and chess followed. This marked the beginning of a long way to the strategy video games as we know them now.

Nowadays, military or combat strategy video games are amongst the most popular [16]. The player adopts the role of a commander, leading the troops into battle and controlling them on the battlefield to secure a victory. Other types worth mentioning might let the player control a whole civilization or take on the role of a mayor running a city.

In general, strategy games require skillful and analytical thinking as well as long term planning [19]. Each individual decision made by the player can influence the outcome of the game therefore its impact should be thoroughly weighted beforehand.

Ultimately, there are two main categories that strategy games can fall into [19]. As mentioned, many early strategy games originated from board games leading to the first successful commercial games to be turn based. The main idea behind this approach is to divide the gameplay into separate turns. This gives the players a chance to think over their strategy before making the next move enabling a more thought-out play. However, with the advancement of available technology, more and more complex games were being developed. Ultimately, this led to the creation of a completely new genre of real time strategy games. This genre brought a fully different approach to the gameplay. Suddenly, the player is forced to make fast decision, without much time to think them through making the game more fast paced. In contrast to turn based games, everything now happens in real time. The next section will shortly discuss the early days of both categories.

1.1 Turn Based Strategy Games

Turn based strategy games sparked the beginning of the whole strategy video games genre. They were amongst the first commercially successful games and remained popular to this day.

In 1981, a game called *Eastern Front (1941)* was released and went on to become one of the first widely popular strategy video games [16]. It was set in Europe during World War II and followed the German advancement into the Soviet Union. This tactical simulation game came with features such as changing weather and scrollable map with rivers and mountains.

Another extremely successful strategy game was *Civilization*, released in 1991 [16]. It's creator, Sid Meier, was considered to be a pioneer in this genre. His game is still to this day considered to be amongst the best strategy games ever

made. To no one's surprise, it's popularity led to multiple spin-off titles. The combination of nation development and battle planning is what makes this game interesting. The player takes control of a whole nation such as the Babylonians, Romans or Egyptians trying to ensure their prosperity and rise of their empire. The ultimate goal is to turn them into a modern civilization.

1.2 Real Time Strategy Games

During the late 1980s and early 1990s, the first RTS games came to existence [16]. *Dune II*, released in 1992, was amongst them. This title was based on the sci-fi novel Dune, bringing the player to the desert planet called Arrakis. Several of the 1990s games became so popular that multiple sequels and spins-offs of the titles were made. Notably, the 1997's *Age of Empires* ended up starting a 3 part series with multiple other spin-offs. Similarly, with the first title released in 2000, the Total War series consists of more than ten games of various themes and settings from feudal Japan to ancient Rome.

As mentioned, *Age of Empires* went on to become a huge hit. All the three main titles are still being sold to this day in their definitive edition versions with updated graphics. In the game, the player must pick one of the available civilizations such as ancient Rome, ancient Egypt and many more. Each one comes with own strong assets and weaknesses [2]. Throughout the game, resources need to be collected in order to be able to develop the civilization and advance their technology. The progress is measured in ages, where each age brings new knowledge and possibilities for improvement. Furthermore, the player has to build a strong army in order to survive possible attacks from enemies or to have the means to defeat the others. Multiple modes are offered in the game, one being a historically based solo campaign, other, an online multiplayer for those who are tired of playing against the AI.

2. Game Description

One of the goals of this thesis was to implement a simple asymmetrical RTS game simulating a castle conquest that would serve as an environment for artificial intelligence testing.

This castle conquest simulation game is a simple strategy game combining approaches used in tower defense games and RTS games like the Total War series [20]. The player takes on the role of a skilled army general picking one of the two sides - attacker or defender. As mentioned, this game is asymmetrical, the defender side can stay inside the castle while attacker is out in the open without much protection. Each side has its own advantages and disadvantages, however the goal is common for both, to eliminate the enemy. In this chapter we are going to discuss various aspects of the design and look at the different elements used in this game.

2.1 Map

Similarly to a chess board, the map, on which the battle takes place, consist of separate fields. Ultimately, it forms a $M \times N$ matrix. On each field, there can only be one object, such as a building or a troop (group of units of the same type). Movement is allowed solely from one field to another that is adjacent and unoccupied. However, diagonal movement is not allowed. Fighting, on the other hand, is not restricted in terms of direction.

2.2 Unit types

Both attacker and defender have the same unit types available for purchase. They are sold only in preset amounts, represented by the bundle count parameter. Each type has fixed price and own unique characteristics and skills, which are defined by these 6 parameters:

- Health
- Speed
- Damage
- Reload rate - Specifies how long said unit type takes between attacks.
- Range
- Bundle count - Represents the count in which they are sold. The group of units of certain type is throughout the text referred to as troop.

Table 2.1 shows values of all mentioned parameters for all implemented unit types.

Additionally, to resemble the real world with more accuracy and make the fighting styles of each unit more authentic, each type has its own way of dealing damage. Also damage modifiers shown in table 2.2 are applied based on the unit's opponent during a fight. For example, cavalry attacking a troop of archers

	Health	Speed	Damage	Reload	Range	Bundle	Price
Swordsmen	5500	0.02	20	0	1	10	5500
Cavalry	11000	0.04	20	0	1	5	8000
Archers	5000	0.02	10	2	5	10	5000
Catapult	80000	0.0054	300	10	4	1	15000

Table 2.1: Parameters of all unit types.

	Swordsmen	Cavalry	Archers	Catapult	Other
Swordsmen	0.9	0.7	0.9	0.95	0.8
Cavalry	0.8	0.9	0.95	0.8	0.9
Archers ranged	0.85	0.8	0.95	0.5	0.8
Archers melee	0.6	0.5	0.95	0.5	0.8
Catapult	0.7	0.7	0.9	0.7	1

Table 2.2: Damage modifiers where the types in the first column represent the attackers

has the damage modifier set to 0.95, which means the archers will only receive 95% of the total damage. In other words, the higher the number, the higher the actual dealt damage. Furthermore, the archers' modifier is not only based on the opponent, but also on the range of the attack.

Even though graphically, the whole troop is represented by a single character as shown in figures 2.1 and 2.2, internally the damage goes to the individual units. With this in mind, the damage of the whole troop is a sum of damages of the units still alive in the troop. This means that the troop gets weaker as its units are killed.

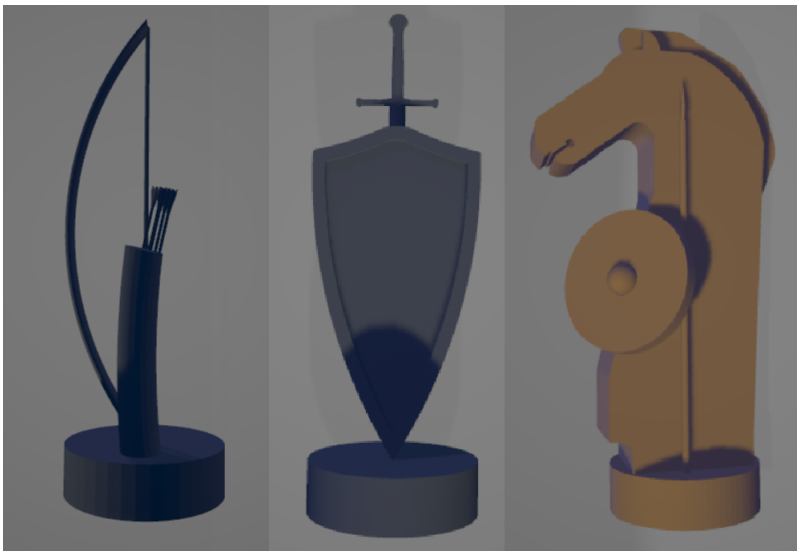

Figure 2.1: From left archers, swordsmen and cavalry

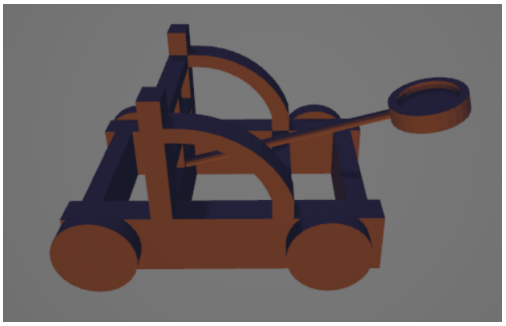

Figure 2.2: The catapult

2.2.1 Swordsmen

Swordsmen represent a melee unit with a decent power to price ratio. Thanks to the 0 reload time, this unit can chain attacks without any breaks. The damage is given to a single unit from the front line of the enemy troop, internally meaning the first in the list representation.

2.2.2 Cavalry

This unit is also a melee one but significantly faster than the swordsmen. In fact, the cavalry is the fastest unit in the game. On attack, the damage is dealt to a couple of enemy of units standing in a line next to each other.

2.2.3 Archers

This type is one of the two ranged units in the game. Even though archers have smaller damage than the previous two types, the ability to shoot at troops far away or even behind buildings comes in very handy. One random unit in the enemy troop receives the damage to mimic the behaviour of a single arrow.

2.2.4 Catapult

Last unit type is the catapult. It deals immense damage but the reload time is high and movement speed low. However, the launched projectile usually hits multiple enemies, therefore random chunk of the enemy troop receives the full damage making the casualties significant.

2.3 Buildings

On top of the previously mentioned units, the defender also owns buildings. However, buildings cannot be bought and come as a part of the selected map. There are passive buildings, such as walls and gates, that simply protect troops behind them. Specifically gates act as always open for the defender and always closed for the attacker, without the need to manage their state. Then there are towers, which can be actively used to attack the enemy in range. All the buildings have set health and can be destroyed by the attacker. Once the health decreases to

zero, the building disappears and the field becomes unoccupied making it passable. On top of the health parameter, the tower shares some characteristics with the units from previous section as shown in table 2.3.

	Health	Damage	Reload	Range
Basic tower	5500	10	4	5
Wall	11000	-	-	-
Gate	5000	-	-	-

Table 2.3: Parameters of all building types.

2.4 Gameplay

At the beginning, the map is revealed and each player is given some money to start with. Purchases can be made right away and bought troops are spawned onto one of the player's spawn points. Once in the game, they can be immediately send to fight. Depending on the side of the player, set amount of money is earned every once in a while.

The defender has the advantage of being able to hide inside the castle. The enemy troops cannot get in unless they destroy some part of the walls and create a way in. To counter balance this advantage, the attacker starts with more money and has higher earnings throughout the game. To be precise, the attacker starts with 16000 and earns 1600, in contrast, the defender starts with 12000 and earns 1200.

Once one of the players looses all of their troops, the player looses and the game ends.

3. Evolutionary algorithms

In this chapter, the discussion will be centered around the topic of evolutionary algorithms as one of the main techniques studied in the field of evolutionary computation.

Evolutionary algorithms take inspiration in Darwinian evolution and are considered to be amongst the main methods used in nature-inspired computing [6]. The approach utilizes the processes of evolution for optimization problems. Essentially, it is based on trial and error principle, where the trials represent possible solutions and the error deviation from the wanted result. Based on the error, suitable candidates can be selected for the further trials. Those should contain some modifications in order to achieve a decrease of the error. Evolutionary algorithms are mostly applied to optimization problems or problems, without a known algorithm yielding their solution [8]. The results are not guaranteed to be optimal, however, relatively good solutions are usually found in tolerable time.

3.1 Algorithm structure

Now, lets look at the structure of evolutionary algorithms. Their course is divided into separate rounds, commonly called generation [8]. Each generation represents one iteration of the algorithm. They operate on a set, called the search space, where points are evaluated by a certain function and assigned a value based on the result. The function is called the fitness function and plays a considerable role in the whole process. A multiset of points from the search space is selected for the algorithms to work on. It is referred to as population and each point in the population is called an individual.

There are countless possible methods how to represent individuals [1]. Usually, they are specific to the examined problem. Namely, binary or string representations are examples of possible approaches.

The general idea is, the search space must be inspected in order to find an adequate solution for the given problem. As the evolution progresses, better regions of the search space are visited. Figure 3.1 shows detailed visualization of a generic evolutionary algorithm structure. As explained in [8] and [9] the algorithms usually follow these 3 main steps:

1. **Initialization** - In the first step of the algorithm a random population is created. However, in some cases it might make sense to use a specific starting population.

2. **Iteration** - This step contains the main evolution based logic. Depending on the termination criteria, it usually ends up being run multiple times.

   (a) **Selection for reproduction** - Individuals for further use must be selected from the population. They are called parents. The process is usually based on their fitness values, where the higher the fitness the higher the chance of being selected. There is no rule prohibiting one individual to be chosen multiple times.

(b) **Variation** - Variation operators take the parent population and randomly apply changes to some individuals. There are two types that can be used - crossover discussed in subsection 3.1.2 and mutation in subsection 3.1.3. Often, not only one, but both are applied.

(c) **Selection for replacement** - Once the offspring population, created in the previous step, is obtained, new population for the next generation must be created. It can combine both individuals from the original population and the offspring. However, as the size of the population usually does not change throughout the algorithm, only certain amount can be selected. Therefore, some individuals will be chosen to be discarded. This is called selection for replacement. Obviously, this approach can be easily redefined as selection for survival, where chosen individuals instead of being discarded become part of the new population.

3. **Termination** - At the end, a termination criteria check is concluded. If met, the algorithms outputs a possible solution and ends. Otherwise, iteration step is run again.

There is one more important part to be discussed, the evaluation. The fitness function is applied to the population to obtain the fitness values for all individuals [8]. As mentioned, it is usually required for selection. It is omitted from the points above under the assumption that the values can be easily computed or are already available.

With this in mind, no strict rules must be followed when designing evolutionary algorithms. They can be created with various slightly different approaches. To quote [8]: "there is hardly any limit to creativity when designing evolutionary algorithms".

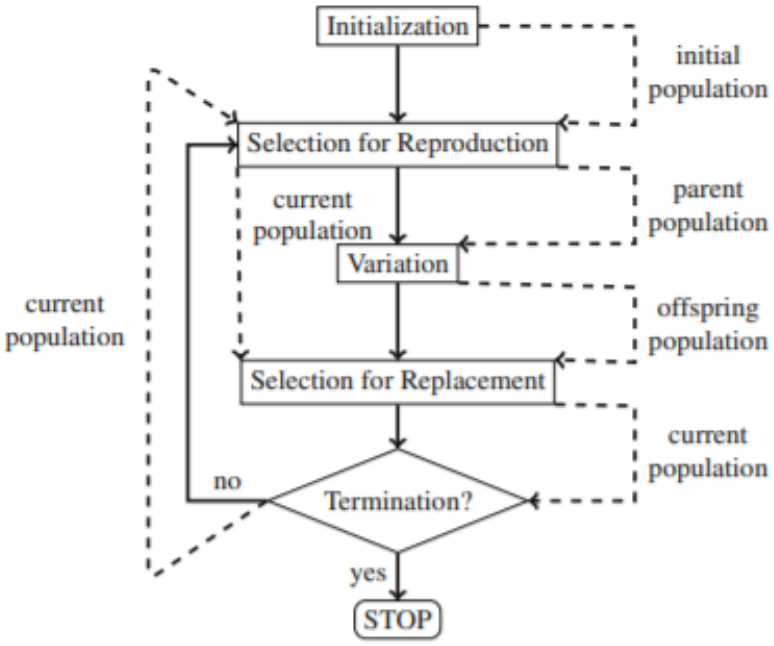

Figure 3.1: Generic evolutionary algorithm outline [8]

3.1.1 Selection

Selection is a crucial part of evolutionary algorithms. It manifests the survival of the fittest principle present in evolution and drives the progresses in the algorithm [8].

In order to ensure higher chance of survival for better individuals, the most commonly used selections in some form utilize the fitness value [8]. Another key point to remember when implementing selection is to specify, whether it operates with or without replacement. In other words, whether individuals can be selected multiple times or not. To demonstrate specific approaches, two common selection methods are discussed.

Roulette wheel

The first example is the roulette wheel selection. As its name suggests, it essentially mirrors the action of consecutively spinning a roulette wheel. Each section on the imaginary wheel represents one particular individual and its size is based on the individual's fitness as can be seen in figure 3.2. This approach is clearly fitness proportional. Higher fitness ensures a larger section on the wheel, therefore the probability of being picked increases proportionally to fitness.

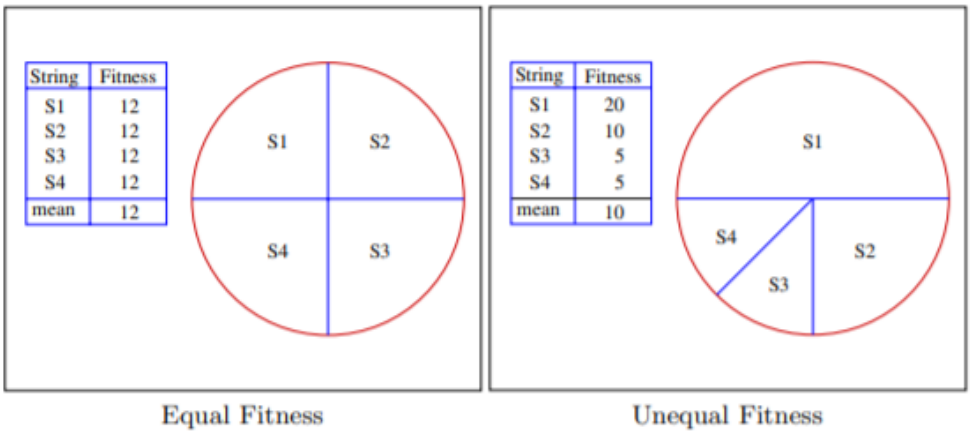

Figure 3.2: Visualization of the roulette wheel selection [5].

Tournament

The tournament selection is another commonly used method of selection [5]. Similarly to the previously mentioned roulette wheel selection, it is also fitness based. However, at first a certain number of individuals from the population is chosen randomly. Those then compete against each other in the supposed tournament. Individual with the highest fitness is declared the winner and is placed in the parent population. This process in repeated until the whole population is formed.

3.1.2 Crossover

As described earlier, the variation operators apply certain changes to the parent population. One of them, called crossover, is a binary operator that resembles

sexual reproduction [10]. At the beginning, there are two individuals, the parents, from which a specified number of offspring are then created. All the children inherit some features from their parents but none will be totally identical. Depending on the individuals' encoding, the approach to crossover implementation might differ. One-point, two-point, multi-point and uniform crossovers shown in figure 3.1.2 are amongst the most popular methods.

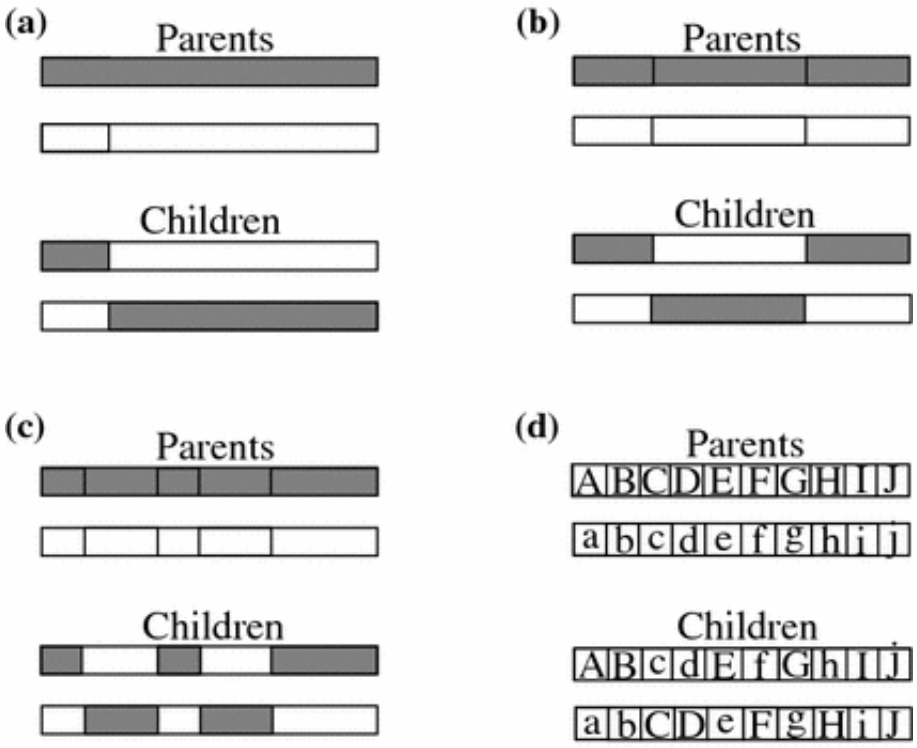

Figure 3.3: Crossover techniques: a. One point b. Two point c. Multi-point and d. Uniform [10]

3.1.3 Mutation

Mutation is the second of the two mentioned variation operators. Unlike crossover, this type of operator is asexual. That means only one individual is needed as input [10]. Mutation generates one offspring by applying, usually small, changes to the individual [8]. Again, the process itself is closely tied to the representation of the given individual, therefore no general approach exists.

To demonstrate, one of the common methods used for binary encoded individuals is the bit mutation [8]. Essentially, each bit in the individual is negated with certain probability. This process creates a new offspring, which may not even be different based on the probability setting. To ensure small changes, regularly used technique is to only change one bit on average.

3.2 Coevolution

Specific method, called Coevolution, can sometimes discover better solutions quicker than normal evolutionary algorithms. It allows the evolutionary algorithm

to utilize the divide and conquer approach [10]. Each problem is represented by a separate population, which is then gradually evolved. Essentially, the main idea is to apply mutual evolution of multiple species through their interactions. Those can be of different nature resulting in 3 different types of coevolution: competitive, cooperative and cooperative-competitive.

3.2.1 Competitive Coevolution

First, as [10] specifies, the competitive coevolution simulates a 'predator–prey' or 'host–parasite' relationship of given species. In other words, one species might be trying to harm the other, that, on the contrary, is trying to survive [14]. Thanks to this competitive environment, they constantly aim to outperform each other, which leads to their continuous evolution.

Example of a successful utilization of this approach is the coevolution of sorting networks and data sets [15]. It uses two populations where one represents the sorting network and the other an unsorted data set designed to assess the networks sorting performance. With the competitive-coevolution approach the algorithm manages to look for the smallest functioning network but also produces more and more challenging versions of data sets.

3.2.2 Cooperative Coevolution

Another type is the cooperative coevolution, also called symbiotic [10, 14]. As the name suggests, this time the species attempt to work together. Each one represents a separate part of the problem and aims to obtain the best solution. Those parts are then merged together to create the full solution. In the previous example, success of one species inevitably implied loss for the other, however, in cooperative approach the success of one species means success for all.

An example of this would be something similar to the approach used in the CovevolutionAI described in section 4.2.4. For simplification, an army consisting of three different unit types would be represented by three separate populations. Each one would correspond to a certain type of unit. Those must be then combined to achieve proper behaviour of the whole army. Even though each population is evolved separately the final performance and fitness value depend on the army as a whole.

3.2.3 Cooperative-Competitive

Finally, the cooperative-competitive coevolution combines both previously mentioned approaches [10]. However, they are employed in different sections of the algorithm. Competition might be used to evolve two populations which internally consist of multiple species using cooperative coevolution. This method was found to be fairly successful in dynamic multi-objective optimization.

3.3 Neuroevolution

Like many other techniques in evolutionary computation, neuroevolution is also inspired by evolution in nature, specifically the evolution of brains [17]. In the

1980's when both evolutionary algorithms and artificial neural networks, a form of artificial brain, were already heavily researched, an idea to combine those two approaches appeared. The research of applying evolutionary algorithms to train neural networks began. As evolutionary algorithms were already described, this section will focus on the neural networks themselves and one of the well-known neuroevolution algorithms called NEAT.

3.3.1 Neural Networks

Neural networks can be divided into two separate categories: biological and artificial [3, 4, 21]. The idea behind the creation of artificial neural networks was to attempt to imitate the process of problem-solving in the biological brain. Similarly to the human brain, the artificial neural networks also consist of linked neurons. For this reason, the following subsection describes the biological neuron and its function.

Biological Neuron

Neuron is the main building component of the biological neural system [4]. The four important parts of a neuron, shown in figure 3.4, are:

- **Dendrites** - The neurons in the network receive communication from others through their dendrites. The receptors on the dendrites detect a chemical called the neurotransmitter, which is then transformed to electrical impulses in the neuron.
- **Cell Body** - The cell body evaluates the signal received in dendrites and based on the results decides what to do.
- **Axon** - If the signal intensity meets a certain threshold the information is then carried via the axon to the dendrites. In other words, the neuron fires.
- **Axon Terminals** - Conversely to the dendrites, the axon terminals send output, the neurotransmitter, to other neurons in the network via connection structures called synapses. Those can be of various effectivity.

Artificial Neuron

In the artificial neuron, the axons and dendrites are represented by the connection between the nodes forming the network [3]. The synapses are realized by the weights, which represent their strength, and the threshold is also set to manage the activity. The learning process of the network is accomplished by balancing the weights.

Once the neuron receives data from its inputs, corresponding to the biological dendrites, it multiplies each signal by the weight assigned to the connection [4, 21]. The computed input signals are then added together to produce a net input. It is then transformed by the activation function to form the output of the neuron. Simple example is the unit step activation function. It takes the net input and subtracts the set threshold value. If the result is not negative, one is received as the output and the neuron fires. Otherwise zero is set as the output.

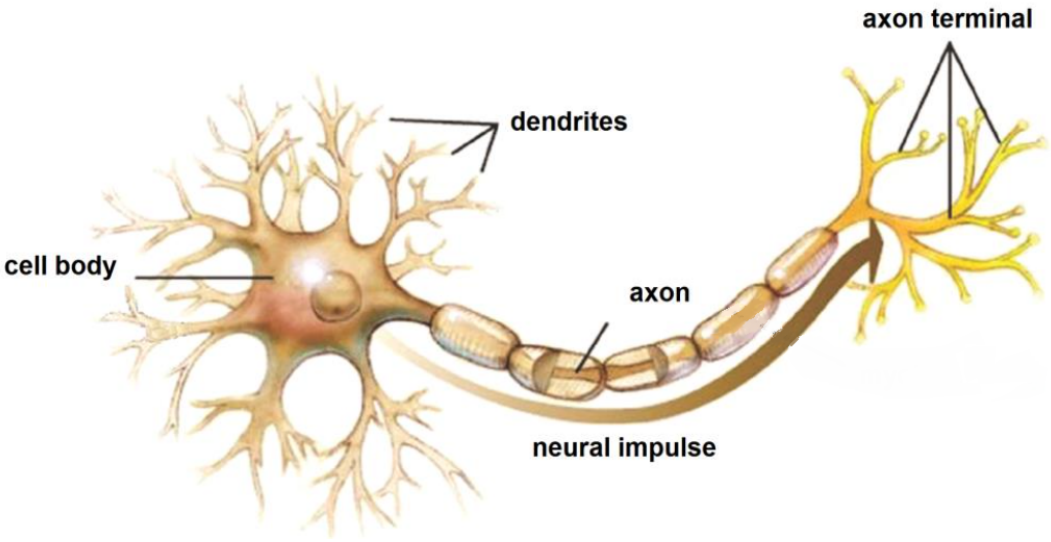

Figure 3.4: Biological neuron structure [3]

Neural Network

As already mentioned, the network is composed of a certain number of artificial neurons. Usually, they are separated into these layers: the input layer, one or more hidden layers, which convert the input to the output, and the output layer shown in figure 3.5 [4]. The connections between the nodes define the networks topology.

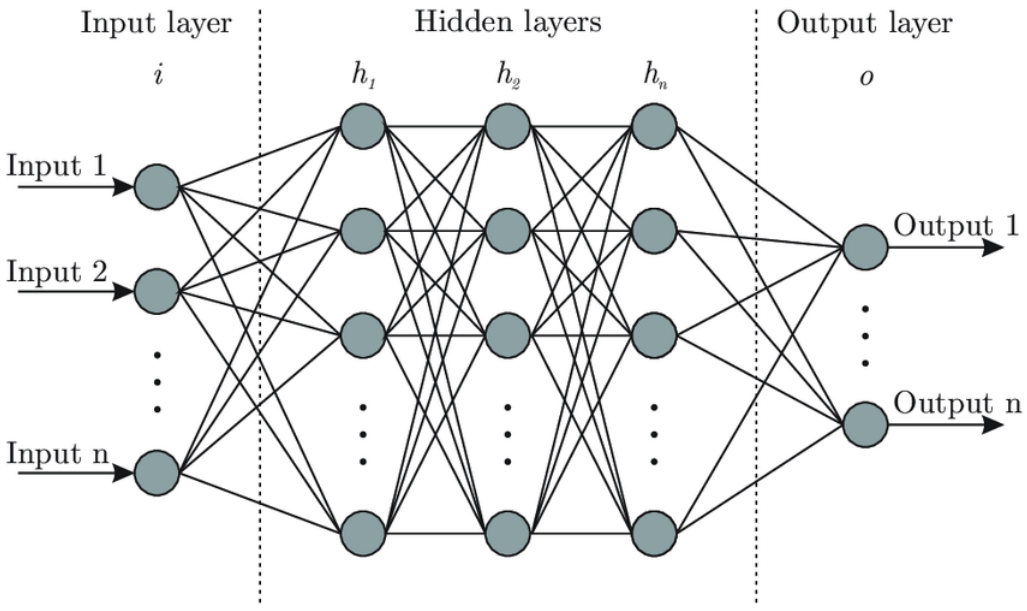

Figure 3.5: Typical neural network structure [12]

3.3.2 NEAT

At first, most neuroevolution algorithms only evolved the weights of networks with fixed topologies [17]. Gradually approaches evolving even the topologies began

to appear. Notably, the Neuroevolution for Augmenting Topologies algorithm gained significant success as it was able to solve many problems introduced by its predecessors. The following paragraphs describing the encoding and genetic operators used for evolving are based on [7].

First, the concepts of phenotype and genotype need to be explained. While the genotype embodies the actual genetic representation, phenotype represents the physical form. The NEAT algorithms encodes its individuals into two list of genes, representing the genotype. The first is formed by a set of nodes, which excludes the input and output nodes. The other contains a set of connections, specifying the start and end node, its weight, innovation number and whether it is enabled. The phenotype on the other hand, is the neural network itself. The relation is shown in figure 3.6.

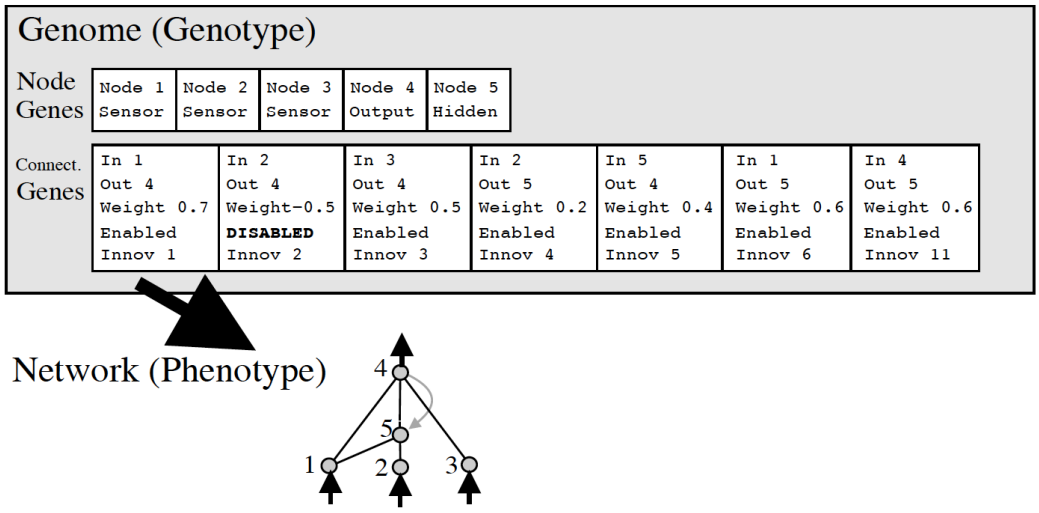

Figure 3.6: NEAT genotype and phenotype [18].

In case of a mutation, either a new structure is added to the network or an already existing connection is modified. The first case allows for additions of both new nodes and new connections. The new node is placed between certain two nodes which are already connected. The old connection is disabled and two new ones are created instead. The first goes from the start node to the newly added node and is assigned a weight of the same value as the old connection. The second, given a weight of one, starts from the new node and goes to the end node of the previous connection. In case a new connection is formed between two nodes, its weight is chosen randomly.

The crossover process revealed itself to be somewhat problematic. There is a high chance of creating a non-functional individual if simple blind crossing is used. For this reason, NEAT introduces something called historical markings. New changes, such as when a new node or connection is added, are marked with a historical number. This allows the genes to be aligned according to those number for crossover, as shown in figure 3.3.

Another key feature introduced by the NEAT algorithm is called **speciation**. It tries to solve the problem of the mutated or crossed individuals having lower performance and therefore not being given enough time to be optimized. The main idea of speciation is to divide the population into several species. This is

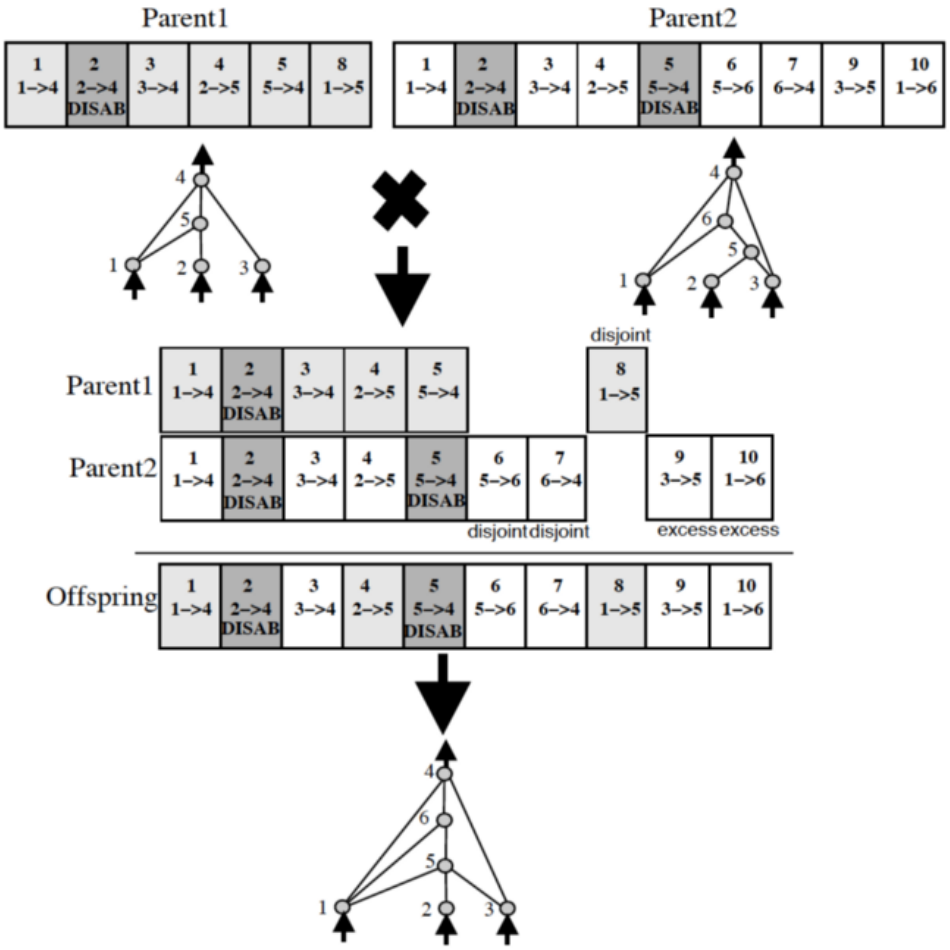

Figure 3.7: The alignment of genes according to the historical markings [18].

done based on the similarity of their connections and topologies. As normally it would be very hard to measure, the available historical markings make this approach a lot easier. The fitness of individuals is divided by the size of their species which enables new small species to survive. General standing is also available to prioritize species with better performance. This approach is called explicit fitness sharing.

Finally, in order to evolve minimal networks, the initial individuals only consist of input and output nodes with connections. The hidden nodes are left out.

4. AI Framework and Implementation

The core goal of this thesis was to implement a framework usable for AI implementation and subsequent testing. Similar frameworks exist, namely microRTS created for the purpose of AI research [13]. However, unlike in microRTS, the game in this framework is asymmetrical.

Main focus of this chapter is to describe the architecture of the AI framework and look at the different examples of AI implementations to illustrate its usage. More detailed overlook can be found in section A.2 containing the developer documentation. Lastly, we are also going to cover simple guidelines for implementing new AIs and adding them to the game.

To begin with, the implementation of both the game and the AI framework is written in C# and utilizes Unity[1]. There are two main modes available in the application. First is the game mode, which enables the player to play against a chosen AI or watch two different AIs play against each other. This mode has full graphical interface which was the main reason for using Unity. On the other hand, the training mode utilizes Unity only handful of times, mainly to enable certain scripts to be attachable to gameobjects in the editor. With this in mind, the implementation of the game logic itself can switch between using Unity in order to display the game or not using it to be able to be run on multiple threads.

4.1 Framework overview

The framework is primarily designed with evolutionary algorithms in mind in terms of one way to implement the AI for this game. Therefore, the fundamental goal of the framework is to support the implementation of trainable AIs and perform the training process itself. Set number of games are played and for each one certain statistical information is collected and given to the AI. This process is repeated for specified number of rounds, throughout this text called generations.

Naturally, non-trainable AIs are also supported, however, a slightly different approach is recommended to make their implementation easier. Even though these might not seem that interesting, they can be useful as opponents to train other AIs or simply just to test and balance the game itself.

For the purpose of efficiency, multiple threads are used for the training. Each thread runs one instance of the game and ends when one of the players looses or a set timeout is reached.

Functionality of the AI itself has to be divided into two scripts, the AI script itself, which handles the decision making process during the game, and the AI trainer script, which takes care of everything related to the training process. Now, lets look at both of these in detail.

[1] Unity Game Engine - https://unity.com/

4.1.1 AI Player

Every AI must derive from the AIPlayer base class. It holds information about the game state for the AI to inspect and manages the requests from the game in terms of action selection. AIPlayer itself derives from IPlayer, which is the common class for both human and non-human player.

The information accessible to the AI in the base classes convey current state of the game and can be used in various ways.

Properties

These are the properties in the base classes:

- **Side** - Indicates the side of the player, either attacker or defender.
- **Info** - Class containing information about the game state, it consist of 3 properties, OwnArmy, EnemyArmy and Map.

Army

As said previously, the AI can access the instances of the own army and the enemy army through the *Info* property. These are represented by the *Army* class, which contains all the units and buildings of a player. There are multiple properties and methods accessible, which this class accommodates. The properties are:

- **Side** - indicates the side of the player, either attacker or defender
- **Count** - total count of towers, buildings and troops
- **Money** - current amount of money available
- **MoneySpent** - amount of money spent until now
- **MoneyAll** - sum of spent money and currently available money
- **Troops** - provides a list of all of the player's troops currently alive in the game
- **Towers** - represents all player's towers, therefore it is empty for the attacker
- **Buildings** - same as towers but contains other buildings such as walls and gates

As shown, some of these properties provide access to the instances of troops, buildings or towers themselves, where more information can be obtained.

There are also multiple methods available, which report on the current game state. Depending on which army these methods are called on, the results differ.

- **SenseLowestHealth()** - Returns an instance of IRecruitable with the lowest health currently in the given army, where IRecruitable is a common interface for all buildings, towers and troops.
- **SenseHighestHealth()** - Similar to the previously mentioned method but looks for the highest health.

- **SenseTroopLowestDamage()** - Returns the troop with the lowest damage in the army.
- **SeneseTroopHighestDamage()** - Returns the troop with the highest damage.
- **SenseTroopLowestSpeed()** - Returns the troop with the lowest speed.
- **SenseTroopHighestSpeed()** - Returns the troop with the highest speed.
- **SenseLowestDamageDecrease(Attacker attacker)** - Based on the type of the given parameter, it checks for which IRecruitable in the army the attacker has the highest damage modifier. In other words, it looks for an IRecruitable which will receive the most damage from the attacker. The Attacker class is a common base class for all the objects that have the ability to attack, such as towers and troops.
- **SenseClosestTo(Attacker attacker)** - Returns the closest IRecruitable in the army to the given object in parameter.

Map

Even though all the troops, buildings and towers can be found through instances of Army, Map allows for inquiries about specific positions. As specified in section 2.1, it is an $M \times N$ matrix and can be indexed in that manner. Another accessible property in Map is a list of vectors, called CastleArea, specifying the positions of the defender's castle.

Methods

In order for the AI to work correctly, the base classes specify some methods that need to be implemented in children.

- **int PickToBuy()** - This method must implement the logic needed for the decision whether to buy a new troop and if so, which one. As there are many different unit types with different characteristics, UnitFinder, a static class, includes a list called UnitStats. It consists of individual structs called UnitInfo. These contain all the parameters mentioned in 2.2. When decision is made, the index of the chosen unit in UnitStats shall be returned or possibly -1 as an indication of no purchase.
- **IAction FindAction(Attacker attacker)** - This is probably the most crucial method where the action picking process must be implemented. Essentially, there are only two core actions that can be carried out - attack and move. However, certain details need to be described, such as who to attack or where to move. Therefore, for the purpose of AI implementation, several MacroActions are specified and will be presented in subsection 4.1.1. Instance of IAction, an out parameter of MacroAction representing the selected action, must be returned. Null is also considered a valid return value, it is simply interpreted as "I want to keep doing what I am doing".

- **AIPlayer Clone()** - As a result of multiple threads being used, the AI-Player child instance can be accessed from multiple threads simultaneously. In order to ensure correct behaviour, a clone of the player is needed. It is up to the programmer to decide, what needs to be copied. All the properties and fields from base classes are naturally taken care of and do not need to be considered in child classes. Specific examples can be found in section 4.2.
- **void RunOver(GameStats stats)** - During training, after a single run of the game finishes, the RunOver method is called. It gives the AI various information about the past round, such as the winner side and performance statistics for both armies.

MacroActions

The MacroActions static class is there to provide the AI with a simple approach to action selection. Multiple preset actions can be called with the use of this class. The general action method signature is as follows - bool DoSomething(Attacker attacker, out IAction result). The only exception is the AttackGiven method, which on top of mentioned parameters also needs the target to be specified. As can be seen, a boolean value is returned based on whether said action can be executed or not. The first parameter, called attacker, represents a troop or a tower trying to carry out the selected action. Finally, an out parameter called result is set in the code, it represents the action itself and its value shall be returned in the PickAction method. As mentioned, the following methods evaluate as false in case of an impossible action. This situation can arise in scenarios such as nonexistent enemy of desired criteria or no path available to a field in range of the target.

- **AttackClosest** - Finds the closest enemy or enemy structure and prepares the attack action.
- **AttackInRange** - Similar to previous method, however, it only looks for enemies and enemy object in range of the given troop. If none found, the action is labeled impossible, therefore, the resultAction parameter is set to null and false is returned.
- **AttackWithLowestHealth** - Looks for an IRecruitable with the lowest health to attack.
- **AttackWithLowestDamage** - An IRecruitable with lowest damage is searched for to be attacked.
- **AttackWeakestAgainstMe** - Picks an enemy troop which has the highest damage modifier, therefore will receive the most damage, against given attacker troop.
- **AttackGiven** - Tries to create an attack on given enemy.
- **MoveToSafety** - This method differentiates between the two sides. The defender moves back onto a field inside the castle. On the other hand, the attacker walks back to one of his spawn points.

- **DoNothing** - Equivalent to returning null in the PickAction method.

4.1.2 AI Controller and AI Trainer

The second component that every AI needs, as said before, takes care of the training process. Additionally, it also administers loading of a trained AI outside the training mode. However, unlike the previously mentioned AIPlayer, this class is always called from a single thread. Based on whether the AI should be trainable or not, there are two different base classes available to derive from. *AIController* is primarily meant for non-trainable AIs as it lacks some functionality. *AITrainer* is an extension of *AIController* adding those things specifically needed for the purpose of training. Both of these are going to be discussed in this subsection.

AI Controller

As non-trainable AIs are less complex, this class provides only a couple of properties and methods. First, here are the properties:

- **AIPlayerType** - Specifies the type of the player, that is controlled by this class.
- **Population** - Represents a read only list composed of instances of players of a certain type. For a non-trainable AI, it only contains a single player.

Second, there is only one method that requires to be implemented in children:

- **GetPlayer()** - This method serves the purpose of retrieving a certain player for the game outside of training mode. For example, it can return a strictly specified one or a random one from the population.

However, there are also 3 methods that can be overridden to gain additional functionality:

- **CustomStart()** - called once at the beginning of the training process
- **BeforeEachGeneration()** - called before the start of each generation
- **TrainingFinished()** - called after the training has finished

Overall, even though it might seem unclear for now, the implementation of a controller derived from AIController is pretty straightforward. Examples will be shown in subsection 4.2.1 and subsection 4.2.2, which describe implementation of two non-trainable AIs including the controller.

AI Trainer

All trainable AIs need a controller class derived from AITrainer, that must implement all the required methods. As noted earlier, this class is an extension of AIController, therefore inherits all the properties and methods from it. However, for the sole purpose of training handling, many new items are needed.

There is only one added field called population. It is an internal representation of the previously mentioned property Population, to allow children to change the content of the list.

On the other hand, methods contain many new additions including these, that are compulsory for implementation:

- **List<AIPlayer>CreatePopulation()** - This method should implement the creation of the AI players and return them in a list. For the evolutionary approaches shown in section 4.2, this method creates the initial population. It is guaranteed that each player from the population will play at least one game in a generation.
- **void GenerationDone()** - After every generation, the GenarationDone method is called. It is a great place, where to implement certain evaluation logic. As will be shown in section 4.2, this is where the evolutionary based AIs execute fitness evaluation, selection and apply the variation operators.
- **AIPlayer GetChampion()** - This method is primarily meant for retrieving a single instance of the best AIPlayer, however, as a matter of fact, any instance of AIPlayer can be returned.

Another key point, AITrainer on top of everything mentioned also manages loading and saving of trained AIs. Working implementation of load/save system is already present, however, these methods can be overridden if needed:

- **void SaveChampion(string file)** - Saves the champion, the return value of GetChampion(), to said file.
- **IPlayer LoadChampion(string file)** - Loads a player from the given file.

Further explanation of how to use the present implementation can be found in section 4.3.

Runner

Additionally, there is a possibility to create a custom training runner. This can be useful if a special approach to training is desired. For example, if a certain experiment requires data from multiple training sessions, a special runner can be implemented to handle that process. This functionality is recommended to be used only by individuals who already have a deeper understanding of the framework. Further explanation of usage and examples of implemented runners, **DefaultRunner** and **MultipleRuns**, can be found in section A.2.4.

4.2 Implemented AIs

As an illustration and to better demonstrate the usage of discussed classes, several AI implementation are explained in this section. To ensure full understanding, mentioned AIs represented both non-trainable as well as trainable examples. Their performance will be examined in the following section 5.

```
public class RandomAIController : AIController
{
    public override Type AIPlayerType => typeof(RandomAI);

    public override IPlayer GetPlayer()
    {
        return new RandomAI();
    }
}
```

Figure 4.1: RandomAIController implementation

4.2.1 Random AI

First very simple non-trainable AI is the RandomAI. As its name suggests, it works on the principle of randomly choosing an action to carry out and unit type to purchase. Even though this approach is very trivial and results tend to be uncertain, it can be useful for testing and showcasing the use of the framework. As mentioned previously, every non-trainable AI should implement a controller class derived from AIController. This implementation is quite trivial as can be seen in figure 4.1.

Furthermore, the implementation of important methods in RandomAI player, child of AIPlayer, is similarly straightforward. Figure 4.2 shows the approach taken for the FindAction method in pseudocode.

```
IAction FindAction(Attacker attacker):
    //If the object is currently not
    //doing any other action
    if (attacker. State == Free):
        return the out parameter a of random MacroAction

    if (once in a while even if state != Free):
        return the out parameter a of random MacroAction

    return null
```

Figure 4.2: Pseudocode of FindAction method for RandomAI

In fact, the PickToBuy method is even simpler. Essentially, it returns a random index from from the the set range - 0 to count of unit types, where the upper bound is excluded, as shown in figure 4.3.

```
protected override int PickToBuy()
{
    return rnd.Next(0, UnitFinder.UnitStats.Count);
}
```

Figure 4.3: PickToBuy implementation for RandomAI

4.2.2 Basic AI

Another example of an AI that is not trainable, is the BasicAI. Nonetheless, this one being somewhat more complex than the preceding one. Although the controller implementation is done in the same manner, the decision making methods are slightly more sophisticated.

First, the FindAction method makes use of all the different information accessible from the received attacker, as well as those present in OwnArmy and EnemyArmy. Furthermore, it considers whether chosen MacroAction is possible and if not, continues with the decision process further until feasible action is found or no options remain.

The logic of the AI is as follows. If the object in question is not already performing an action, it first tries to carry out an attack on an object in its range. If such action is impossible, it attempts to attack an enemy of type that is considered weakest against its type. If unsuccessful, it looks for an enemy with lower health to attack and then potentially for an enemy with lower damage in case the first search failed. On the other hand, if not free, it checks the type of the closest enemy. In case it is already attacking the closest one, nothing happens. Otherwise, if a building is the closest then it tries to attack a tower. If there are no towers left, again, the weakest type of enemy is attacked. If a troop is the closest enemy and has twice as much health as the action picking object which additionally must be the only troop on his side, then it tries to run to safety. Finally, if everything mentioned fails, the object just searches for the closest thing to attack.

In like manner, the PickToBuy method tries to utilizes various values from all the available data. The pseudocode can be seen in figure 4.4.

```
int PickToBuy():
    if (OwnTroops.Count is below set constant value):
        return Index of the cheapest unit

    if (MySide == Defender and I have no ranged units):
        for (int i = 0; i < UnitTypes.Count; i++):
            if (UnitType[i] is ranged and in my budget):
                return i

    //Tries to save up money if enemy is outnumbered
    if (OwnTroops.Count >= EnemyTroops.Count * 1.5):
        return -1

    //Gradually goes through the available UnitTypes
    tobuy = (tobuy + 1) % UnitTypes.Count
    return tobuy
```

Figure 4.4: PickToBuy pseudocode for BasicAI

Not to be overlooked, the Clone method for both of these AI implementations simply returns new instance of themselves. Nothing needs to be specifically copied

unlike in the following examples. The RunOver method is not needed at all and immediately returns.

4.2.3 Rules AI

Unlike the previously mentioned AIs, the RulesAI is the first example of a trainable AI. It utilizes evolution to improve itself, based on the value obtained from the fitness function.

Essentially, there are two different populations inside the RulesAI trainer:

- **the strategy population** - The strategy population consist of individuals, which take care of the decision making process in the PickAction function of the player. Those individuals consist of a list of rules. Each rule is composed of multiple conditions, which assess the current state of the game or a given object, and an index representing a possible action to carry out.
- **the shopper population** - Next, the trainer includes a population of individuals managing the selection of units to be purchased. Each individual is a simple sequence of indexes, representing the unit types to be bought.

In the first place, a strategy and shopper population of randomly generated individuals are created. The populations are then mapped onto a group of RulesAI players, which undergoes the evaluation. In other words, each player is assigned one of the individuals from both populations. This happens before every generation. Each individual is then assigned a fitness value from the player instance it resides in. This approach can be considered an example of the cooperative coevolution, described in section 3.2.2, between the two types of individuals.

Conditions

As mentioned, the rules forming a strategy individual consist of multiple conditions. To point out, their implementation is part of the AI implementation, not the framework itself. There are multiple condition available:

- **Damaged** - true if given object is damaged
- **Free** - state of the given object must be set to free for this condition to be true
- **HealthierThanClosest** - true if the closest object has lower health
- **ClosestIsTroopBase** - true if the closest IRecruitable is of type Troop
- **ClosestIsBuilding** - true if the closest IRecruitable is a building
- **ClosestIsTower** - true if the closest IRecruitable is a tower
- **IsDefender** - true if the given object belongs to the side of the defender
- **IsAlone** - true if the side of the given object currently has less than two troops

- **IsWinning** - true if the side corresponding to the side of the given object has more troops than the enemy
- **IsInsideCastle** - true if the object is inside the castle area
- **IsInTowerRange** - true if the object is within a tower's range

Inside the PickAction method, each rule in the player's appointed strategy individual is checked. If all of the conditions are met, the action corresponding to the index set in the rule receives a vote. RulesAI uses serializable versions of actions described in subsection 4.1.1, except AttackGiven. The left out action requires a target to be specified which is not usable with this approach. At the end, after all of the rules are evaluated, the action with the most votes is carried out.

For the PickToBuy method, a slightly different approach is adopted. The AI simply returns the value from the current position in the sequence, representing the shopper individual, and increments the position.

After each generation, all individuals in group of RulesAI players are evaluated based on their performance. The fitness function takes into account several factors. Constant amount of points is awarded for a win. Further, the difference between the total damage dealt by own troops and damage received from the enemy is multiplied by a constant and added to the fitness value. Last component are the kills, again multiplied by a certain value. In case an individual plays more games and is therefore assigned more than one fitness, the median is taken as the final value.

All individuals in both the PickAction managing population and PickToBuy population are assigned fitness corresponding to the fitness of a player that utilized them. Again, median value is taken if more fitnesses are stored. In both, roulette wheel selection is applied in order to produce the parent population and elitism is employed to ensure survival of the best individual. However, crossover and mutation differ between the two populations. The strategy population utilizes the uniform crossover with two offspring. Each rule in the parent has a 50% chance of being placed in the first child. Naturally, the rule from the second parent goes to the other child.

On the other hand, the shopper population uses a simple one-point crossover. The sequence of numbers in each individual is split at a random point and the parent parts then form the two children exactly as shown in figure 3.3. The process of mutation is fairly simple for individuals from both populations. In case of the strategy individual, the action index present in each rule is the value subjected to possible change. The probability of this change will be one of the parameters studied in the subsection 5.2. Mutation in the shopper individual is even more straightforward. Random index in the sequence contained in the individual is chosen and is assigned a random value from appropriate interval.

Final point to discuss is the implementation of the Clone and RunOver methods. The latter one is trivial, it only saves the received statistical information for future use. While quite simple itself, the Clone method requires certain functionality from the two individuals in the player. Both need to be able to provide a copy of themselves. That is something the clone method cannot handle by itself as it does not know the full internal structure of the individuals.

4.2.4 Coevolution AI

This next AI is actually a variation of RulesAI. It works exactly as explained in the previous chapter with one significant difference. There are three separate strategy populations. Each one corresponds to certain category of units. This approach corresponds to the cooperative coevolution described in subsection 3.2.2. In other words there are three different species that evolve separately.

First is the species representing melee units or simply units with range equal to one. Ranged units form the second category. Finally, the last category of species represents the towers.

With this in mind, the player individual now contains two additional strategy individuals that take part in the action selection process. The PickAction function now first classifies the object and then uses the individual of corresponding category to select an action for the object.

The PickToBuy method stays the same as there were no changes made to the shopper population.

4.2.5 NEAT AI

The last implemented AI is the the NEAT AI. As its name suggests, it utilizes the principle of the NEAT algorithm described in section 3.3.2. The implementation of the NEAT itself is a public version of the UnitySharpNEAT library, a port of SharpNEAT for Unity [2]. It is used under the MIT license. Therefore, this section will only briefly describe how it was utilized in the AIPlayer and AITrainer child implementations. The code and logic of the UnitySharpNEAT will not be discussed.

For the NEAT player an additional base class was created, called INeatPlayer, which essentially replaces the UnitController base class described in the library description. It contains a field of IBlackBox, library interface representing the "brain" of the algorithm, which is then used in the override of the FindAction method. First the UpdateBlackBoxInputs, it is an abstract method which must be overridden in the children and give input to the IBlackBox provided. Then the Activate method calculating the outputs is called on the IBlackBox. Final step of the FindAction method executes UseBlackBoxOutpts method, which must be implemented in the children. For NEAT player this method must implement the action selection based on the given outputs from NEAT. The action corresponding to the index of the highest output is selected for execution. The implemented player uses the conditions as input for the IBlackBox and the MacroActions as output, both were described earlier in this chapter.

In the trainer script, each player instance is assigned a single IBlackBox. After each generation the population is evaluted and modified by the NEAT library. Additionaly, this is the only AI implementation using a custom load/save system. The methods from the base class are overriden and system provided by the library is utilized.

[2] UnityNEAT - https://github.com/flo-wolf/UnitySharpNEAT

4.3 Creating your own AI

As this framework is designed to test and train AI players for the game, it is desirable for new AIs to be added. In order to successfully implement a usable AI, several rules must be followed.

4.3.1 Scripts

To begin with, one must make the decision whether the artificial intelligence will possess the ability to improve itself through training or not. Based on this decision, a script containing the implementation of either a child of AIController, for a non-trainable AI, or a child of AITrainer, for a trainable AI, must be created. As already mentioned, this class is always called from the main thread. This fact enables the usage of any Unity related methods or properties in the class.

Equally important is a properly implemented child of AIPlayer. This part of the AI is the one that actually plays the game and makes all the decisions. To make simultaneous training on multiple threads possible, this class must be able to be used in other threads besides the main one. For this reason, the usage of anything from the UnityEngine API is prohibited in all predefined methods, as it can be only run from the main thread. Vector is the only exception to that rule. However, any custom methods that are to be called only from the controller or the trainer may also make use of the Unity scripting API.

For the same reason, the clone method must make a deep copy of everything, that might cause a race condition if shared between multiple instances.

4.3.2 Load and save system

One of the important responsibilities of the framework is the ability to save and afterwards load the trained AIs. In order to achieve this, the child of the AIPlayer, that implements all the game logic, must be savable. As there are many possibilities for serialization, to simplify the implementation of own artificial intelligence, ready to use load/save system is already present. However, to ensure its proper functionality, serializable classes and properties must be labeled. The class itself must be marked with [DataContract] attribute and the properties and fields with [DataMember]. Short example can be seen in figure 4.5.

```
[DataContract]
public class Example
{
    //Will not be serialized
    public int SomethingUseless;

    [DataMember]
    public int SomethingToSerialize;
}
```

Figure 4.5: Example of a serializable class and field

In case a different approach to serialization is needed, the two AITrainer methods SaveChampion and LoadChampion mentioned in subsection 4.1.2 can be overriden with custom implementation.

4.3.3 Adding AIs in Unity

For the framework to properly register the newly programmed AI, it must be added inside the Unity editor. Once the required scripts are implemented, a new prefab game object with the AITrainer or AIController child is required. It then must be placed into the AIOption list, which can be found in the Menu scene on the UIController game object. The scene structure is described in section A.2.6 in the attachments.

5. Experiments

Finally, the goal of this thesis, besides creating the framework itself, was to test several possible approaches to AI creation. This last chapter will discuss all the different experiments performed both to balance the game, certain parameters of the several AIs and compare the implemented AI variants.

5.1 Game Balancing

The first important goal of the experiments was to properly balance the game. Due to the asymmetrical gameplay, it was crucial to ensure neither the attacker nor the defender are in a dominant position from the start of the game. Both sides should have a an equal chance of winning and it should be only influenced by their true performance. However, absolute fairness is hard to achieve, therefore the objective was to at least get as close as possible.

There are several parameters that influence the flow of the game. First, there are the characteristics of the implemented unit types, namely health, speed, damage, reload rate, range, price and bundle count. All of them were discussed in section 2.2. Even thought all of the movable units are common for both the attacker and defender, significant superiority of certain unit type is not desired. Additionally, the tower setting is truly important. They represent one of the advantages of the defender side, however, if too strong the defender might become unbeatable. The second category of essential parameters consists of the amount of starting money and the income.

Game balancing can be incredibly time consuming, especially for asymmetrical games. Simple playtesting not only requires a lot of time but also people to actually play the game. For this reason a different approach using AI was employed. Both the attacker and defender were assigned a player of the type RandomAI, described in section 4.2.1. Total of 500 games were repeatedly played collecting data about the winner of each game. If one side significantly dominated the other, the parameters were modified in favor of the loser. Mainly the money related parameters and tower settings were balanced via this trial and error method. Additional changes to other parameters were made based on the experience of a human player in the game.

The final setting for the unit types was presented in chapter 2. The remaining parametrs are presented in table 5.1.

	Starting money	Income
Attacker	16000	1600
Defender	12000	1200

Table 5.1: Starting money and income settings

5.2 AI Settings Experiments

As RulesAI and CoevolutionAI both evolve and use several genetic operators, the probabilities of those need to be set to certain values. To maximize the performance of mentioned AIs, several experiments were conducted in order to find a good setting.

Each AI was run as both attacker and defender with three different settings of crossover, mutation and action mutation probability. BasicAI was selected as the opponent for this experiment as RandomAI might cause less consistent results. Once the best setting was discovered, several distinct individual length settings were also tested.

In the following section multiple graphs displaying the outcome will be presented. All experiments were run for fifty generations with the AI playing three games in each one ,setting the median as the final fitness. In order to get more accurate results, the graphs show the average fitness counter over specified number of training sessions.

5.2.1 Experiment 1

As already mentioned, the goal of the first experiment was to test several different genetic operator probabilities settings. The tested values for each run are described in 5.2.1.

The three values being changed are crossover probability, mutation probability and action mutation probability. The first simply determines, how big is the chance a crossover will take place. Both RulesAI and CoevolutionAI use uniform crossover. The second is very similar and determines whether an individual will undergo mutation or not. Once an individual is selected for mutation, the action mutation probability is applied. As described in the section 4.2, the individuals consist of multiple rules and each has an appointed action index. The action mutation probability is the probability of the action index actually being modified.

First the **RulesAI**, described in 4.2.3, performed to some extent differently as attacker and defender. When used on the attacker side, all of the three settings yielded very similar results as can be seen in figure 5.1. None seem to display significant improvements over the fifty generations. That might be due to the fact a relatively successfully individual might have been generated right away. However, setting number two seems to be slightly worse than the others as the fitness values throughout generations tend to be around the 1.10 mark. On the other hand, for the defender side there is more variation between the settings as shown in figure 5.2 . Unlike in the previous case, the setting number three did not perform well in comparison to the other two. The setting number 2 seemed to be having the best results in the first half of the training but did not show a significant rate of improvement. Therefore, the setting number one with its great progress around generation 25 ended up scoring the best results. With this in mind, the setting number one was selected for further experiments and use of the RulesAI for both the attacker and defender side.

The **CoevolutionAI**, introduced in section 4.2.4, also produced varying results for the two possible player sides. As shown in figure 5.3, when the AI

	Crossover	Individual Mutation	Action Mutation
Setting 1	0.25	0.05	0.1
Setting 2	0.3	0.02	0.08
Setting 3	0.2	0.03	0.05

Table 5.2: Probability of variation operators

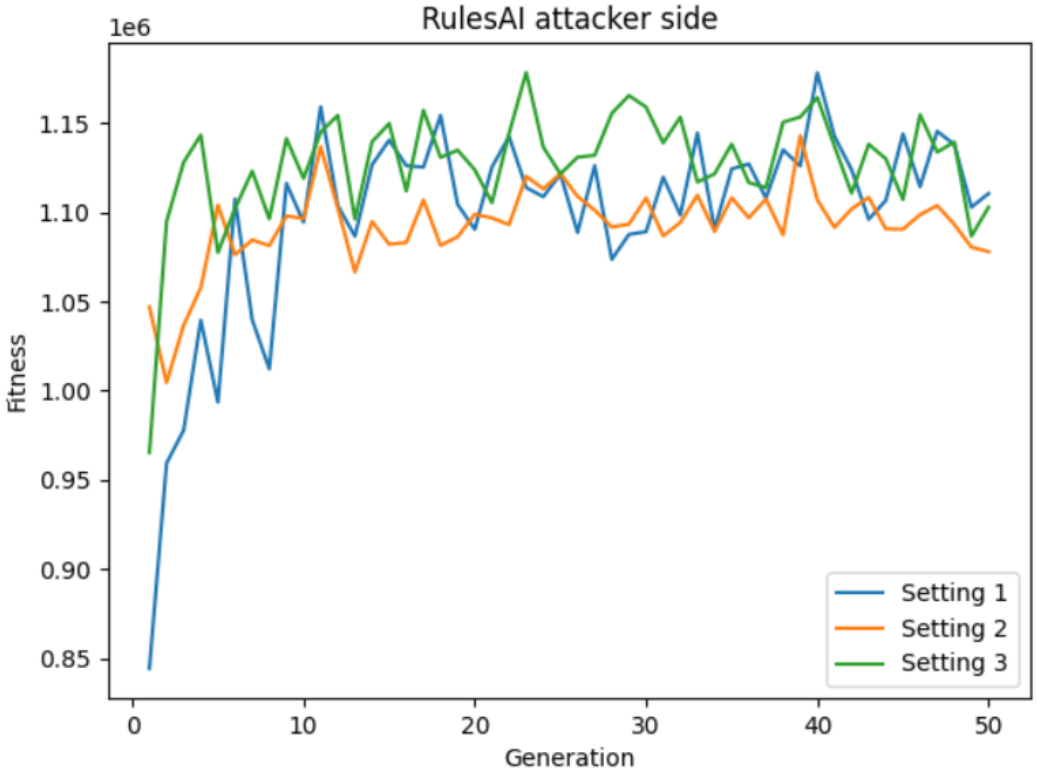

Figure 5.1: Average **attacker** fitness value taken from 5 training sessions.

trained on the attacker side, there were only moderate differences between the three tested settings. Conversely to the RulesAI experiment, this time the setting number two ended up achieving the best results and setting number one did not perform that well. Similarly to the previously tested AI, the defender side yielded more variation between the three settings as can be seen in figure 5.4. This time, the CoevolutionAI seemed to accomplish best performance with the setting number one, which was interestingly the weakest one for the RulesAI. On the other hand, the two remaining settings produced truly underwhelming results in comparison to the best one. Unlike in case of the previous AI, this time the chosen settings differ between the attacker and defender. As already mentioned the attacker performed the best with the second setting and the defender with the third one. For this reason, those are the settings used for further experiments.

5.2.2 Experiment 2

The second experiment continues with the task to find a good setting to then be used later when comparing different implemented AI approaches. This time, the

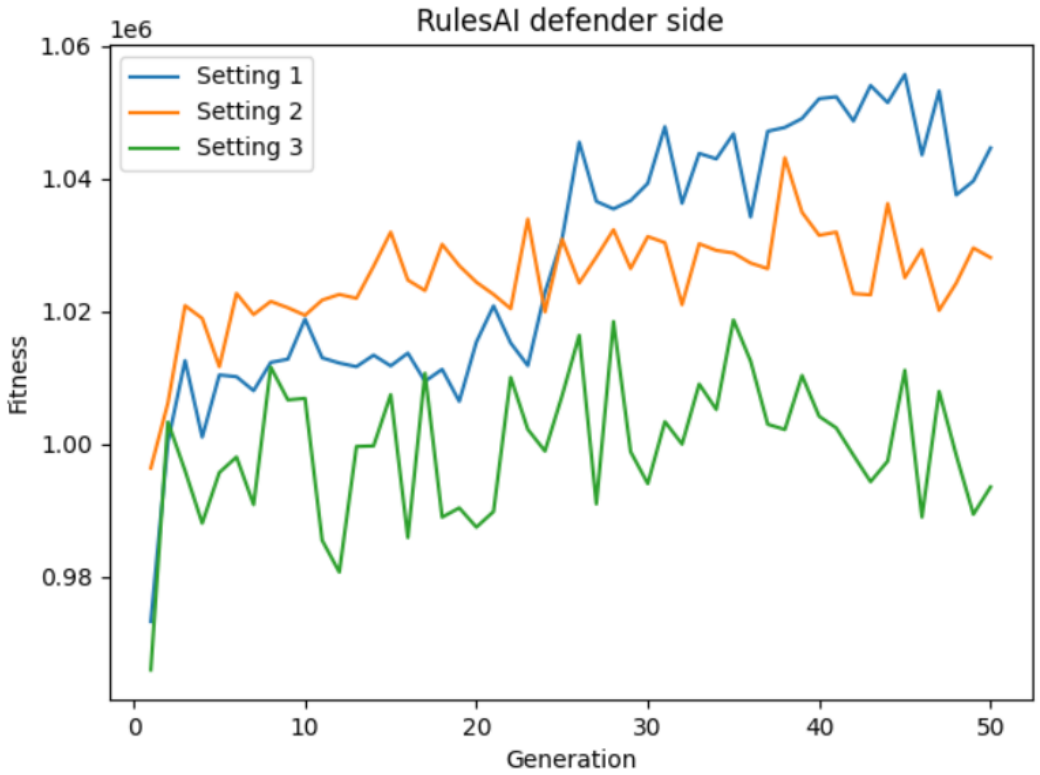

Figure 5.2: Average **defender** fitness value taken from 5 training sessions.

length of an individual is the parameter in question. To clarify, the size specifies how many rules are present in an individual. Further description of the RulesAI and CoevolutionAI individuals was presented in section 4.2. Based on the results of the previous experiment, the best performing probability was picked for each pair of AI and side. This experiment utilizes those settings, which can be seen in table 5.2.2.

In this experiment, three different individual length settings are tested: 7, 15 and 20. Again, the experiments are done for both the attacker and the defender side.

	Setting	Crossover	Mutation	Action Mutation
RulesAI attacker	1	0.25	0.05	0.1
RulesAI defender	1	0.25	0.05	0.1
CoevolutionAI attacker	2	0.3	0.02	0.08
CoevolutionAI defender	3	0.2	0.03	0.05

Table 5.3: Genetic operator settings used for each AI

To begin with, the **RulesAI** uses the same operator setting for both attacker and defender. The individual length that seemed to work the best for the attacker side were 7 or 20 as shown in figure 5.5. Those two yielded very similar fitness results throughout the fifty generations. On the other hand, the individuals consisting of twenty rules performed slightly worse. As can be seen in figure 5.6, the results of the experiment for the defender side followed somewhat similar

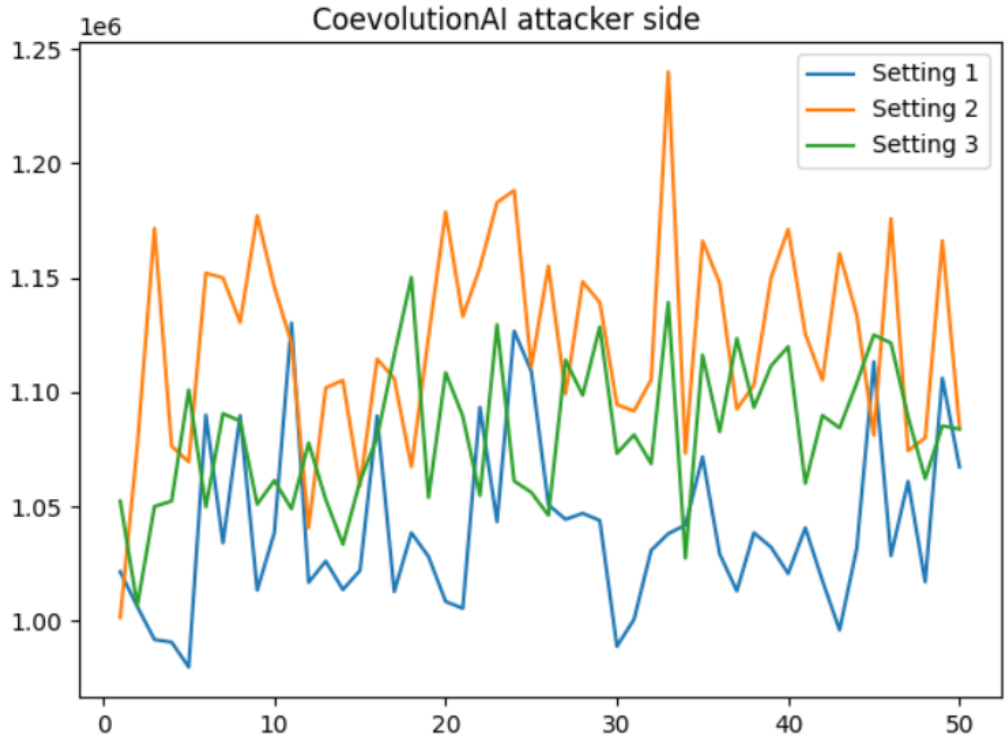

Figure 5.3: Average **attacker** fitness value taken from 3 training sessions.

pattern. However, this time the individuals of size 7 managed to outperform the ones of set size of 20. Similarly to the attacker side, the last setting, size of 15, overall produced worst fitness values.

Finally, the CoevolutionAI was also tested with different individual length settings. The attacker side performed somewhat similarly for all three lengths. Even though the population with individuals of size 7 produced the best results as shown in figure 5.7, it does not display a significant improvement over the training session. That might be due to the reason a winning individual was present from the start. However, its fitness results still consistently outmatched the other two settings and also earned best fitness overall. On the other hand, individuals of length 20 scored worst fitness values and only a couple of time managed to match the results of setting of length 7 and 15. As can be seen in 5.8, the defenders performance yielded truly different results in terms of distinction between the three settings. Again, the length set to seven managed to produce the best results. The second setting with length of 15 did not show much improvement and the fitness values stagnated throughout the fifty generations. Last, the population with individuals of size 20 achieved truly inconsistent results and overall showed an underwhelming performance in comparison to the other two.

Overall, the setting using length of 7 performed the bests in all discussed cases. This might be due to the fact, that there are 7 possible actions from which the rule's action index can be selected.

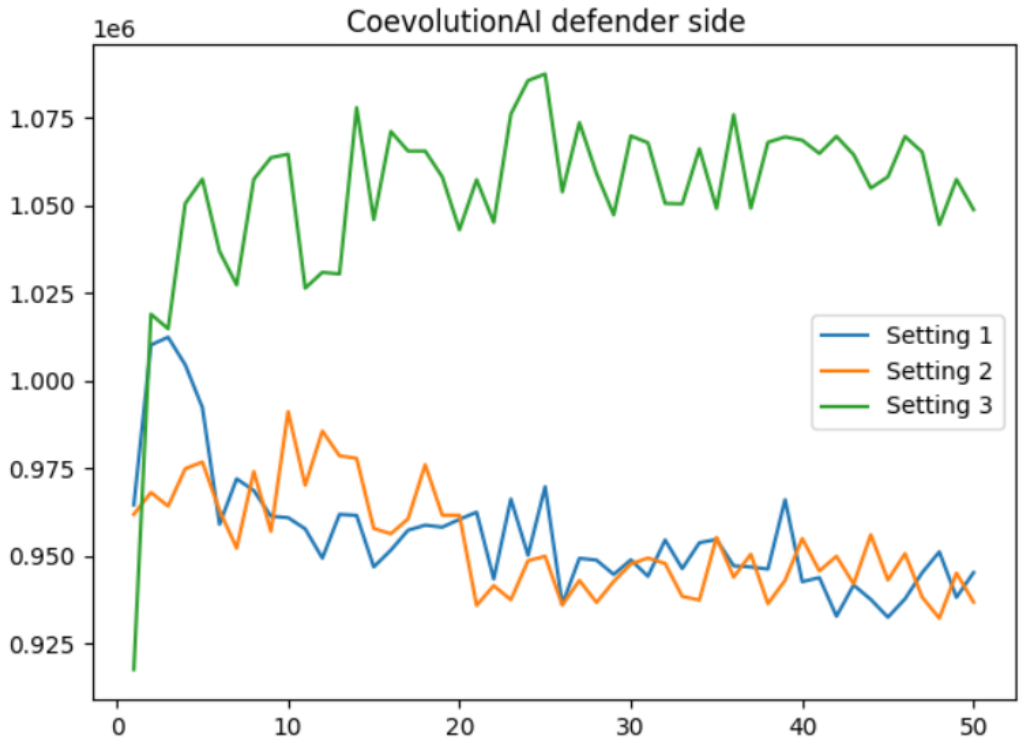

Figure 5.4: Average **defender** fitness value taken from 3 training sessions.

5.3 AI Comparison Experiments

The final part of this thesis focuses on assessment of the implemented AI approaches and used training methods. Each trainable AI was trained over the course of 50 generations. They were run three times to get more accurate results and limit cases of individuals getting a lucky win. Several different opponents were used for the training to examine the influence the change has on the trained individual. To point out, these experiments utilize the settings selected in the previous sections.

5.3.1 Experiment 1

The goal of the first experiment was to analyze the differences in performance between AIs trained against the BasicAI and ones trained against RandomAI. Each of the three trainable AIs was run on both the attacker and the defender side to obtain trained individuals for both sides. The saved trained individuals were then faced against the mentioned AIs, BasicAI and RandomAI, in a total of 100 games and data about the win-tie-loss ration was collected. The assessed options for each AI will will be as follows:

- **AttackerBasic** - This represents an individual trained for the attacker side against the BasicAI.
- **AttackerRandom** - Similarly to the previous one, but RandomAI was used at the opponent for the training session.

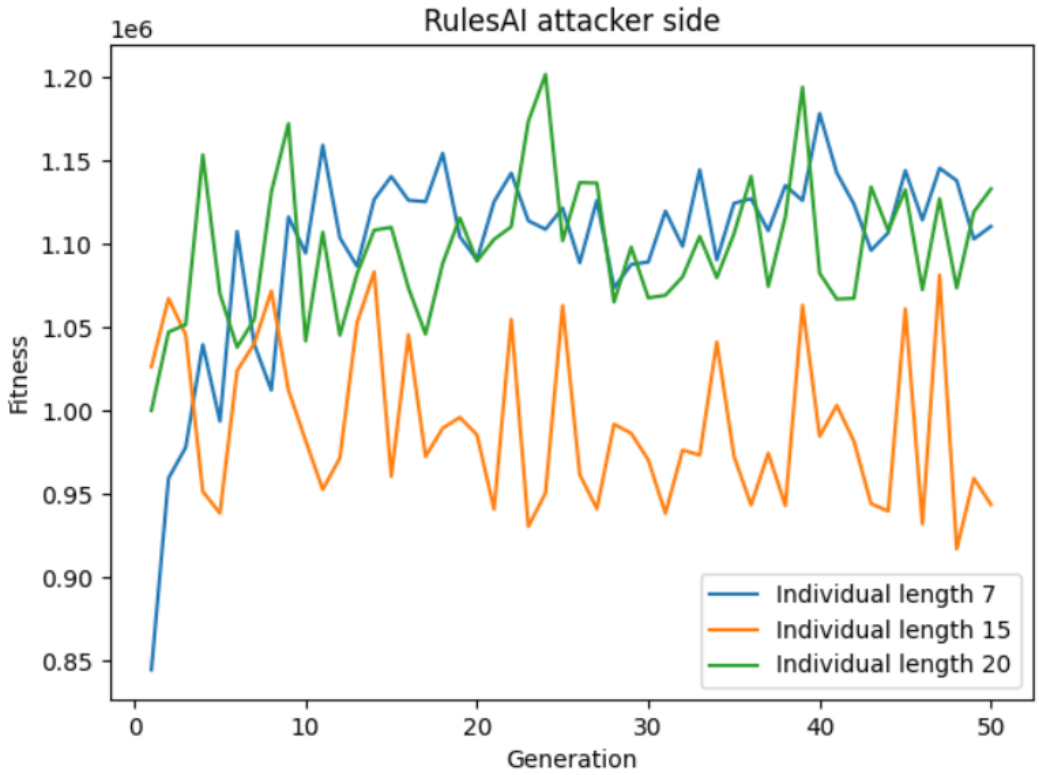

Figure 5.5: Average **attacker** fitness value with different tested individual lengths. Taken from 3 training sessions.

- **DefenderBasic** - Individiual, which was trained to play on the defender side against the BasicAI as attacker.
- **DefenderRandom** - The last one was trained on the side of the defender and played against the RandomAI.

First, the RulesAI on the side of the attacker performed better when trained against the BasicAI, as can be seen in table 5.3.1. The trained player managed to win 74 games against the BasicAI and all 100 games with RandomAI as opponent. On the other hand, the individual taught to play against RandomAI only managed to win 50 out of a 100 games against the BasicAI player. In this case, the first training option achieved to obtain a more versatile individual that was able to defeat both tested AI opponents.

On the other hand, the results for the defender side yielded somewhat different results. The individual trained against the BasicAI managed to then beat the AI in 98 games. However, the one trained against RandomAI was able to win in 97 games. Both performed similarly well against the RandomAI. Unlike in the previous example, the difference of the opponents chosen for training did not affect the final results very much. The RandomAI could be even considered to be slightly better as it scored less losses over the two matches than the other AI player.

The CoevolutioAI trained on the attacker side followed similar pattern to the previous example, as can be seen in table 5.3.1. The attacker trained against BasicAI managed to then beat it 76 times while the one trained against RandomAI

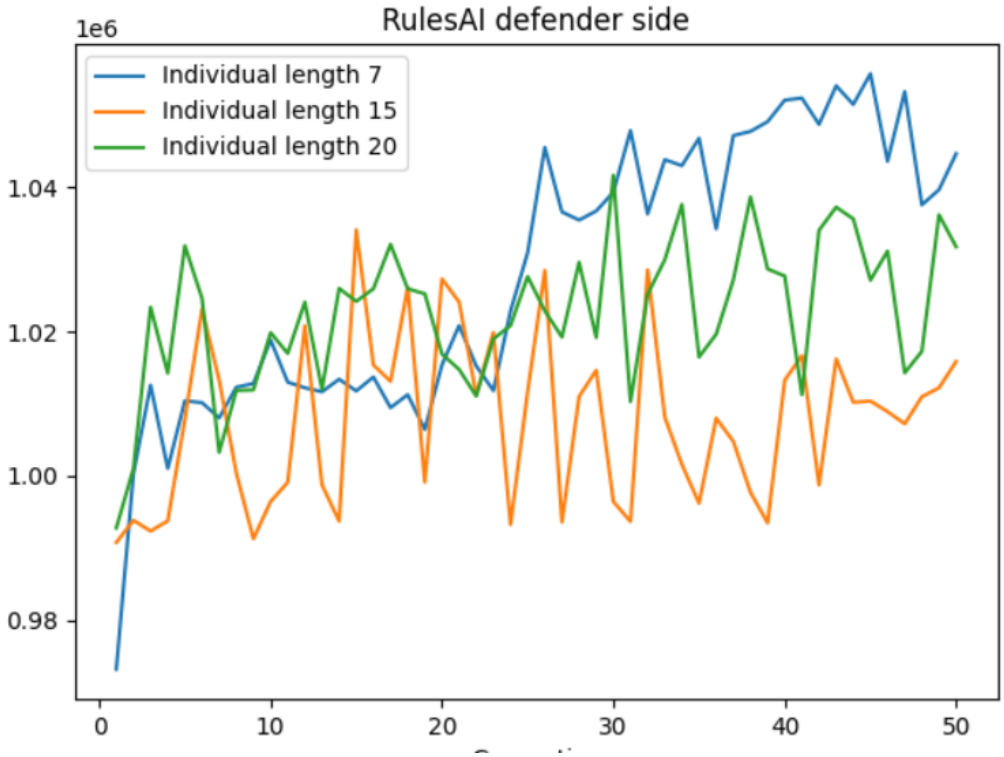

Figure 5.6: Average **defender** fitness value with different tested individual lengths. Taken from 3 training sessions.

	BasicAI	RandomAI
AttackerBasic	74/26/0	100/0/0
AttackerRandom	50/49/1	82/17/1
DefenderBasic	98/0/2	68/17/15
DefenderRandom	97/3/0	70/19/11

Table 5.4: Win/Tie/Loss ratios for differently trained **RulesAI** players faced against BasicAI and RandomAI

only four times. This is even significantly smaller than in the case of RulesAI. Against RandomAI both trained variant performed comparably winning over a 90 games.

However, the defender side trained individuals achieved drastically different results that in the case of RulesAI. The individual trained against RandomAI did not manage to win a single game against the BasicAI player. Unlike his rival, who managed to win all 100. With this in mind, for the coevolutionary approach the type of opponent selected for training had significant impact on the final performance.

Finally, as can be seen in table 5.3.1, the NEAT AI did not manage to achieve great results at all when playing against the BasicAI. Both the AttackerBasic and AttackerRandom trained individuals scored below 20 wins, the latter one only 2. However, most games were not total losses but ties between the players. When facing the RandomAI, the individual trained against BasicAI managed to win 54 games but lacked the behind the AttackerRandom's 98 wins. Clearly,

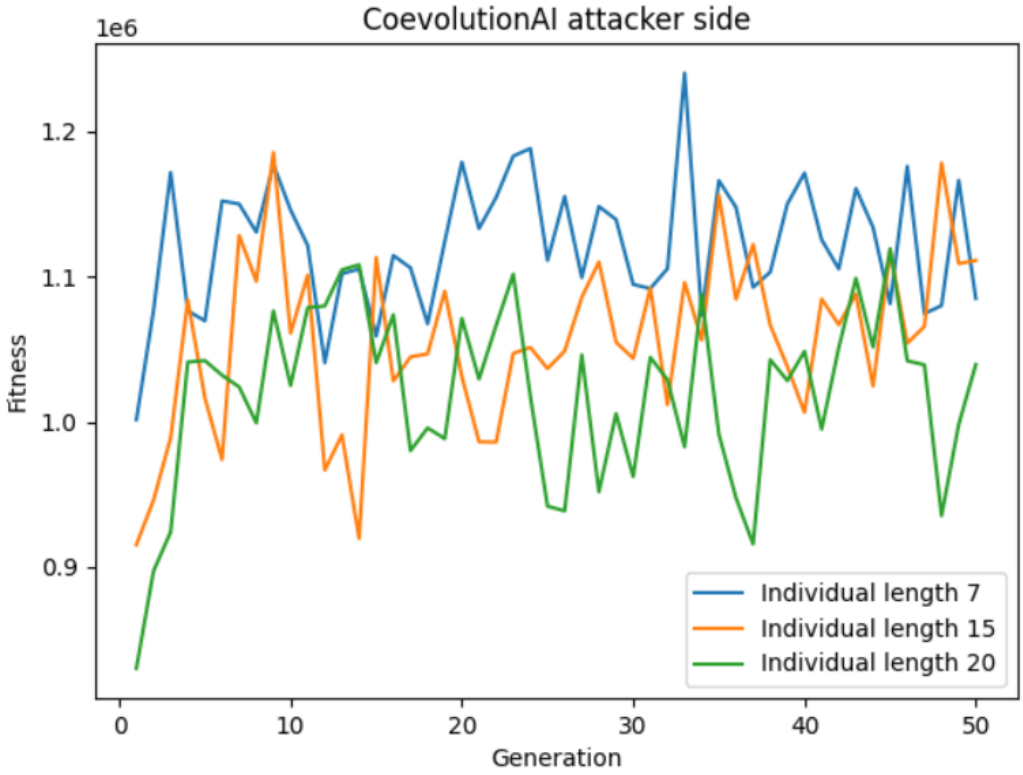

Figure 5.7: Average **attacker** fitness value with different tested individual lengths. Taken from 3 training sessions.

	BasicAI	RandomAI
AttackerBasic	76/0/24	99/1/0
AttackerRandom	4/33/63	97/3/0
DefenderBasic	100/0/0	75/13/12
DefenderRandom	0/0/100	79/19/2

Table 5.5: Win/Tie/Loss ratios for differently trained **CoevolutionAI** players faced against BasicAI and RandomAI

AttackerRandom was able to utilize the skill gained from the training against the RandomAI player.

The defender side performed even worse. Neither of the two trained individuals managed to beat the opponents more that three times and most lost more than 90% of the hundred games. There is one exception, the DefenderRandom tied in all 100 games against the RandomAI player.

As the presented experiments shown, the individuals trained against BasicAI tend to achieve better results and therefore will be used for further experiments. However, as the RulesAI's DefenderRandom performed similarly, it was chosen instead of DefenderBasic.

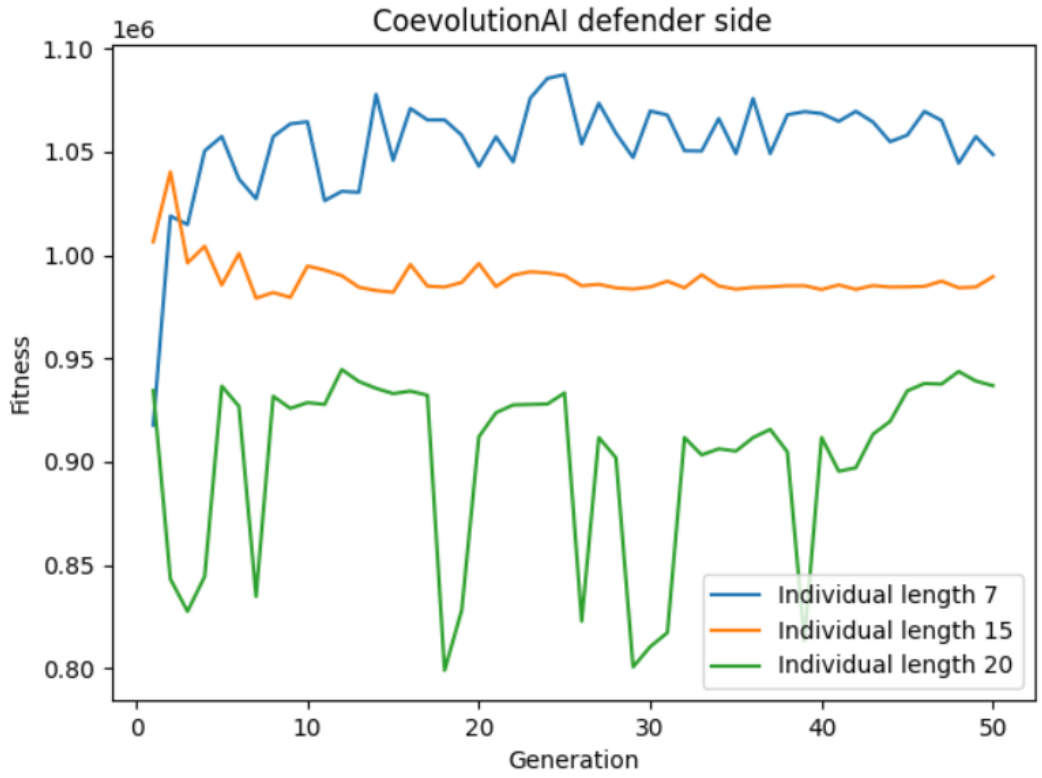

Figure 5.8: Average **defender** fitness value with different tested individual lengths. Taken from 3 training sessions.

	BasicAI	RandomAI
AttackerBasic	17/69/14	54/13/31
AttackerRandom	2/76/22	98/2/0
DefenderBasic	0/0/100	3/3/94
DefenderRandom	0/1/99	0/100/0

Table 5.6: Win/Tie/Loss ratios for differently trained **NEAT AI** players faced against BasicAI and RandomAI.

Experiment 2

In the second experiment, two slightly different approaches to training were used and evaluated. This time, the trainable AI variants were selected as opponents for the training.

First utilized the approach called competitive-coevolution, described in section 3.2.1. Essentially, RulesAI player was set as both attacker and defender and the training session was launched. As both players used were trainable, it enabled them to actually train each other. To analyze the results of this approach, both the trained attacker and defender were than run against the BasicAI and RandomAI. These individuals are further throughout the text referred to as AttackerRules, trained as attacker, and DefenderRules, trained as defender. As can be seen in table 5.3.1, the attacker trained with this approach achieved great results. The individual managed to win over 95 games against both opponents.

On the other hand, the individual trained as defender only managed to beat the RandomAI. Against the BasicAI, all 100 games ended up with a loss.

	BasicAI	RandomAI
AttackerRules	96/4/0	100/0/0
DefenderRules	0/0/100	88/12/0

Table 5.7: Win/Tie/Loss ratios for trained **RulesAI** players via coevolution faced against BasicAI and RandomAI.

Second approach tested in this experiment utilized the CoevolutionAI player. Instead of taking two and training them against one another, they both first faced the BasicAI for 25 generations. One as an attacker and the other as defender. Then the trained defender played against the CovelutionAI player set as attacker and vice versa for the trained defender. This was run for another 25 generations. Similarly to the previous method, the trained individuals were than faced against BasicAI and RandomAI to asses their performance. They are referred to as DefenderCross and AttackerCross. As can be seen in table 5.3.1, the results are vastly different from the coevolutionary approach. Conversely to the previous method, the defender's performance is drastically better. It manages to win all games against BasicAI and 78 games against RandomAI. The attacker, on the other hand, behaves similarly to DefenderRules in previous section, It manages to outplay the RandomAI play but looses every game against BasicAI.

	BasicAI	RandomAI
AttackerCross	0/2/98	85/3/12
DefenderCross	100/0/0	78/15/7

Table 5.8: Win/Tie/Loss ratios for trained **CoevolutionAI** using the approach described in experiment 2.

Experiment 3

In the final experiment, data over the course of a hundred games are collected for all the variously trained AIs. Performance was tested for each pair of trained individuals, one trained as attacker, the other as defender. The analyzed option are as follows:

- **RB** - This represent the RulesAI individual which was trained against BasicAI.
- **RR** - This one in the RulesAI individual trained against RandomAI.
- **CB** - Similar to the first one, however for CoevolutioAI player.
- **NB** - NeatAI individual trained against BasicAI.
- **RRule** - The individual presented in the previous experiment, where RulesAI trained again itself.

- **CCross** - Also presented in the previous experiment, it represents individuals who first trained against BasicAI and then against already trained individual of CoevolutionAI.

As shown in table 5.3.1 containing the results of all games played between each pair of AIs, it can be said that they behave somewhat differently depending on the opponent. On the attacker side, the RRule easily managed to beat the RR, CB, NB and CCross but seems to have had problems against itself as the defender. The RB and CB, while not having as strong performance as RRule against the four mentioned opponents, they do not seem to struggle against RRule on the defender side. The worst two trained individuals are NB and CCross, which are outperformed by each opponent except NB.

Overall, the defender side seemed to struggle in most settings. The RR as defender, along with CB and CCros all managed to beat NB. Additionaly, NB and RRule showed great performance against CCros both scoring 100 wins. The rest of the matches were pretty much all in favor of the attacker side, This may be an indicator that further game balancing might be needed. However, as previous experiments shown, the game is not always dominated by the attacker side and defender can easily end up winning all 100 games.

	RR	CB	NB	RRule	CCross
RB	99/1/0	97/1/2	100/0/0	90/8/2	100/0/0
CB	93/6/1	97/3/0	100/0/0	99/1/0	99/1/0
NB	0/11/89	0/6/94	76/3/21	3/97/0	0/8/92
RRule	100/0/0	100/0/0	100/0/0	65/35/0	100/0/0
CCross	25/75/0	21/79/0	96/4/0	0/0/100	24/76/0

Table 5.9: Win-Tie-Loss ratios for differently trained AIs faced against each other. The left column represents the attacker side while the row the defender side.

Conclusion

For the purpose of this thesis, a simple strategy games was implemented as an environment for AI testing. It simulates two armies of various types of units, where one army resides inside its castle which is being attacked by the other one. Along with the game, a framework for AI implementation and training was created. The framework was then utilized for implementation of several AIs methods using evolutionary approaches. Final part of the thesis described several experiments conducted to balance the game, tune the AI settings and analyze their performance.

In the first experiment, the goal was to ensure neither the attacker or the defender significantly dominate the game. Multiple game sessions were run with RandomAI to ensure that the win-rate ratio is somewhat evenly distributed. Clearly, the defender's castle poses an important advantage. Therefore, in order to counter balance this leverage, the attacker's money income along with the starting money ended up being higher than of the opponent.

Second part of the experiments focused on analyzing the implemented AIs and different approaches to their training. The initial test aimed to asses how much the opponent selection for training influences the performance of the trained individual. The obtained results showed that generally the individual trained against the BasicAI player outperforms the one trained against RandomAI. This stands for both the attacker and defender sides.

In the following experiment, two different approaches for training were analyzed. The main goal was to utilize training against other trainable AIs or their already trained individuals obtaining both the trained attacker and defender. However, the used methods achieved truly contradictory results. While one seemed to be working well for the attacker side and not the defender, the other did the exact opposite. With this in mind, it can be concluded that this approach might work well for obtaining a well performing individual for one of the two sides but not both.

The third and last experiment finally let all the multiple trained individuals play against each other. The results showed that the NEAT trained individual significantly lacked behind others on both sides of the war conflict. On the other hand, the RulesAI and CoevolutionAI individivudals, both trained against BasicAI, managed to achieve consistent performance with excellent win to loss ratio on the attacker side. The CoevolutionAI, trained against BasicAI, or the RulesAI, trained against itself, could be presented as the best performers on the defender side. However, the defender overall seemed to be mostly ending up on the losing side. This may be an indication of some imbalance still present in the game although some of the early experiments showcase that the defender side is capable of winning.

On the concluding note, the game itself was designed with easy extensibility in mind. Therefore, there are many open possibilities for expansion other than just adding AIs. New types of units or buildings can be easily added without many necessary adjustments. This would allow the game to become a more complex and interesting environment for AI testing as new possible strategies would arose. Similarly, new maps could be also introduced. However, this action

would probably require reevaluating of the game settings as the new layout of the map could easily disrupt the balance between attacker and defender. Finally, as it was not the main goal of this thesis to implement a game primarily meant to be played by humans, the user interface is very simple and could be improved.

Bibliography

[1] Thomas Back. *Evolutionary Algorithms in Theory and Practice : Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. First Edition. Oxford University Press, Oxford, 1996. ISBN 978-0195099713.

[2] The Editors of Encyclopaedia Britannica. Age of Empires, 2021. URL `https://www.britannica.com/topic/Age-of-Empires`.

[3] Gayle Cain. *Artificial Neural Networks*. First Edition. Nova Science Publishers, New York, 2016. ISBN 9781634859646.

[4] Giuseppe Ciaburro and Balaji Venkateswaran. *Neural Networks with R*. First Edition. Packt Publishing, Birmingham, 2017. ISBN 9781788397872.

[5] Carlos Coello Coello, Gary B. Lamont, and David A. van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Second Edition. Springer, Boston, 2007. ISBN 978-0-387-36797-2.

[6] A.E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. First Edition. Springer-Verlag, Berlin, 2003. ISBN 978-3-642-07285-7.

[7] Hunter Heidenreich. NEAT: An Awesome Approach to NeuroEvolution, 2019. URL `https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f`.

[8] Thomas Jansen. *Analyzing Evolutionary Algorithms*. First Edition. Springer, Berlin, 2013. ISBN 978-3-642-17339-4.

[9] Richard Kliman. *Encyclopedia of Evolutionary Biology*. First Edition. Elsevier Science Publishing Co Inc, San Diego, 2016. ISBN 978-0-12-800426-5.

[10] Ke Lin Du and M. N. S. Swamy. *Search and Optimization by Metaheuristics*. First Edition. Birkhäuser, Cham, 2016. ISBN 978-3-319-41191-0.

[11] Mark Mancini. 11 Ancient Board Games, 2016. URL `https://www.mentalfloss.com/article/62089/11-ancient-board-games`.

[12] Gabriel Mayers. Artificial neural networks demystified, 2020. URL `https://medium.com/@gabriel.mayers/artificial-neural-networks-demystified-d7cbfb6c6916`.

[13] Santiago Ontanon. The Combinatorial Multi-Armed Bandit Problem and Its Application to Real-Time Strategy Games. *AIIDE*, pages 58–64, 2013.

[14] Evangelos Papacostantis, Andries P. Engelbrecht, and Nelis Franken. Coevolving Probabilistic Game Playing Agents Using Particle Swarm Optimization Algorithms, 2005. URL `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.728.5120&rep=rep1&type=pdf#page=195`.

[15] Elena Popovici, Anthony Bucci, R. Paul Wiegand, and Edwin D. de Jong. Coevolutionary Principles, 2012. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.189.4593&rep=rep1&type=pdf`.

[16] Richard Sheposh. Strategy video game. *Salem Press Encyclopedia*, 2020.

[17] Kenneth Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1, 2019.

[18] Kenneth O. Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *The MIT Press Journals*, 10(2), 2002.

[19] Starborne. A Brief History Of Strategy Games, 2020. URL `https://starborne.com/post/a-brief-history-of-strategy-games`.

[20] Wikipedia: the free encyclopedia. Total War (video game series), 2021. URL `https://en.wikipedia.org/wiki/Total_War_(video_game_series)`.

[21] John V. Urbas. Neural networks. *Salem Press Encyclopedia of Science*, 2020.

List of Figures

List of Tables

A. Attachments

A.1 User Documentation

A.1.1 Installation and requirements

To run the game Windows 10 operating system is required. The game came be started via the Bakalarka.exe present in the Build folder.

A.1.2 Graphical and User Interface

This section will describe the three menus present in the application.

Main Menu

Once the application is launched, the user is greeted by the main menu screen shown in figure A.1. Its structure is very simple. There are three buttons available:

- **Training** - Similar to the previous button, but it shows the **Training Menu** instead.
- **Play** - When this button is clicked, the **Game Menu** described further in the text pops up.
- **End** - This last button simply closes the application.

Figure A.1: Image of the main menu

Training Menu

The training menu serves as a setup menu for the training mode described later in this chapter. Its contents heavily depend on the selected option in the runner

drop-down box in the middle, which can be seen in figure A.2. This element together with the two buttons on the bottom of the screen are always there. The runner selection provides options of different implemented runners for the training mode. Based on the selected item, input fields for parameter specification for the runner show up underneath it. The two available runners will be discussed later in this chapter.

In case the selected runner requires both attacker and defender players to be specified, the two drop-down elements labeled Attacker and Runner appear. They provide selection of all the available implemented AIs. Based on the selected option, additional input fields can appear below as shown in figure A.3. Those represent the parameters of the AI which can be specified for the training. In case any trained AIs of the corresponding type are available, a selection will be shown. The descriptions of the saved trained AIs is given in chapter 5.

Finally, the two buttons have pretty simple functionality. The start training button launches the training session. The menu button returns the user back to the main menu.

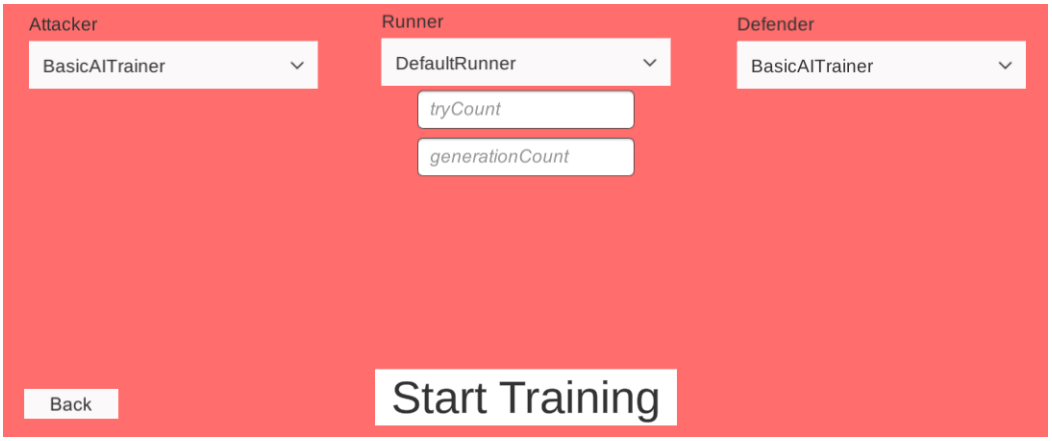

Figure A.2: Image of the training menu 1

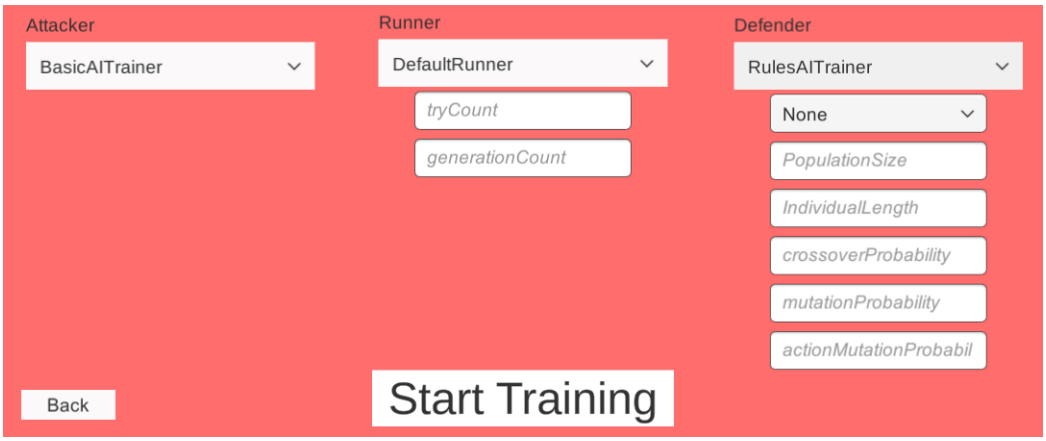

Figure A.3: Image of the training menu 2

Game Menu

The game menu has a similar purpose to the training menu. However, this time it sets up the game mode itself. The game mode is described later in this chapter. Unlike in the previous one, this menu only contains the AI player selection without the need to specify the runner as can be seen in A.4. Additionally, no parameters for the AIs are shown, only the options of available saved players for the trainable versions as shown in fire A.5. Besides that, the game can be played by a human player. There are to checkboxes available to specify whether either attacker or defender are going to be controller by the user. However it is not possible for both to be humans.

There are also two button available, one starts the game and the other takes the user back to the main menu.

Figure A.4: Image of the game setup menu 1

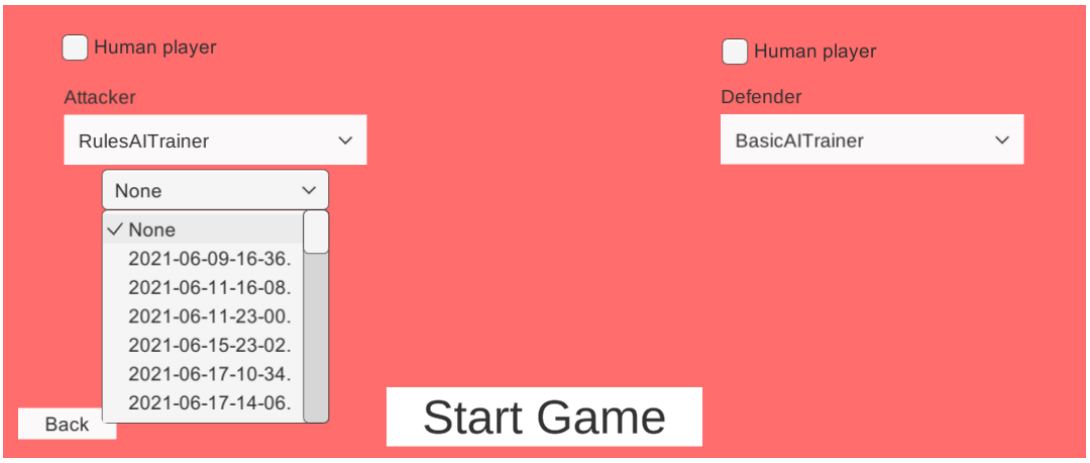

Figure A.5: Image of the game setup menu 2

A.1.3 Gameplay and controls

As already mentioned, the application has two usable modes: training mode and game mode. This section will describe both in detail.

Training Mode

The first mode, called training mode, provides the functionality to train an AI player according to the setup given in the training mode menu. It might also be used for performance comparison in terms of win/loss ratio.

To clarify, the training mode takes the specified AI players and makes them play the game a certain number of times.

As shown in figure A.6, a progress bar showing the testing session progress is displayed along with a check box for enabling an in-game debug console [1].

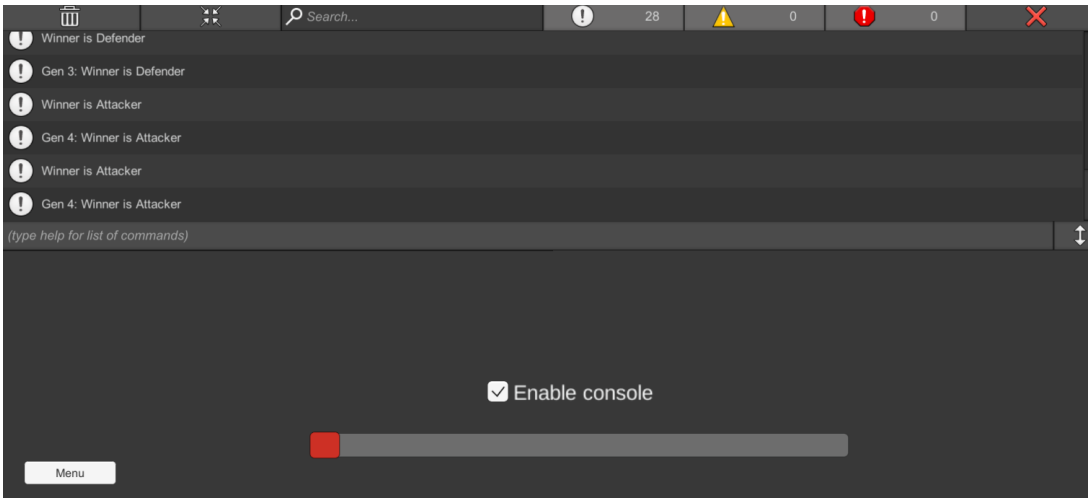

Figure A.6: Image of the training screen with the in-game console

There are two non-trainable AIs, BasicAI and RandomAI, and three trainable ones available for selection. The trainable ones get better throughout the training and at the end of the session their best representative is saved. The saved player can be used further either to train others or as an opponent for the user in the game mode. The trainable versions contain several parameters that are adjustable from the training menu:

- **RulesAI & CoevolutionAI**
  - **PopulationSize** - Specifies the number of individuals in the AIs population (10-50 recommended).
  - **IndividualLength** - Sets the number of rules in an individual (7-20 recommended).
  - **crossoverProbability** - It is the probability of crossover of two individuals from the population (must be 0-1).
  - **mutationProbability** - The probability that an individual will be subjected to mutation (must be 0-1).
  - **actionMutationProbability** - Last is the probability of a single action in the rule being actually modified (must be 0-1).

The available runners are:

[1] Implementation available at https://github.com/yasirkula/UnityIngameDebugConsole

- **DefaultRunner** - This runner launches a single training session with specified parameters.
  - **tryCount** - Specifies how many times is the game played by individuals/players in each generation.
  - **generationCount** - Represents the total count of generations to be performed. One generation means each player in the populations of both AIs has played at least certain number of games, specified by the previous parameter.
- **MultipleTraining** - Unlike the first one, the MultipleTraining runner can be set to perform multiple successive training sessions.
  - **GenCount** - Equivalent to the generationCount parameter in the previous runner.
  - **timesAttackerDefender** - Specifies the number of training sessions to run.
  - **timesDefenderAttacker** - Also represents the number of training sessions to run, however, with the attacker and defender specified players switched. For example, if RulesAI was set as the attacker in the training menu, it is now used as the defender for a new training session with a new population created.

Game Mode

The second mode is the game mode. It takes the players selected in the game menu and if neither one is human, it lets the user watch their match. This might be useful to observe the behaviour of the trained AI players.

Naturally, the game comes also with support for a human player. If the user decides to play the game, first the side must be picked in the menu. There are two options, to play as an attacker or defender. It is not possible to select both to be a human player. The selection of the AI opponent stays the same as already explained in the previous section. Once the choice is made, the game can be started by clicking on the start button.

Once in the game, the user can see the amount of saved money in the left-hand side corner as shown in figure A.7. The available units with their prices are right below and can be purchased by clicking on the chosen unit's button. However, only if enough money is available, new troop consisting of units of selected type will show up in the player's spawn area corresponding to the grey places on the map. The troops, and towers if user picked the defender side, can be easily controlled by the mouse. First, the troop or tower must be selected by pointing at it and pressing the left mouse button. Once selected, it will turn white. Then two actions are available, the user can either click on an empty field to move the troop there or click on an enemy object to attack it. The first action is obviously not available for towers. To make it easier for the human player, the towers and troops have auto-attack on. Once an enemy is in their range, they attack it. To deselect the object, the right mouse button must be pressed.

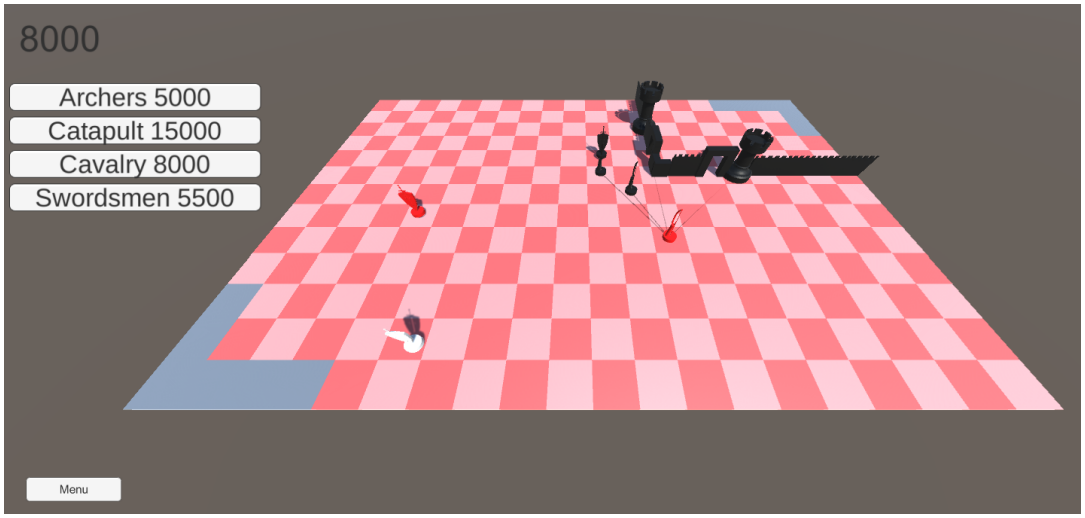

Figure A.7: Image of a game with a human player selected

A.2 Developer Documentation

A.2.1 Prerequisites

In order to open the project, Unity 2020.2.7f1 and Visual Studio 2019 are required along with the Windows 10 operating system.

A.2.2 Structure

The whole implementation can be found in the solution file called Bakalarka.sln. This is a file created by Visual Studio, which organizes the separate projects. The solution consist of two projects:

- **GameFramework** - This project contains the whole game logic as well as the AI framework. It provides the public API for AI implementation.
- **AI** - The second project is where all the AI implementations should be created together with anything else needed for the AI to work.

The main idea behind this two-part structure is to separate the internal game and framework logic from the AI implementation. As said, the GameFramework provides a public API to the AI project with everything needed for the AI creation. It is designed with the intention to ensure all things accessible from the AI project are safe to use and cannot disrupt the internal state of the game.

All the scripts of both projects are stored in separate folders which can be found in the Scripts folder inside the Assets folder. They are further divided into several sub-folder for better orientation. The GameFramework folder consist of the following sub-folders:

- **AIBase** - contains all the base classes for AI implementation
- **Game** - contains the classes with core game logic and game elements implementation
- **Map** - contains several files implementing the map and field classes

- **Pathfinding** - contains a single file with pathfinding implementation for the whole game.
- **UI** - contains UI related implementation files
- **Units** - contains implementation of the unit

The AI folder contains:

- **BasicAI** - contains the player and controller scripts for BasicAI
- **CoevolutionAI** - consist of the CoevolutionAI and CoevolutionAITraner scripts.
- **RulesAI** - contains the player and controller scripts for RulesAI
- **NEAT** - containst the NEAT library[2] along with the NEATAI player and NeatSupervisor
- **CustomRunners** - contains implementation of classes from the Genetic namespace
- **CustomRunners** - contains implemented custom runners used to manage training sessions

A.2.3 Game and Training Logic

This section describes the core game logic, which is divided into three parts, the instance, the loop and the scheduler, each with a specific task. However, first the various interfaces and classes that represent the game elements will be introduced.

Interfaces and abstract classes

There are multiple different interfaces and abstract classes used throughout the game implementation. Most of them serve the purpose of creating a certain structure amongst the game elements and prescribe certain characteristics for objects of distinct types.

- **IMappable** - To start with, the Imappable interface consist of one field called **Position** of Unity's type Vector2Int. Every object must implement this interface in order to be placed on a map.
- **IObject** - The IObject interface, a child of the IMappable interface, is a common interface for all of the objects in the game. It declares only one method called **CanPass**, which should implement the decision making process whether an object of given role can pass through the object.
- **IMovable** - Everything in the game with the ability to move must implement this interface. It inherits from the IObject interface and adds two properties - **ActualPosition** and **Speed**.

[2]UnitySharpNeat - https://github.com/flo-wolf/UnitySharpNEAT

The first property introduces a new kind of position. While the previously mentioned Position represents a spot in the two-dimensional field representing the map, the ActualPosition is a location of the game object in the game. In other words, it is a position of the 3D model inside the game space.

Second property, called Speed, is pretty self explanatory. It is a float value representing the movement speed of the object.

- **IDamageable** - The next interface is called IDamageble. It is essentially meant as a common interface for all objects that can be attacked and therefore damaged. Again, it also derives from IObject. It introduces 5 properties:
  - **Type type** - The first one, called type, should return the System.Type of the object.
  - **int Health** - Second property represents the current health of the object.
  - **Role Side** - Side is an indication whether this object belongs to the attacker, defender or is neutral. Those are the three values available in the Role enum.
  - **int ReceivedDamage** - This represents the amount of already received damage.
  - **State CurrentState** - Last is the CurrentState property. The State enum provides five different states, namely Moving, Fighting, UnderAttack, PreparingForAction and Free.
- **Damageable** - First abstract class to discuss is the Damageable abstract class. It implements the IDamageable interface and extends its functionality. Essentially, the main reason for this approach is to be able to separate public API of the objects available to the AIs and internal methods and properties which must not be accessible outside the game assembly. The class adds two properties - **Visual** and **CurrentInstance**.

  The Visual property represents an instance of a **VisualController** class. It is only used in the game mode when the game is being displayed. Its main purpose is to manage the 3D model represented by a MeshRenderer class provided by Unity.

  The second property, called CurrentInstance, is simply an instance of the game or training in which the objects exists. It is mainly used for the purposes of training as multiple sessions run simultaneously and the object must identify which one it belongs to. Additionally, the class also contains a basic implementation of a virtual method called **TakeDamage**. It received an integer representing the amount of damage the object has suffered and lowers its health by said amount.
- **IAttack** - This one extends the IDamagaeble interface for the purposes of objects that can not only be damaged but are also able to carry out an attack. It consist of several additional properties:

  - **int Damage** - The amount of damage the object can deal to the enemy.
  - **int Range** - Specifies how far the object's attack can reach. As will be later explained, the map is grid based, therefore the range is represented as a whole number.
  - **int DealtDamage** - Total amount of damage dealt to enemies.
  - **int EnemiesKilled** - Number of enemies killed by the object.
  - **int BuildingsDestroyed** - Represents the number of buildings destroyed by the object.
  - **Damageable Target** - Last property contains an instance of the current target, possibly null if none set.

- **Attacker** - In a similar manner to the previous example, the Attacker abstract class expands the IAttack interface. It introduces a single new property, called **Action**, which contains the currently performed action. Several new methods are also present. There are two abstract methods **GiveDamage** and **GetDefenseAgainstMe** which must be implemented in the children and will be described later. **PrepareForAttack**, **StartAction** and **StopAction** are three simple auxiliary methods that manage the state of the object and set Action and Target property. Finally, the virtual **Attack** method implements basic damage dealing logic.

- **IRecruitable** - It inherits from the IDamageable interface and it's the common interface for all the object that be become part of the player's army - Troop, Building and Tower. The mentioned classes are discusses later in this section. For the purposes of training, it includes a declaration of the **GetStats** method, which returns and instance of the **Statistics** struct. It contains several information about the object, namely dealtDamage, receivedDamage, killedEnemies, destroyedBuildings and the UnitType.

- **IAction** - The final mentioned interface represents the common interface for the two available actions to carry out - **MoveAction** and **AttackAction**. Both must implement the declared method called **Execute**.

Map

There are two main places, where a certain variant of the map is needed. First, it is used for internal purposes of the pathfinding algorithm and second, it serves as a representation of the game plan. For this reason a generic base class, called **Map**, was implemented with support for all IMappable objects. It contains a two-dimensional array of the specified type and provides a public indexer for both integer values as well as a Vector2Int value.

For the game plan representation, there is class **IObjectMap** which inherits from the generic map giving a parameter called **Field**. The game plan is essentially a square grid, where the Field class represents a single square. It contains information about whether any object is present on the field and whether it is in the defender's castle. Only one object can occupy the field at a time. Furthermore, the IObjectMap provides methods like GetFreeSpawn, which returns a

position for the troops to spawn, and initialization methods such as **LoadMap** and **ReloadMap**.

Pathfinding

The pathfinding algorithm implemented by the **Pathfinding** static class is a simple version of the A* algorithm. It provides a public method called **FindPath**, which requires the starting and end positions and returns a list of coordinates forming the path. It also requires the role of the object looking for a path and the instance of the game, in order to be able to detect possible obstacles.

Units

Additionally to the previously mentioned interfaces, the are two abstract classes and one interface forming the hierarchy of the units used in the game. The **Unit** interface is a base interface for all the units available in the game. There are essentially two types of units, **HumanUnit** and **TowerUnit**. Those are their base abstract classes. The reason of this distinction is the fact, that human units must have the ability to move while tower units do not. The shared properties present in the classes are:

- **Health**
- **Damage**
- **Range**
- **Price**
- **ReloadRate**

The HumanUnit has three other properties, namely Speed, BundleCount and UnitPrefab. The first one is obvious, the BundleCount represents the amount of individual units that can be bought at a time and the UnitPrefab is a prefab of the **VisualController** which is then displayed. Currently, there are four types of implemeted human units - **Archers**, **Catapult**, **Cavalry** and **Swordsmen**, and one type of tower called **BasicTower**.

For the purpose of setting the parameters of available units from the Unity editor, there exists a generic class called **Setup**. For each unit type a child of the generic class must be created, setting its parameter type to the type of the unit. This class is then placed on a game object in the scene. Even though this might seem unreasonable, it is there to enable Unity to show the desired fields in the editor. For some reason, it does not work well with generic classes. Not to mention, all this is needed, because the units themselves cannot inherit from MonoBehaviour as they can be used throughout multiple threads.

Each type of human unit also implements its own version of the **GiveDamage** method, which takes the enemy instance and total damage to deal. This enables each type to deal damage in its own manner. New unit types should be easy to implement and add to the game. A new class inheriting from one of the abstract classes must be created and the parameters either set directly in the code or via Unity Editor within a child of the Setup class.

Troop

The **Troop** represents one of the three main types of objects available in the game. If certain bundle of units is bought, it essentially forms a troop represented by this class. This generic class, where the type parameter must be a child of the HumanUnit class, is derived from the abstract class **TroopBase**. This abstract class inherits from Attacker and implements the IMovable and IRecruitable interfaces. It provides several methods needed for management of the troop:

- **MoveForAttack(Vector2Int targetPos)** - This method is called from the AttackAction in case the target is out of range. If not already moving, the route to the target is computed by the **RecomputePath** method. Additionally, randomly once in while or if the target field becomes occupied, it is also recomputed. However, first the **FindSpottInRange** is executed to find a position in range of the target to move to. Once the path is obtained, the **Move** method takes care of the movement itself. With every call it moves the troop slightly closer to the target, following the computed path.

- **bool MoveTo(Vector2Int targetPos)** - Similarly to the previous example, it computes the path and call the **Move** method. However, this time the MoveAction is what uses this method. Therefore the position to move to is already given and FindSpotInRange is not used.

- **bool FindSpotInRange(Vector2Int target, out Vector2Int pos)** - As its name suggests, the goal of this method is to find a unoccupied position in range of given target. It essentially looks at the the map's Fields around the target which are close enough.

The generic Troop class adds a couple of method as well. However, most of them are either implementation of a certain interface or overrides of previously defined methods.

- **bool TakeDamage(int damage)** - This method simply lowers the troops health by given amount and checks whether it should be destroyed or not. If it is killed, true is returned to indicate its death.

- **bool TakeDamage(int damage, int index, int countToDamage)** - To start with, certain clarification is needed. The Troop class groups the units together. The damage of the whole troop is computed from the count of units still alive and damage of their type. Therefore, the Troop class internally keeps a list of integers, which represents the health of its units. This overload of the TakeDamage method specifies and index in that lists, where the damage should be applied, and count of how many units are affected. This method is primarily used by certain unit types in order to mimic specific attack type. For example, the catapult usually hits a random chunk of units in the troop.

- **bool GiveDamage(Damageable enemy)** - Last method is called GiveDamage. It first checks whether specified time needed to reload has elapsed and then calls **GiveDamage** method in its unit type to execute the attack. If the target is killed, it evaluates as true.

Building

Another type of objects found in the game is the **Building**. This abstract class implements the IRecruitable interface and the Damageable abstract class. It overrides the provided TakeDamage method implementing additional functionality, which manages the destruction of the building. The **Destroy** method is what takes care of that action. It wipes the object from the map and if the visual is turned on it destroys it.

There are two implemented types of buildings - **Wall** and **Gate**. In this case, the towers are considered a separate type because they are able to attack enemies.

Tower

Similarly to the Troop class, the Tower class is also generic and inherits from a base class called TowerBase. However, this time the parameter type must be a child of the TowerUnit. The base class is there mainly to be able to specify parameters without the need to make the class or method generic. The generic Tower class implements methods such as GiveDamage and TakeDamge with similar functionality to the the ones introduced for the Troop class.

Army

Finally, the class that groups all the IRecruitable objects belonging to a player is called **Army**. It contains three list for each type Troop, Building and Tower. Their read only versions called **Troops**, **Towers** and **Building** are public and part of the API accessible to the AI project, which will be firther discussed in the A.2.4 section.

Several important methods, that play an important role for the core functionality of the game must be named:

- **AddStructure** - This is a method for adding building and towers to the army instance from the map. It utilizes the static **CreateStructure** provided by the **UnitFactory** class. It construct the proper instance based on the given parameters, such as position and Instance.
- **TryByuing** - The TryBuying method takes and index of the unit type to buy and an Instance, which is described later. It first checks whether specified unit is in the army's budget and then proceeds with the purchase. The static **CreateUnit** method in **UnitFactory** returns the properly setup Troop instance.
- **MoneyGain** - Simple method which increases **Money** variable by a specified amount based on whether the army belongs to the attacker or defender.

Actions

There are two action classes that implement the IAction interface - **AttackAction** and **MoveAction**. The main task of these classes is implemented within the **Execute** method.

When instantiating the AttackAction, the attacker and the enemy must be given. The Execute method then first checks whether both are still alive. If they

are, the distance between them is computed and checked against the attacker's range. If the enemy is out of range and movement is possible, the MoveForAttack method is called. If the target is in range PrepareForAttack is executed followed by the Attack method. If enemy is killed, the action stops.

For the MoveAction class construction, a Troop and its target position must be specified. Similarly to the preceding action, the Execute method starts with a check of the troops heath. If alive, it simply executes the MoveTo method provided by the Troop class.

Instance

First, there is the abstract **Instance** class. It serves as a base class representing a general instance of the game. It contains information about both players, sets up the start of the game and provides a method, which executes a single loop of the game. Overview of the most essential methods is as follows:

- **SetPlayers** - This method manages the proper setup of both players. It initializes their armies, gives them information about the game and sets their side. Then it proceeds to reload the map and setup the scheduler.
- **GetArmy** - The GetArmy, and the GetEnemyArmy variant, are methods which return the proper **Army** instance for given side.
- **OneLoop** - OneLoop is a very essential method which provides way for the children classes to execute one loop of the game. Most of the functionality utilizes the scheduler and its methods, which are described further in this section. First, it checks whether its time to give the player money. If it is the Scheduler's **MoneyGain** method is called. Then the **Shopping** method followed by the **ScheduleActions**, both provided by the scheduler, are executed. In other words the players are prompted to pick a unit type for purchase and then select an action to do. Final part tells the scheduler to run the action which is the job of the **RunActions** method.

The instance class is there to provide common base for instances run in the game mode and also the ones run in the training mode. Respectively, the **GameInstance** and the **TrainingInstance** are classes representing and managing the whole game in given modes.

To begin with, the GameInstance class is slightly less complicated. It provides a starting method called **Run**, which simply initiates the game. Once launched, the Unity provided method named **FixedUpdate**, called in short preset intervals, executes the RunGameStep method. Each time RunGameStep is called, the termination criteria is checked and if not fulfilled, the OneLoop method from the base is run. In case it is met, meaning one player has lost all troops, the game ends.

On the other hand, TrainingInstance not only manages the game loop but also collects data about the game and sends them to AI players. Unlike the previous example, it does not use FixedUpdate. This is due to the fact TrainingInstance is not run on the main thread therefore cannot utilize any Unity provided methods. Instead, it is replaced with a simple while loop in the **RunGameTrainingLoop** method which additionally to the troop counts also checks the number of loops

already executed. If certain amount of time has elapsed and the game is still in progress, it is stopped and declared as a tie. This is a simple precaution measure for situations when the AIs are unable to outperform each other. Such scenario could potentially lead to an infinite game which would be a considerable problem. At the end of the game, the data from both players are collected and stored inside a struct called **GameStats**. The GameStats struct contains information about the winner, number of loops left before reaching the limit and finally collected statistics for both armies. The last one is represented by a list of structs named Statistics, where each corresponds to data collected from a single IRecruitable. The IRecruitable interface will be described later in this section. Each struct stores dealt damage, received damage, number of killed enemies, number of destroyed buildings and type of the unit in the troop.

Loop

The second part of the core logic is the abstract **Loop** class. It contains only one function called LoadMap, which transforms the map game object created in Unity to a list of lists to be later saved into a 2D array. This class serves as a base class for the **GameLoop** and **TrainingLoop** classes. Similarly to the previously mentioned classes, they are used in different modes. First, the GameLoop's main purpose is to instantiate the game instance, set it up properly, load AI players if needed and then run the game.

On the other hand, the TrainingLoop must take care of lot more. Besides the process of initializing a instantiating, it additionally handles all the logic managing the proper flow of training. Once everything, such as the map and players, is set up, a new thread is created where the **PerformOneGeneration** method is run. As its name suggests, it controls the run of a single generation. The process works as follows, first the populations of both AIs are retrieved, then based on the set number of threads to use, multiple games are started. Once every individual from both populations has finished at least one game, the generation is done. However, in order to obtain more accurate results, this process may be repeated multiple times. It depends on a parameter called **tryCount** that can be set when starting the TrainingLoop. For example, this might be utilized to let the AI player collect multiple fitness values and then take the median or average as the final fitness value. To point out, it is possible an individual might be playing more games at a time. For this reason, the **Clone** method, which will be further discussed in A.2.4, must be properly implemented.

Meanwhile, the Unity provided Update method, which runs every frame, checks if a generation is still in progress. Once it finds out it has finished, the AIPlayer methods **GenerationDone** and **BeforeEachGeneration**, described in section A.2.4, are called and a new generation is launched. However, if it was the last generation, **Save** and **TrainingFinished** from AIPlayer are executed instead of BeforeEachGeneration.

Scheduler

The final part of the core of the game is the Scheduler class. It administers four main operations - it schedules action received from the players, executes scheduled actions, manages purchases and controls the money gain. This can be described

by four public methods that are called from the OneLoop described earlier in this section. To point out, the Scheduler class is labeled as internal, therefore none of these methods are available from the separate AI project as it is a different assembly.

- **ScheduleActions** - Once the ScheduleActions method is called, it executes the **Schedule** method for both attacker and defender. The Schedule method simply asks the given method to provide an action to carry out and information about the object trying to do so. This is done by using the player's **GetActions** method. Once it is obtained, it is either added to the queue of running actions or thrown away based on the contents. For example when either the actor or the action are null, it is thrown away. Another possible scenario is the object in question is already doing a different action, in that case its action is swapped for the new one.
- **RunActions** - The RunActions method goes through the queue of running actions and call their **Execute** method. If evaluated as true, the action is queued up again as it has not been finished yet.
- **Shopping** - The goal of this method is to obtain the index of a unit type for purchase from the player. For this purpose, the player must implement the **PickToBuy** method. The result is then provided to the **TryBuying** method of the appropriate Army instance.
- **MoneyGain** - MoneyGain is a simple method which for both players executes a method of the same name on the corresponding Army instance.

A.2.4 AI Framework

The main goal of this section is to describe the structure of the AI framework and the available classes. It extends the information provided in the chapter 4 of the main text.

API

In order to be able to implement a usable AI, some information about the game must be obtainable. As will be shown later, the AI has access only to certain class instances provided by the **AIPlayer** base class. This section lists several classes, which were described in section A.2.3, with everything accessible to the intelligence.

- **Army** The Army class provides several publicly accessible methods and properties including an indexer and enumarator. All the properties from the API are:
  - **Side Role**
  - **int Count**
  - **int Money**
  - **int MoneySpent**
  - **int MoneyAll**

  - **ReadOnlyCollection<TroopBase> Troops**
  - **ReadOnlyCollection<TowerBase> Towers**
  - **ReadOnlyCollection<Building> Buildings**

  All the method publicly accessible are:

  - **SenseLowestHealth()** - Returns an instance of IRecruitable with the lowest health currently in the given army, where IRecruitable is a common interface for all buildings, towers and troops.
  - **SenseHighestHealth()** - Returns the troop with the highest damage in the army.
  - **SenseTroopLowestDamage()** - Returns the troop with the lowest damage in the army.
  - **SeneseTroopHighestDamage()** - Returns the troop with the highest damage.
  - **SenseTroopLowestSpeed()** - Returns the troop with the lowest speed.
  - **SenseTroopHighestSpeed()** - Returns the troop with the highest speed.
  - **SenseLowestDamageDecrease(Attacker attacker)** - Based on the type of the given parameter, it checks for which IRecruitable in the army the attacker has the highest damage modifier. In other words, it looks for an IRecruitable which will receive the most damage from the attacker. The Attacker class is a common base class for all the objects that have the ability to attack, such as towers and troops.
  - **SenseClosestTo(Attacker attacker)** - Returns the closest IRecruitable in the army to the given object in parameter.
  - **List<Statistics> GetAllStats()** - It collects data about all towers, buildings and troops, even the dead and destroyed one, and returns it in a list.

- **Building** - This class inherits from the Damageable abstract class and implements the IRecruitable interface. Naturally the properties only provide public getters.

  - **Role Side**
  - **State CurrentState**
  - **int Health**
  - **int ReceivedDamage**
  - **Type type**
  - **bool CanPass(Role role)** - Checks whether object of given role can pass through this object. Usually false, except for Gate.
  - **Statistics GetStats()** - Returns the statistical informtion about the object.

- **TowerBase** - TowerBase also inherits from the Attacker abstract class. Therefore, one method and several additional properties are available:
  - **Role Side**
  - **State CurrentState**
  - **int Health**
  - **int Damage**
  - **int ReloadRate**
  - **int Range**
  - **int DealtDamage**
  - **int ReceivedDamage**
  - **Damageable Target**
  - **int EnemiesKilled**
  - **int BuildingsDestroyed**
  - **bool CanPass(Role role)** - Checks whether object of given role can pass through this object. Usually false, except for Gate.
  - **float GetDefenseAgainstMe(Damageable enemy)**
  - **Statistics GetStats()** - Returns the statistical information about the object.
- **TroopBase** - The TroopBase provides information about the troop represented by the instance. It implements two interfaces - IMovable and IRecruitable and inherits from the Attacker class. That means all their properties and methods are accessible to the AI. There are two new properties accessible on top of already listed for TowerBase.
  - **int Count**
  - **Vector2 ActualPosition**
- **IObjectMap** - Finally, the last class provides a public indexer and a list called **CastleArea**, which contains coordinates for all Fields that are inside the castle.

UnitFinder

In order to provide an easy way for the AI to pick a unit to buy, the static UnitFinder class provides a readonly list with all the available types and their characteristics. It utilizes reflection to find all the Setup scripts containing parameters for the unit types and saves the information for further use. This approach makes it possible for newly implemented unit types to be automatically registered for the purpose of this class. Additionally, a simple method called PickOnBudget is provided which simply returns an index for a unit type within the specified budget. If none are in budget, -1 is returned resulting in no purchase.

AIPlayer

The AI implementation consist of two scripts. One takes care of the training process and the other represents the AI player and implements the decision making for the game itself. Every AI must derive from the AIPlayer base class. It holds information about the game state for the AI to inspect and manages the requests from the game in terms of action selection. AIPlayer itself derives from IPlayer, which is the common class for both human and non-human player. These are the properties in the base classes:

- **Role Side** - Indicates the side of the player, either attacker or defender.
- **GameInfo Info** - Consist of 3 properties, OwnArmy, EnemyArmy and Map. Both the Map class and Army class were described in the previous section A.2.3.

The base classes specifies some abstract methods needed to be implemented in children:

- **int PickTobuy** - This method must return an index of the unit type that the AI wants to buy. In case no purchase is desired -1 can be returned. The unit types are specified by the UnitFinder class described earlier in this section. To mention, returning an index that cannot be bought due to lack of money is not a problem. Internal check inside the Army class is active.
- **IAction FindAction(Attacker attacker)** - This action must return an instance of IAction representing the desired action for the attacker object to carry out. The IAction instance can be obtained from one of the static methods present in the MacroAction class. These represent the possible action to be taken and will be further described later in this section.
- **AIPlayer Clone()** - A clone of the player must be returned by this copy. The Player can be used on multiple threads simultaneously therefore everything representing a reference type must be deep copied unless suitable for concurrent environment.
- **void RunOver(GameStats stats)** - Once the training finished, this method is executed supplying the AI with statistical information about the game.

To point out, the first three mentioned methods and any other method executed within them **MUST NOT** utilize any Unity provided functionality as they are not run from the main thread. Vector being an exception to the rule.

As mentioned, the MacroActions class contains different methods properly setting up the IAction instances to be returned in FindAction. The typical method signature is as follows - bool DoSomething(Attacker attacker, out IAction result). The attacker instance of the object must be supplied and the IAction is returned as an out parameter. In case the method evaluates as false, null is returned as the action was found to be impossible. The AttackGiven method additionally requires an instance of the selected target.

- **AttackClosest** - Finds the closest enemy or enemy structure and prepares the attack action.

- **AttackInRange** - Similar to previous method, however, it only looks for enemies and enemy object in range of the given troop. If none found, the action is labeled impossible therefore the resultAction parameter is set to null and false is returned.
- **AttackWithLowestHealth** - Looks for an IRecruitable with the lowest health to attack.
- **AttackWithLowestDamage** - An IRecruitable with lowest damage is searched for to be attacked.
- **AttackWeakestAgainstMe** - Picks an enemy troop which has the highest damage modifier, therefore will receive the most damage, against given attacker troop.
- **AttackGiven** - Tries to create an attack on given enemy.
- **MoveToSafety** - This method differentiates between the two sides. The defender move back onto a field inside the castle. On the other hand, the attacker walks back to one of his spawn points.
- **DoNothing** - Equivalent to returning null in PickAction method.

Additionally, serializable versions of those methods are available in the SerializableMacroActions class as its subclasses. They are realized in the form of separate classes implementing the IMacroActions interface. It provides the TryAction method which essentially calls the method mentioned previously. This is mainly done for the purpose of the load/save system. AIPlayer also itself implements the **GetActions** method called from the Scheduler class described in section A.2.3. Every call, it takes one Attacker instance from the army and executes the FindAction method with this instance as the parameter. Then it returns a Tuple consisting of the Attacker instance and obtained IAction instance.

AIController and AITrainer

The second part of the AI implementation must implement a child of one of the two abstract classes: AIController or AITrainer. Both derive from the IPlayerController that is also a base class for the HumanPlayerController. AIController base class is recommended in case of an non-trainable AI. Trainable AIs should implement a child of the AITrainer.

First, here are the properties present in the AIController class:

- **Type AIPlayerType** - Specifies the type of the player, that is controlled by this class.
- **ReadOnlyCollection<AIPlayer> Population** - Represents a read only list composed of instances of players of a certain type. For a non-trainable AI, it only contains a single player.

Second, there is only one method that requires to be implemented in children:

- **IPlayer GetPlayer()** - This method serves the purpose of retrieving a certain player for the game outside of training mode. For example, it can return a strictly specified one or a random one from the population.

To point out, all the fields in children representing integers and floats labeled as public will be registered by the implementation and will be displayed in the training menu as adjustable parameters.

There are also 3 virtual methods that can be overridden to gain additional functionality:

- **void CustomStart()** - This method is called once at the beginning of the training process.
- **void BeforeEachGeneration()** - At the start of each generation, this method is executed.
- **void TrainingFinished()** - Called after the training has finished.

The AITrainer derives from the AIController and extends its functionality for the purpose of training. There is only one added field called population. It is an internal representation of the previously mentioned property Population, to allow children to change the content of the list. Additional abstract method required to be overridden are present:

- **List<AIPlayer>CreatePopulation()** - This method should implement the creation of the AI players and return them in a list.
- **void GenerationDone()** - This method is called after every generation.
- **AIPlayer GetChampion()** - This method is primarily meant for retrieving a single instance of the best AIPlayer, however, as a matter of fact, any instance of AIPlayer can be returned.

AITrainer also manages loading and saving of trained AIs. Working implementation of load/save system is already present, however, these methods can be overridden if needed:

- **void SaveChampion(string file)** - Saves the champion, the return value of GetChampion(), to said file.
- **IPlayer LoadChampion(string file)** - Loads a player from the given file.

The present load/save system utilizes the DataContractSerializer class as it does not require parameterless constructors and can serialize private fields if properly marked with [DataMember] attribute. The class itself must be given the [DataContract] attribute. The serializer saves the player obtained from the GetChampion method to a file which can be found in a folder with name corresponding to the name of the trainer class placed in the build folder. The file itself specifies the name of the player class together with the save time.

TrainingRunner

In case a special approach for the training session is required, a child of the TrainingRunner class can be implemented in order to achieve the desired behaviour. It provides the StartTraining method which takes the specified controllers and runs them for a given number of generations witch each performing specified number of tries. Files containing saves of the corresponding AI players can be also specified. However, it is up to the programmer to ensure such file actually exists. Additionaly, OnStart and OnUpdate methods can be overridden from derived class as an replacement for Unity's Start and Update. In order to check whether a training session is already in progres, the base class provides a TrainingInProgress property.

There are two runners implemented for the application: DefaultRunner and MultipleRuns runner. The first does nothing special, it runs the training session once with the given AI types for a specified number of generations where each get a specified number of tries. The MultipleRuns is similar, but it adds the possibility to specify a greater number of training sessions and also number of training sessions where the AIs will play with switched sides.

In order for the parameters of the runner to appear in the training setup menu a special attribute is specified. Each field marked with the [ShowInMenu] attribute will be modifiable from the menu. However, only integers, floats and exactly two AIControllers/AITrainers are supported. Other types unfortunately will not show up in the menu.

A.2.5 AI Implementations

There are several AIs implemented for the game. The first two, BasicAI and Random AI, are non-trainable. The other three are trainable.

BasicAI

This AI consist of two classes, the BasicAI deriving from the AIPlayer and the BasicAIController inheriting from AIController. The Clone method simply just returns new instance of the class. In the FindAction method a slightly more complex decision making process is implemented. Detailed comments describing the process can be found in the BasicAI.cs file.

The BasicAI controller implements the GetPlayer method which simply returns a new instance of the BasicAI player.

RandomAI

The RandomAI consist of the RandomAI class and the RandomAIController class. Both are very simple. The RandomAI class implements all the required classes and essentially randomly selects what to return. The RandomAIController does the same thing as in the previous example.

RulesAI

The RulesAI is implemented by the RulesAI class and RulesAITrainer class. The RulesAI class derives from the AIPlayer. The RulesAI player contains more data

that the previously mentioned examples, notably:

- **StrategyIndividual individual** - This is an instance of an individual that manages the action selection process. The class will be discussed later in this section.
- **ShopperIndividual shopper** - It represents an individual managing the selection of units to be purchased. Each individual is a simple sequence of indexes, representing the unit types to be bought.
- **IMacroAction[] possibleActions** - This is an array containing all the possible actions that can be execute.
- **ConcurrentQueue<GameStats> accumulatedResults** - In order to be able to gather results from all clones of the player, a queue usable from multiple threads is present. It is shared between all the instances of the class and is not copied in the clone method.

The StrategyIndividual class plays an important role in the RulesAI implementation. Internally, it consist of a list of Rule instances where the Rule class consist of the following fields and properties:

- **readonly ICondition[] conditions** - This array represents conditions that are evaluated in the action selection process. The ICondition interface will be described later in this section.
- **int ActionIndex** - Each rule contains an index of a certain action which is to be carried out.
- **int ActionCount** - Action count represent the number of total possible actions that can be selected for the action index.

As mentioned, the rules forming a strategy individual consist of multiple conditions. Those are represented by the classes implementing the ICondition interface. It requires the Evaluate method, which takes an Attacker instance and GameInfo. Again, this approach is used to enable serialization. There are multiple condition available:

- **Damaged** - Evaluates as true if given object is damaged.
- **Free** - The state of the given object must be set to free for this condition to be true.
- **HealthierThanClosest** - If the closest IRecruitable object has lower health, this condition is satisfied.
- **ClosestIsTroopBase** - Evaluates as true if the closest IRecruitable is of type Troop.
- **ClosestIsBuilding** - Similar to previous condition but the closest must be a building.
- **ClosestIsTower** - Again, similar but tower must be the closest.

- **IsDefender** - This condition is satisfied if the given object belongs to the side of the defender.
- **IsAlone** - True if the side of the given object currently has less than two troops.
- **IsWinning** - It is true if the side corresponding to the side of the given object has more troops than the enemy.
- **IsInsideCastle** - The given object must be inside the castle for this condition to be true.
- **IsInTowerRange** - Satisfied, if the given object is in range of a tower therefore can be hit by it.

The FindAction method in the RulesAI player relies heavily on the functionality of the StrategyIndividual class. For each Rule instance in the individual all its conditions are evaluated. If all true, the action corresponding to the ActionIndex in the rules gets a vote. At the end the action with the most votes is carried out.

The RulesAI method also provides a method used in the trainer for fitness evaluation. The method, called EvaluateFitness, basically goes through the collected data in accumulatedResults and accordingly sets the fitness of the ShopperIndividual and StrategyIndividual.

Second part of the AI implementation, the RulesAITrainer class, derives from the AITrainer abstract class. There are two important fields in the class:

- **Strategy all** - This represents a population of the StrategyIndividual individuals that takes care of the evolution process and applies selection, crossover and mutation. The Strategy class will be further described later in this section.
- **ShopperPopulation shoppers** - Similarly, this is a representation of the population consisting of instances of the ShopperIndividual class. These simply represent a sequence of indexes which is followed when purchasing units. The ShopperPopulation takes care of the evolution process using one-point crossover and mutation of a random index.

The three important method implementations are:

- **List<AIPlayer> CreatePopulation()** - All the used conditions for Rule creation are specified along with the IMacroActions. The the initial population of StategyIndividuals and ShopperIndividuals is created.
- **void GenerationDone()** - This method is called after every generation and manages the evaluation of the data collected in the player. Additionally, it launches the evolution process on both present populations. After that, the evolved individuals from are again mapped onto new RulesAI players.
- **void TrainingFinished()** - This method is called one the whole training session is done. It takes all the saved data for analysis and writes it to two separate files. One contains only the best fitness values from each generation, while the other additional data such as damage, winner etc.

CoevolutionAI

This AI consist of the CoevolutioAI class and CovelutionAITrainer class. The implementation vastly similar to the previously described AI. However, in this case there are multiple strategy populations present in the trainer and multiple strategy individuals in the player representing certain categories of the unit types. Each one population evolves separately.

The decision making process inside the PickAction method depends on the type of the given attacker. Its category is evaluated and based on the result corresponding individual is used to pick the action.

NEAT AI

The last implemented AI is the the NEAT AI. The implementation of the NEAT itself is a public version of the UnitySharpNEAT library, a port of SharpNEAT for Unity [3]. It is used under the MIT license. The code and logic of the UnitySharpNEAT will not be discussed.

Additional base class was created for the NEAT AI player called INeatPlayer.It essentially replaces the UnitController base class described in the library description. It contains a field of IBlackBox, representing the neural net. It is utilized in override of the FindAction method, which basically consist of three parts:

- **UpdateBlackBoxInputs** - This method is required to be implemented in the children. It must set the input signals for the neural network. An input array is provided in the function parameter. For that, the NEAT AI player utilizes the conditions presented in previous sections. Each condition is evaluated and the result is given as input to the presented array.

- **blackBox.Activate** - This activates the IBlackBox which calculates the outputs.

- **UseBlackBoxOutpts** - This essentially replaces the FindAction method as a place, where to do the action selection process. It must be implemented in the children instead of the FindAction method. The NEAT AI looks at the values of given outputs and takes index corresponding to the one with the highest value. The index corresponds to the action which should be carried out.

Additionally, the NEAT AI player must implement the GetFitness method which returns the players fitness value.

In the NEATAITrainer class, first the Experiment class must be created. This class creates the evolution algorithm, which takes care of the evolution process. It is represented by the NeatEvolutionAlgorithm class and incorporates speciation and offsrping creation.

Each player instance is assigned a single IBlackBox. After each generation, the population is evaluated and modified by the NEAT library. Additionally, this is the only AI implementation using a custom load/save system. The methods from the base class are overridden and system provided by the library is utilized.

[3]UnityNEAT - https://github.com/flo-wolf/UnitySharpNEAT

Genetic namespace

The Genetic namespace contains important auxiliary classes and methods utilized by the AI implementations. From already mentioned classes, it contains all the described conditions along with the Rule class forming the StrategyIndividual.

The Operators.cs file defines several genetic operators belonging to the Genetic namespace along with the fitness function used throughout the code. Namely the EvolutionFunctions class implements static methods TournamentSelection, RouletteWheelSelection, Crossover, Mutation and ComputeFitness. The first four mentioned methods are generic and can be used on any class implementing the Individual interface. This interface requires an implementation of a Clone method and Fitness property. Both the ShopperIndividual and StrategyIndividual implement this interface. The crossover and mutation methods do not perform the modification itself but require a specific delegate with the specific crossover and mutation functionality. Essentially, these method go through the population and with specified probability call the given delegate.

Final part of the Genetic namespace are the StrategyIndividual and Strategy classes. The Strategy class represent the whole StrategyIndividual population and takes care of the evolution process. It implements the Evolve method, which is called from the trainer, and utilizes the RouletteWheelSelection, Crossover and Mutation methods. For the latter two it specifies the methods to be used. Uniform crossover and mutation of the action index in the rules are used in this case. As already mentioned, the StrategyIndividual itself consist of a list of Rule instances.

A.2.6 Unity Scenes and UI

The whole application consists of three separate Unity scenes - menu, training and visual. This section will describe these scenes and the UI elements used.

Menu

First, the menu scene is the first thing to appear once the program is started and displays the main menu of the game.

- **UIController** - This object contains three simple scripts AIOptions, Buttons and RunnerOptions. AIOptions groups together all the implemented AIs. If a new one is implemented, its trainer or controller script must be added to the element list in this object. The Buttons script manages the end button present in the scene. Finally, the RunnersOption is similar to the AIOptions script. In case new runners are implemented, they must be added to the element displayed list in the editor.

- **GameSetting** - Once the game mode is run, the Settings script placed in this gameobject loads the selected setting from the game setup menu and delivers them to the launched game instance.

- **Canvas** - This object groups together all the UI elements visible in the scene. Additionally, it also contains the game menu and training menu. Those are displayed if the corresponding buttons are pressed. To point

out, both setup scene contain several drop-down elements. Those contain a special script, TrainingSettings for the attacker and defender dropdown and RunnerSettings for the runner selection. These scripts analyze the selected option and via reflection look for parameters that require an input field to be displayed for modifications.

Training

The Training represents the scene being displayed if a training session is initiated. It displays a simple progress bar along with an in-game console which can be turned on. There are multiple game objects forming the scene:

- **UnitSetup** - This objects contains all the Setup scripts for available unit types. Their public parameters are displayed in the editor and can be easily edited without changing the code. However, they should not be modified in the scene but on the UnitSetup prefab placed in the Prefab folder. That way the changes will be applied everywhere where it is used.
- **TrainingRunner** - This object is disabled as it is only used for debugging purposes. It contains different version of runners which where tested.
- **UIController** - The UIController contains the DebugControllerScript, which manages the in-game debug console, the TrainingProgressBar controlling the progress bar and finally the Buttons script which provides the menu button functionality.
- **UnitFinder** - It contains only a single script called UnitFinder, which basically registers all the implemented unit types as explained in the previous section.
- **InGameDebugConsole** - This objects contains a script handling the debug console present in the training scene. The implementation of this feature was done by Süleyman Yasir KULA[4] and is used under the MIT license.
- **UI** - UI contains three child objects - Toggle, Slider and Menu, which represent the UI elements visible on the screen.
- **Starter** - This game object contains a simple script which looks for object with instance of a TrainingRunner and starts it.
- **Main Camera and EventSystem** - These are objects created by Unity. The Camera manages the scene view and EventSystem is required if any UI elements are present.

Visual

This scene represents the game visualization, which is displayed once the game mode is launched.

- **GameMaster** - The GameMaster object contains the GameLoop script described in section along with the already mentioned Buttons script. A.2.3

[4]In-game debug console package - https://github.com/yasirkula/UnityIngameDebugConsole

- **UnitSetup** - Equivalent to the UnitSetup mentioned in the previous scene describing the Training scene.
- **Map** - This game object represents the map present in the scene. It consists of individual child game object forming the whole map square by square.
- **UnitFinder** - Already described in the previous section.
- **Main Camera & Directional Light** - Unity provided objects handling the scene and light rendering.