FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Implementation of 2D turn-based strategy game with AI
Student:	Ivan Štěpánek
Supervisor:	Ing. Eliška Šestáková
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

The goal of this bachelor thesis is to create a prototype of 2D turn-based strategy game that is inspired by the game Advance Wars. In single-player mode, the strategy of the computer will be an AI that was trained to play through self-play, using a minimum of human input.

Requirements:
- It should be implemented as a desktop application using the C++ language.
- The play must support a mode for several players and playing against artificial intelligence (AI).
- Provide at least 4 various pre-created game levels.

Research part:
- The AI component in the requirements above constitutes the research part.
- Evaluate different algorithms for developing the AI strategy.
- Design a strategy that relies mostly on self-play and uses a minimum of human input.
- Implement the strategy within the game.
- Evaluate the quality of the game played by the AI by evaluating its success rate against players of varying skills.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 9, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Implementation of 2D Turn-based Strategy Game with Artificial Intelligence

Ivan Štěpánek

Department of Software Engineering
Supervisor: Ing. Eliška Šestáková

January 9, 2020

Acknowledgements

Firstly I would like to thank my former supervisor Juan Pablo Maldonado Lopez for accepting me and my work when I was struggling to find a supervisor. Secondly, I would like to thank my present supervisor Ing. Eliška Šestáková for helping me with writing. Both of them are very kind and eager to help. My thanks go to my former classmates as well. Namely Minh Trieu, Marek Kodr, Jan Fara, Adam Jirovský, and Lukáš Renc for encouraging me to finish my work. They and many others were also a great help during my whole studies. Also, I am grateful to my parents for allowing me to concentrate on studying without much worries about money and living. Last but not least, I would like to thank my girlfriend for her patience and help with the English grammar.

Declaration

Citation of this thesis

Štěpánek, Ivan. *Implementation of 2D Turn-based Strategy Game with Artificial Intelligence*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Práce se zabývá tvorbou strategické hry a umělé inteligence, která se ji naučí hrát. Je prozkoumán žánr strategických her, ale i důležité části pro umělou inteligenci. Dále práce analyzuje několik profesionálně vytvořených her a programů hrající tahové strategie. Závěrem hodnotí kvalitu jak implementované umělé inteligence, tak implementované hry a navrhuje možná zlepšení.

Klíčová slova umělá inteligence, strategie, genetický algoritmus, neuronové sítě

Abstract

The thesis deals with creating a strategy game and artificial intelligence learning the game via self-play. It looks into a strategy game genre as well as important components of artificial intelligence. It analyzes several examples of real-world games and programs playing strategy games. In the end, the thesis discusses the quality of implemented artificial intelligence but also the game and suggests possible improvements.

Keywords artificial intelligence, strategy, genetic algorithm, neural network

Contents

List of Figures

List of Algorithms

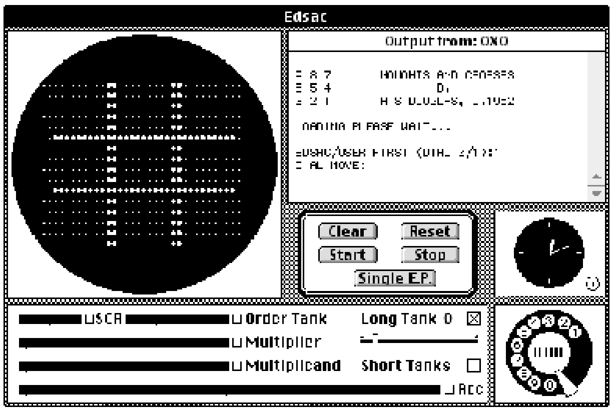
List of Tables

Introduction

The significance of videogames is growing considerably every day. According to the ESA (Entertainment Software Association) survey from 2019, every third household in the USA has at least one videogame player. Moreover, in 2018 the consumers spent 43.4 billion dollars on videogames [1]. Continual progress in electronics enables us to create more complex and remarkable games. Nowadays, playing videogames is not even bound to desktops and consoles anymore, as they are performed on devices such as smartphones, tablets, and virtual reality headsets as well. There are several milestones in the early stages of videogame development. One of them is the game OXO. The very first game was created by student Alexander S. Douglass during his dissertation work in 1952 [2].

Videogames took a big leap in their graphic adaptation since the OXO of A. S. Douglass as can be seen in Figure 0.1. Nowadays, with each upgrade, videogames look more realistic. However, an impression from virtual reality can be disrupted by abnormal and absurd behavior of a game world, especially if artificial intelligence is controlling NPC (Non-player Character).

Development can be a big challenge for video game developers, especially in genres such as strategy and RPG, where players interact with their surroundings on a large scale. For example, there is a known error in RPG game *The Elder Scrolls V: Skyrim*. If you put a bucket on NPC's head, you can rob it without it noticing. Some people made fun of the error as illustrated by Figure 0.2. Hence the development of artificial intelligence should be considered as necessary as the development of graphic adaptation in the game industry for some genres.



```
= 8 7        NOUGHTS AND CROSSES
= 5 4              D.
= 2 1        H S DOUGLAS, 1952

 LOADING PLEASE WAIT...

EDSAC/USER FIRST (0/1):
= AL MOVE:
```

Figure 0.1: Screenshot of EDSAC simulator with OXO running on Mac OS [1]

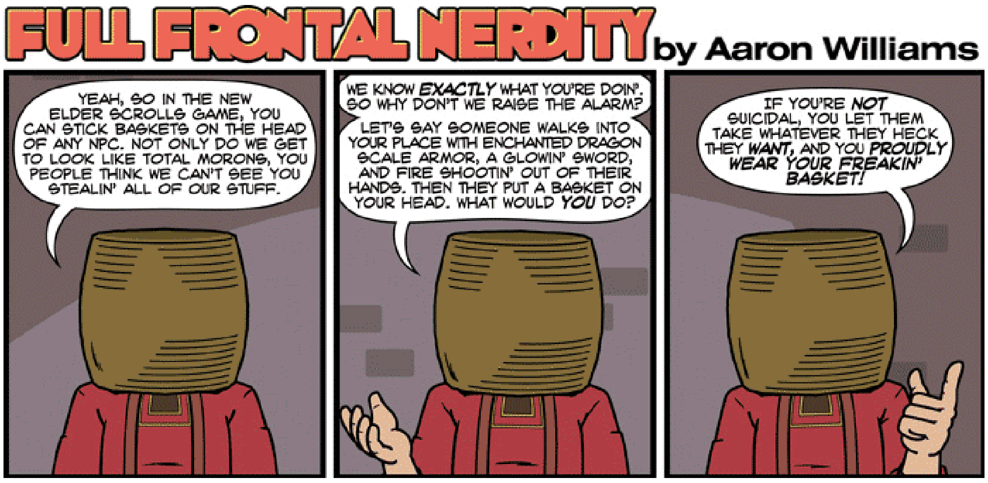

FULL FRONTAL NERDITY by Aaron Williams
YEAH, SO IN THE NEW ELDER SCROLLS GAME, YOU CAN STICK BASKETS ON THE HEAD OF ANY NPC. NOT ONLY DO WE GET TO LOOK LIKE TOTAL MORONS, YOU PEOPLE THINK WE CAN'T SEE YOU STEALIN' ALL OF OUR STUFF.
WE KNOW EXACTLY WHAT YOU'RE DOIN'. SO WHY DON'T WE RAISE THE ALARM?
LET'S SAY SOMEONE WALKS INTO YOUR PLACE WITH ENCHANTED DRAGON SCALE ARMOR, A GLOWIN' SWORD, AND FIRE SHOOTIN' OUT OF THEIR HANDS. THEN THEY PUT A BASKET ON YOUR HEAD. WHAT WOULD YOU DO?
IF YOU'RE NOT SUICIDAL, YOU LET THEM TAKE WHATEVER THEY HECK THEY WANT, AND YOU PROUDLY WEAR YOUR FREAKIN' BASKET!

Figure 0.2: Comics making fun of *The Elder Scrolls V* [2]

Thesis Goal

The thesis aims to implement a prototype of a 2D turn-based strategy game. However, it does not suggest to create a full-fledged game competing with other game titles often made by an entire studio. The thesis implements a program learning to play the game via self-play. Furthermore, it discusses some design patterns and algorithms that are useful not only for the implementation of artificial intelligence but also for the implementation of the 2D game. Research will be conducted by studying samples of actual strategies. The thesis also examines some programs playing turn-based games. In detail, it focuses on an algorithm called *Blondie24* that was capable of learning checkers with minimal human input.

Thesis Structure

The thesis is organized in a way that does not require a deep knowledge of games or artificial intelligence. The text tends to go from simpler to more complex ideas. Though it still assumes some technical knowledge from the reader since the thesis aims to program a game and artificial intelligence. Chapter 1 describes several concepts needed for the implementation of the game as well as artificial intelligence. It introduces subjects like pathfinding algorithms, heuristic, etc. Chapter 2 shows real-world examples of strategy games and artificial intelligence playing strategy games. These examples are used as input for specifying game requirements or creating a design of the learning algorithm. Chapter 3 presents some parts of the actual implementation. It also reasons for some design pattern usage. Chapter 4 looks into the testing of the implemented learning algorithm. It also evaluates the quality of created artificial intelligence. The last part of the thesis summarizes achieved goals and discuss possible future work.

Chapter 1

Theoretical Background

This chapter addresses the necessary subjects that need to be examined to implement a strategy game as well as the artificial intelligence playing the game. It describes the strategy game genre in more detail and introduces the common features of strategies. Also, it explains fundamental algorithms that are often useful in the genre. The chapter looks into basic algorithms on how to teach computer programs play games as well. It shows a link with optimization problems and introduces classic optimization algorithms.

1.1 Game Genres

Unlike books and movies, the usual categorization of games does not heavily rely on their story and narration. Videogames are often categorized based on their interaction with players, what experience they made to players or game objectives. Some of the common videogame genres are:

- **Action Game**

  The main objective of an action game is usually the elimination of enemies using combat skills. The genre contains fast, exciting and dynamic actions. Therefore action videogames require quick decisions and reactions from players.

- **Sports Game**

  The subject of this genre is any kind of sport, often directly simulating real life sport situations. An opponent in the game can be either a computer or another person, the latter being more popular among the players. As a result, the games are programmed so that multiple human players play the game on a shared device.

- **Role-playing Game**

  In RPG (Role-playing Game), players often identify themselves as heroes taking quests in a world full of adventures. Good storytelling, massive game world and extensive interaction with the player are typical characteristics of the genre. Generally, there are lots of NPC in the game world which are not controlled by a player. Their purpose is to make the game more interactive. The games are often set in the Middle Ages or a fantasy world.

- **Strategy Game**

  Players often perform the role of a leader with the ability to control a group of game units. The game world is affected by player commands in each game unit. The objective of the strategy genre is typically the domination of enemies with the fastest growing economy, the biggest military, or the most developed infrastructure.

Apart from the videogames genre mentioned above, there are other types of game genres, such as adventure, racing, puzzles, and so on. Moreover, videogame genres can have their subgenres. So the categorization can be much more complex. On the other hand, videogames do not have to abide by their categories. Developers are often trying to put several genres together to bring new experience to players.

1.2 Strategy Game

Common strategy games have no personification of a player, unlike RPGs or action games, where every player typically represents a certain character. Normally, the player of a strategy game affects a game world indirectly. He only gives orders for some game units to take action. Then the game units perform these actions by themselves. These actions are usually simple and do not require any planning nor coordination. After all, planning and coordination is the responsibility of a player.

Success in strategy games depends on tactical decisions and on the ability to plan. Players need to reasonably manage their resources and use them to accomplish the assigned tasks or optimize their state in a game. Resources in strategy games represent money, raw material, number of game units, and so on.

1.2.1 Categories of Strategy Game

Strategy games can be divided into 2 basic categories:

1. **Turn-based Strategy**

   Players alternate between 2 stages in TBS (Turn-based strategy). In the first stage, the player is waiting until the other players finish their moves. During the second stage, the player is making his moves, and the others are waiting for him to finish.

2. **Real-time Strategy**

   There is no switching turns between players in RTS (Real-time strategy). Moves of players are not made one after another anymore. In fact, the moves can be executed simultaneously. Consequently, the final result of the game is affected by additional factors like constant player attention, speed of player moves or the ability to react to unexpected situations.

1.2.2 Common Features of Strategy Game

The following sections show some basic characteristics of strategy games. All these characteristics are not obligatory for every strategy, on the other hand, they can be found in most of the games.

1.2.2.1 Different Types of Units

A classic feature of strategy games is the different types of game units. Every unit type has its own abilities and interactions with other units. As a result, different types of units enable the player to play the game in several different ways. Players can create their own specific strategy on how to approach the opponents based on a selection of types of game units.

However, to create different types of units with different interactions and effectiveness between each other can be a potential problem. The types need to be well balanced. One type should not overwhelm all the others to avoid the degradation. For example using only one powerful strategy with only one powerful type. Strategies should be balanced like the rock, paper, scissors game as illustrated by Figure 1.1.

1.2.2.2 Game Map

A game map of a strategy game is often drawn or shown as a 2D area. A good map can extend the number of possible strategies the player can choose from. The map usually consists of different types of terrains. One game unit could have an advantage over the others on a terrain. As a result, players have to take not only different types of units into consideration but also their position on the map. The game map is usually shown to a player from a bird's-eye view for a clearer perspective and efficient control of the game.

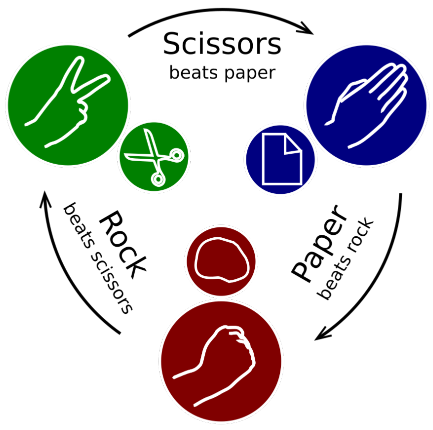


Figure 1.1: A diagram of the Rock paper scissors [3]

1.2.3 Fog of War

Fog of War is a common concept, especially for RTS combat games. In the beginning, players are able to observe only a small portion of the map. Little by little, players uncover the unknown parts of the map with their units. Once a location is discovered, the player will be able to see its location on the map.

However, to get present information about an already explored location, the player needs to place their units in the location. That information is only temporary. Once the player removes his units from the location, the map does not show the present state anymore. The player is exposed to an uncertain situation when Fog of War comes into play. He does not know where his opponents are or what they are doing. Therefore, the player is forced to explore the game map actively.

1.2.4 Aim of Strategy

The general goal of a strategy is to eliminate all game units of an enemy. Singleplayer games often offer sets of a mission or a campaign the player needs to finish. These missions are mostly similar to each other. The objectives of the missions can be for example:

- defend a town from an enemy
- build 30 factories
- destroy the base of an enemy

To make the missions more interesting, they are commonly connected with a story. The story develops as the player is finishing his missions. Another common approach is to challenge the player to complete his missions under specific additional conditions. These conditions could run as follows:

- defend a town from an enemy *and make no loss of your units*
- build 30 factories *under 25 minutes*
- destroy the base of an enemy *and do not use your special units*

When the player accomplishes a task with these additional requirements, he usually receives a special reward. In multiplayer, we often do not need any story-telling or additional mission requirements since playing against a human opponent is already challenging and interesting by itself, unlike playing with an emotionless computer.

1.3 Artificial Intelligence for Turn-based Strategy

TBS includes classic board games such as tic-tac-toe, checker, chess, or go. With the rapid development of computing technology, the effort to create computer programs capable of playing the games on human level was increasing. The programs were trying to not only reach the level of expert human players but also to overcome them. That effort is quite the opposite of the common effort in the game industry. The usual aim of game developers is to create an opponent that is at its most equal to a human player.

The game world of TBS is usually deterministic. A victory in TBS depends mainly on experience and tactical planning. As a result, success is earned fairly because random variables have a minimum impact on the outcome of the game.

Even though TBS games have convenient properties with minimum randomness and complete information about game board, it is quite dificult to play them perfectly. After all, the experience of most games is based on the inability to play them perfectly

It is not possible for the human mind, or even a computer to think about all the possible developments of the game. A simple game of chess has 10^{120} valid configurations [3]. The number is painful to figure out for a human just as much as for machine computing capabilities.

A human player handles combinatorial explosion of possible game states with intuition, experience from previous matches or trying to plan several moves ahead.[1] In spite of rapid advances in computer technology, it is still tough to explore all the possible moves. From a practical point of view, computer programs must use different kinds of approximations in order to play complicated games. For example, the quality of specific configuration in chess can be simply calculated as a ratio of one's chess pieces to the opponent's chess

[1]Expert chess players can memorize a valid board configuration immediately. On the other hand, they are no better than beginners at memorizing random board configurations. It appears that expert players are better at intuitively recognizing patterns for solving problems and eliminate pointless moves. [5]

pieces. The question is, how good the results produced by the approximation are.

TBS games can be easily represented by a game tree. To be more precise such games are called games in extensive form [4]. The game tree consists of:

- root representing an initial game state
- edges representing game moves
- vertices representing other game states
- leaves representing terminal game states

1.3.1 Pathfinding Algorithms

Considering the nature of strategy games pathfinding algorithms play a key role in strategy game implementation. They are an answer for questions such as what distance unit can reach, what the shortest path is to the base or how many units can attack in a specific range. Answering these questions helps not only in developing artificial intelligence but also computing valid game moves. Sometimes pathfinding algorithms can be extremely crucial when developing a game. For example, they took over 60 percent of processor time for AI (Artificial intelligence) in an RTS game called *Age of Empire 2* [6].

Pathfinding algorithms operate on a graph. A graph is usually represented by a set of vertices and a set of edges. An edge represents a connection between 2 vertices. The algorithms explore a graph via its edges starting at an initial vertex. Usually, some pieces of information are stored as the algorithm is exploring a graph.

There are lots of algorithms in graph theory. Some of the fundamental algorithms are BFS (Breadth First Search) and DFS (Depth First Search) [7]. Although BFS is very useful in finding the shortest path, it can be replaced with a more general algorithm called Dijkstra. The thesis focuses only on DFS and Dijkstra algorithm. DFS was chosen because it emerges from another algorithm called minimax.

1.3.1.1 Depth First Search

As the name suggests the algorithm searches the graph as "deep" as possible. Firstly the algorithm starts at a vertex. Secondly, it visits its adjacent vertex. Then it visits another adjacent vertex of the previous adjacent vertex. The process continues deeper and deeper as long as there are unvisited vertices.

DFS is a simple algorithm to implement as we can see in Algorithm 1.1. Its memory requirements are low since it expands only one neighbor of a vertex at a time. DFS does not guarantee to find the shortest path from one vertex to another.

Algorithm 1.1 Depth First Search

```
Input: an undirected graph $G$=($V$,$E$), $V$ – list of vertices, $E$ – list of edges,
    all vertex colors are WHITE, all vertex parents are NIL
Input: a vertex $u$ of $G$
Output: visited vertices have color attribute set to BLACK
1: function DFS($G, u$)
2:     $u.color$ = GRAY
3:     for all vertex $v$ adjacent to $u$ do
4:         if $v.color$ == WHITE then
5:             $v.parent = u$
6:             DFS($G, v$)
7:     $u.color$ = BLACK
```

1.3.1.2 Dijkstra Algorithm

Dijkstra algorithm is able to find the shortest path as the BFS algorithm is. However, it takes into account the “cost” of every move, unlike BFS. Every movement from a vertex to an adjacent vertex is the same for BFS.

Common features of TBS games are different types of terrains. A game unit can move through the terrains variously. One unit could travel longer distances via a terrain than the others. Which is why the Dijkstra algorithm fits TBS games better. BFS would treat all different terrains as they are the same.

The cost of a move is represented by a function in graph theory. The function takes an edge of the graph as an input and outputs a real number. However, the algorithm works properly only if every edge is evaluated as a positive number.

1.3.2 Minimax

One of the simplest program generating a non-random move in TBS would be based on comparing different game states. The program would need to have a function to evaluate a game state. The function would take a game state as an input and would output a number telling how promising the game state is for a player. Bigger the number, the more promising a game state for a player and the lesser promising game state for an opponent.

The algorithm would generate all possible player moves from a given game state. Then it would evaluate all generated game states with the function and pick a move that leads to the highest computed value. In summary, the program picks a move based on looking a move ahead.

However, looking only a move ahead is the biggest disadvantage of the program because it does not consider an opponent’s response. Luckily the program can be improved.

Algorithm 1.2 Dijkstra's algorithm

```
Input: an undirected graph $G = (V, E)$, $V$ – list of vertices, $E$ – list of
    edges, all vertex colors are WHITE, all vertex parents are NIL, all vertex
    distances are set to $+\infty$
Input: a function $l$: E $\to \mathbb{R}_{\geq 0}$
Input: a vertex $u$ of $G$
Output: visited vertices have color attribute set to BLACK and computed
    distances from the initial vertex $u$
 1: function Dijkstra($G$,$u$)
 2:     $u.color$ = GRAY
 3:     $u.distance = 0$
 4:     while exists a vertex $v$ such that $v.color$ == GRAY do
 5:         choose a vertex $v$ with the smallest $v.distance$
 6:         for all $w$ adjacent to $v$ do
 7:             if $w.distance > v.distance + l((v, w))$ then
 8:                 $w.distance = v.distance + l((v, w))$
 9:                 $w.color$ = GRAY
10:                 $w.parent = v$
```

Instead of calling evaluation function on all player's generated game states, the program generate all possible opponent's response to all player's generated game states. In the end, the program evaluates game states from an opponent's point of view since it looks two moves ahead. Moreover, the opponent picks a game state with the lowest value.

The program could be improved again to look three moves ahead. The general idea would be a program capable of looking n moves ahead universally. The program is called the minimax.

There are MAX and MIN players alternating moves in the minimax as can be seen in Algorithm 1.3. The minimax is building a game tree as players are making a turn. It stops at a leaf vertex or at the chosen depth. After that, a function evaluates the last game states and outputs a value. High values are good for the MAX player and bad for the MIN player. MAX's effort is to maximize the value of a game state and the MIN's effort is to minimize it.

Minimax searches a game tree recursively in a similar manner as DFS does. Then it executes the evaluation function on the last game state. As recursion unwraps it propagates up the computed value via the game tree.

The function starts evaluating at the second depth in Figure 1.2. At the first look, MAX's best option is to turn right to get 9 as the highest evaluated leaf. However, the MIN player will make a turn after the MAX and MIN would choose leaf evaluated with 2. As the minimax algorithm propagates computed values up, it is clear from Figure 1.3 that the MAX player should turn left.

Algorithm 1.3 Minimax algorithm

```
Input: state representing a vertex from a game tree
Input: depth representing at what depth algorithm shall stop, depth must
    be bigger than 0
Input: an evaluation function eval: $state \rightarrow \mathbb{R}$
Output: best move based on the function eval
 1: function MINIMAX(state, depth)
 2:     $bestMove$ = NIL
 3:     $bestScore$ = -∞
 4:     for all move in all possible moves of MAX player do
 5:         newState = generateNewState(move, state)
 6:         $newScore = MinPlayer(newState, depth - 1)$
 7:         if $newScore > bestScore$ then
 8:             $bestScore = newScore$
 9:             $bestMove = move$
10:     return $bestMove$
11: function MINPLAYER(state, depth)
12:     if depth == 0 then
13:         return eval(state)
14:     else if isStateTerminal(state) then
15:         return eval(state)
16:     minimalScore = +∞
17:     for all move in all possible moves of MIN player do
18:         newState = generateNewState(move, state)
19:         $newScore = MaxPlayer(newState, depth - 1)$
20:         minimalScore = MIN(newScore,minimalScore)
21:     return minimalScore
22: function MAXPLAYER(state, depth)
23:     if depth == 0 then
24:         return eval(state)
25:     else if isStateTerminal(state) then
26:         return eval(state)
27:     maximumScore = -∞
28:     for all move in all possible moves of MAX player do
29:         newState = generateNewState(move, state)
30:         $newScore = MinPlayer(newState, depth - 1)$
31:         minimalScore = MAX(newScore,minimalScore)
32:     return maximumScore
```

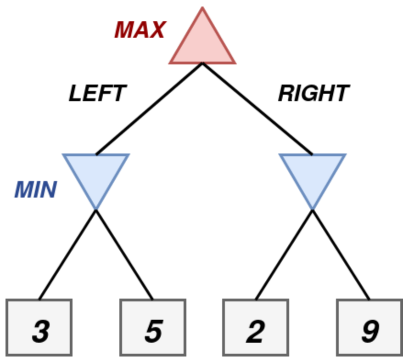


Figure 1.2: Minimax diagram with evaluated leaves

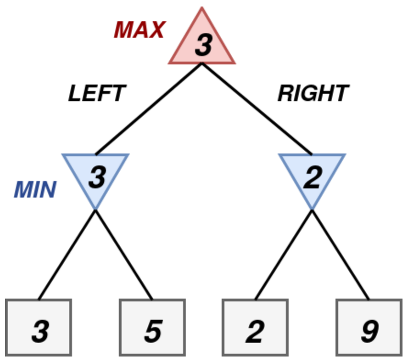


Figure 1.3: Minimax diagram with propagated values from leaves

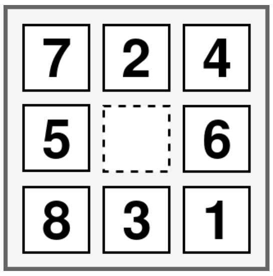


Figure 1.4: An initial state of the puzzle

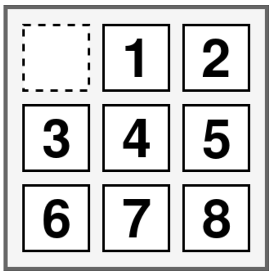


Figure 1.5: The goal state of the puzzle

1.3.3 Heuristic

The book *Swarm Intelligence* defines heuristics as shortcuts in the search strategy that reduce the size of the space that needs to be examined [9].

Space can be searched with an uninformed search such as BFS/DFS if no extra information about the space is provided. However, when space contains information that made to be used of, it can be possibly embedded into the original uninformed search to make it more efficient. Uninformed searches using the extra information are called informed searches or heuristic searches. [8]

One of the earliest heuristic search problems was the 8-puzzle. It is a 3×3 board with 8 tiles and an empty tile. The objective of the puzzle is to slide the tiles horizontally or vertically into the empty tile until all tiles match the goal configuration. [8]

To find solution of the puzzle efficiently a proper heuristic is needed. According to [8], commonly used heuristics are:

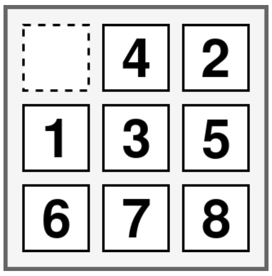


Figure 1.6: A state of the puzzle

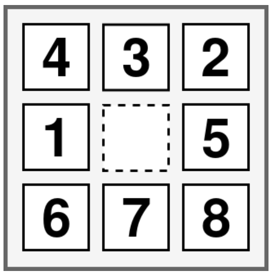


Figure 1.7: A state of the puzzle

- h_1 – a number of misplaced tiles
- h_2 – a sum of Manhattan distances [2] of the tiles from their goal positions

By definition h_2 seems to be more sophisticated than h_1. If both of the heuristics evaluate states of a puzzle in Figure 1.6 and Figure 1.7, the heuristics will differ. h_2 computes values of the boards shown in Figure 1.6 and Figure 1.7 as 6 and 4 respectively. Thus a state from Figure 1.7 seems to be more promising to get to the final goal state. On the other hand, h_1 is unable to tell the difference between the boards because the evaluation for both states is 2. It appears that h_2 is more sensitive as h_1 is unable to tell a difference between the states.

When the goal state of the puzzle is 22 moves from an initial state an informed search with h_1 must explore 18 094 possible states. If the informed search uses h_2, then it must explore only 1219 possible states to find a solution or a goal state. If one heuristic is better than the other, it is said the heuristic dominates the other. The result shows that a good heuristic can dramatically improve the performance of an algorithm. [8]

In practice, uninformed searches are useful mostly for trivial problems as they require exploring larger space. For practical problems often informed searches with some heuristics are needed. On the other hand, uninformed searches are more general as they do not make use of any specific piece of information and can be used on various problems. Usually, for a domain-specific problem, a domain-specific heuristic is needed such as heuristics for solving a maze, the 8-puzzle or game tree with minimax. Each of these problems requires different domain knowledge. However, the requirements to provide heuristic can be unwanted. Heuristic invented by a human is not equal to artificial intelligence. Artificial intelligence itself should come up with a heuristic instead of a man. [11]

To let the machine create a heuristic itself seems challenging. One of the possible solutions is to learn from experience. In this context, experience

[2]It is a distance measured along axes at right angles. There are no diagonal movements in Manhattan metrics. In two dimensions the Manhattan distance between coordinates (x_1, x_2), (y_1, y_2) is defined as $|x_1 - x_2| + |y_1 - y_2|$. [10]

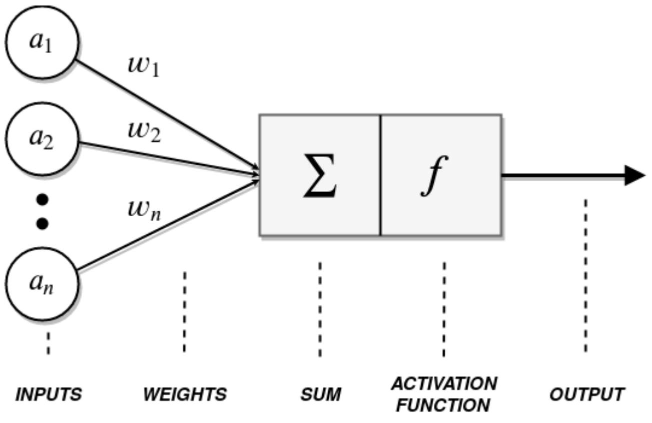


Figure 1.8: Diagram of the artificial neuron

means letting a machine solve a problem again and again until it starts to produce desired results. It requires a method of telling the machine if it does well or bad. For instance, providing examples of states with correct output that a machine heuristic should produce. Hopefully, from these examples, a machine should be able to create heuristics that compute reasonable outputs of other states. One of the techniques to learn from experience uses an artificial neural network. [8]

1.3.4 Artificial Neural Network

ANN (Artificial Neural Network) is a computational model inspired by the structure of a brain. It consists of units representing a simplified biological neuron. The units are connected with directed links. They transfer signals between each other and transform the signals with an activation function. Moreover, each link is amplified or unamplified with a numerical weight.

ANN can be simply considered as a black-box taking sets of inputs and transforming them into sets of outputs. It is often hard to reason and interpret specific settings of weights and biases for a given problem. There are researches specifically aiming to understand more ANN such as studying the role of memorization and generalization in ANN [14, 15].

1.3.4.1 Neuron

It is a basic unit of ANN. In general, a neuron has an arbitrary number of inputs. A model of the neuron is given in Figure 1.8. The inputs are transformed into a single output. At first, neuron computes a weighted sum of its inputs:

$$weightedSum = \sum_{i=1}^{N} w_i a_i + b$$

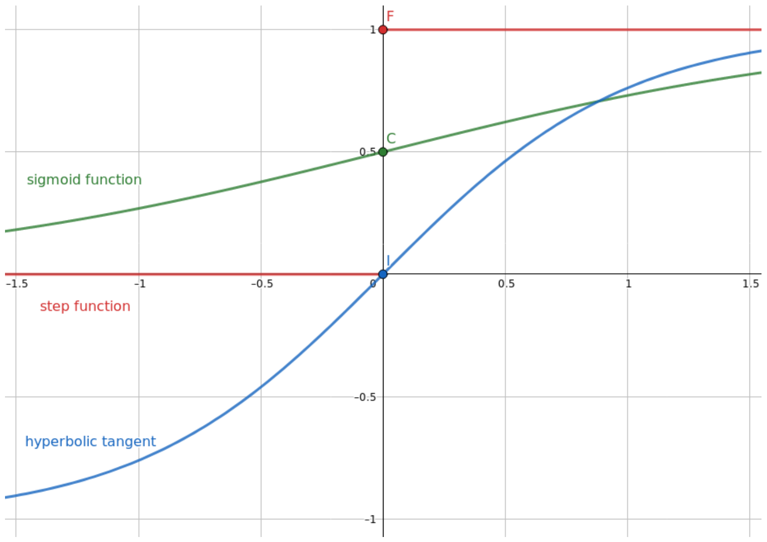


Figure 1.9: Graph of activation functions

where:

- N – number of neuron inputs
- a_i – i^{th} neuron input
- w_i – numeric weight of i^{th} neuron input
- b – threshold, bias representing how willing is a neuron fire its signal

Then an activation function f is applied to get a final output:

$$finalOutput = f\,(weightedSum) = f\left(\sum_{i=0}^{N} w_i a_i + b\right)$$

1.3.4.2 Activation Function

An activation function affects a network output and performance of optimization to get the desired value of weights and biases. For example, one activation function can be more computationally expensive than others, thus a convenient approximation might be applied instead [13].

There are many functions that can be used in ANN [16]. Typically presented functions are sigmoid function and step function (hard threshold). They are shown in Figure 1.9

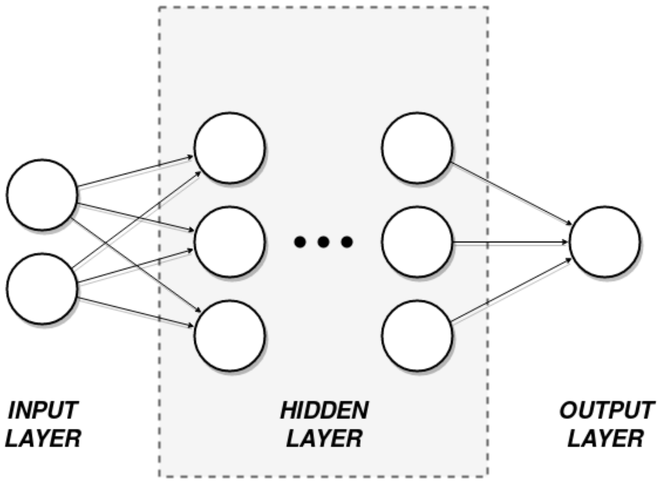


Figure 1.10: Diagram of the feed-forward neural network

1.3.4.3 Feed-forward Neural Network

There are many ways to connect neurons to form a network [17]. One of the most basic structures that are used in almost every complex network is feed-forward neural networks.

Neurons are arranged in layers in a feed-forward neural network as detailed in Figure 1.10. Their connections are only in one direction. Neuron inputs are strictly connected with neuron outputs from the previous adjacent layer. The layers are distinguished into 3 types:

- input layer – the first layer of the network
- output layer – the last layer of the network
- hidden layer – layers between the input layer and the output layer

1.3.4.4 Learning of Artificial Neural Network

As stated in the previous chapter about heuristic, an algorithm learning from experience needs examples of correct outputs and a method telling how wrong or right it works. The function quantifying how well an algorithm is achieving the desired goal is called the cost function. A smaller value of cost function means a better solution. One of the cost functions is the MSE (Mean Square Error). It is defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

where:

- n – number of predictions
- Y_i – vector of actual values (these are examples of correct outputs)
- $\hat{Y}_i$ – vector of predicted values (these are outputs predicted by an machine)
- w_i – numeric weight of i^{th} neuron input
- b – threshold, bias representing how willing is a neuron fire its signal

In the context of ANN, the algorithm needs to set weights and biases in a way to minimize the cost function. In other words, the algorithm needs to minimize multivariable function since cost function depends on a vector of weights and vector of biases.

A powerful method to minimize an multivariable function f is gradient[3] descent algorithm. It starts at a point a_0. Then it computes other point with following formula:

$$a_{n+1} = a_n - \alpha \, \nabla f(a_n)$$

where:

- $\alpha \in \mathbb{R}_{\geq 0}$ – a learning rate
- $\nabla f(a_n)$ – a gradient of f at a point a_n

Since a negative gradient indicates a direction of steepest descent in a point then the gradient descent algorithm can get to a local minimum using the formula recursively with small enough α.

In terms of weights and biases, the gradient of a cost function is computed with an algorithm called backpropagation. More about backpropagation can be learned at [19], [20].

The success of the gradient descent depends on a cost function and a learning rate of α. If a function is not differentiable at a point, the gradient cannot be computed using classic methods. Moreover changing gently the weights and biases may not show any improvements in the cost function [21]. It is similar to the heuristics of 8-puzzle since one heuristics is not able to tell the difference and is dominated by the other. Thus gradient descent might be inapplicable for some problems and other algorithms that need to be used.

[3]Not rigorously, gradient of an f is a vector of all partial derivatives of f. Partial derivative is just like "usual" derivative with respect to only one variable since all the other variables are considered as constants. The partial derivatives of $f(x, y) = 3x^2 + xy^2$ are $\frac{\partial f}{\partial x}(x, y) = 6x + y^2$ and $\frac{\partial f}{\partial y}(x, y) = 0 + 2xy$. Thus the gradient is $[6x + y^2, \, 2xy]$.

1.3.5 Optimizations

Optimization is a task minimizing or maximizing a function $f(x)$ by changing x [20]. Some optimization methods work with well-known or well-defined problems such as a method of least squares, linear programming [22].

When the optimization problem is not well-defined, more general methods need to be used. One of the methods are iterative optimization algorithms. They solve the problem by trial and error. Moreover, the algorithms often require minimum knowledge about an optimized function.

1.3.5.1 Hill Climbing

To optimize function f the algorithm starts at a point a. Then it generates a new adjacent point B. If $f(b) > f(a)$ holds then original a is replaced with the point b. The process is repeated until specified criteria is met. Hill climbing pseudocode is presented in Algorithm 1.4.

Hill climbing does not have any inner states or memory. It simply takes the first good solution without looking ahead. Thus hill climbing is sometimes called greedy local search. Even though the hill climbing initial point could be a poor solution, it often quickly moves towards a better solution. [8]

Generate neighbors differ among various problems. If the point a would be a vector, then the adjacent point could be a vector with a specific distance from the a. For a binary vector, $\{0, 1\}^n, \ n \in \mathbb{N}$, the adjacent vector could be generated by flipping $k \in \mathbb{N}$ bits.

According to [8], drawbacks of hill climbing are:

- local maxima – if hill climbing finds a point that is better than all its neighbors then the algorithm get stuck since it always looks for the best local maxima.

- flat area – if hill climbing searches space where all points are evaluated equally, then it gets lost because no local uphill from the flat space exists.

Algorithm 1.4 Hill Climbing

```
Input: an initial point $a$
Input: a function $f$
Output: a local or global maxima of $f$
1: function HillClimbing($a$,$f$)
2:     while shouldStop() == false do
3:         $b$ = generateNeighbour($a$)
4:         if $f(b) > f(a)$ then
5:             $a = b$
6:     return $a$
```

1.3.5.2 Simulated Annealing

Simulated annealing is an alternative to hill climbing. Problem with hill climbing is that it never makes a move towards less promising areas temporarily to escape from small local maxima. Simulated annealing is very similar to hill climbing. It starts at a point and tries to improve a solution via a continual generation of new points.

Simulated annealing pseudocode can be seen in Algorithm 1.5. The main difference from hill climbing is when simulated annealing accepts the adjacent point. It picks a new solution with probability p. The probability depends on how much worse or better neighbor solution is. Better the neighbor bigger the probability to pick it. The probability also depends on the variable T called temperature. Higher the temperature bigger the probability to pick neighbor solution as well. The temperature is dynamically changing during algorithm iterations. According to [23], the probability p can be computed by the following function:

$$P(a,b) = \frac{1}{1 + e^{\frac{-\Delta E}{T}}}$$

where:

- b is a point generated from a
- $\Delta E = f(b) - f(a)$ and f is a maximized function
- T – temperature at a current iteration

A challenging area in simulated annealing is how to change the temperature step by step during the iteration and what initial temperature shall be set. There are lots of temperature schedules for changing it. One of the simplest ways is to compute temperature at k^{th} iteration as $T_k = T_0 \times \alpha^k, k \in \mathbb{N}, \alpha \in \mathbb{R}$. [24]

Algorithm 1.5 Simulated Annealing

```
Input: an initial point $a$
Input: a function $f$
1: function SimulatedAnnealing($a$,$f$)
2:     while shouldStop() == false do
3:         $b$ = generateNeighbourFrom($a$)
4:         $\Delta E = f(b) - f(a)$
5:         $T$ = computeTemperatureFromSchedule()
6:         $P = \left(1 + e^{\frac{-\Delta E}{T}}\right)^{-1}$
7:         change value of $a$ to value of $b$ with probability $P$
8:     return $a$
```

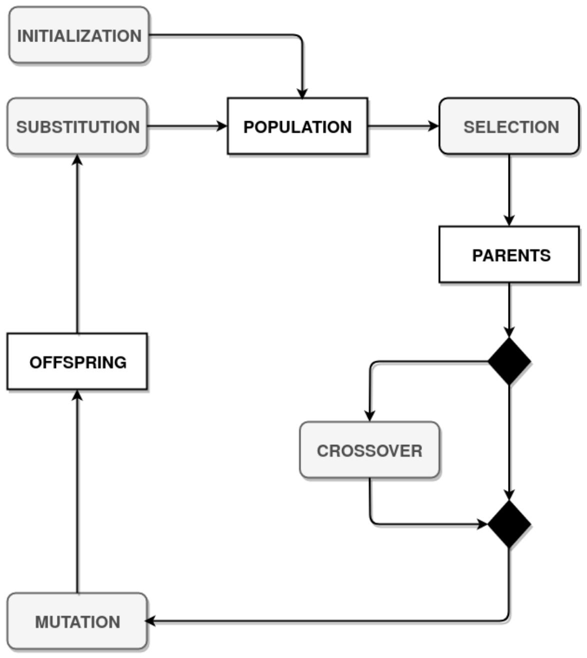


Figure 1.11: Genetic algorithm diagram

1.3.6 Genetic Algorithm

The genetic algorithm belongs to a much wider group of Evolutionary algorithms. Evolutionary algorithms are based on natural evolution where only the fittest individuals survive. In a natural evolution, survival is achieved through reproduction. With a little luck, the most favorable characteristics are passed from parents to offspring and good genetic information is preserved. Individuals inheriting bad characteristics most likely lose the battle to survive. [25]

Algorithms like hill climbing or simulated annealing often deal with a single promising solution. The evolutionary algorithm works with several solutions. Moreover, they are introducing interactions between solutions. Candidate solutions can influence each other and create a final solution with collective effort. Solutions of evolutionary algorithms are represented by genotype like binary strings, a vector of numbers, etc. Evolution algorithms maintain a population of candidate solutions. The algorithms need to have a criterion function, called a fitness function to determine the quality of solutions. [22]

Evolution algorithms are a compromise between exploration and exploitation principles. Exploitation refers to local search improving already found solutions and exploration relates to random search in a wider area to explore different promising regions. [22]

1.3.6.1 Genetic Operators

The genetic algorithm searches a space via genetic operators. Its general flow can be seen in Figure 1.11. According to [26], the main driving operators of the genetic algorithm are:

1. **Selection operator**

   At the end of each iteration or generation selection operator chooses individuals for a new population. There are many possibilities for selection operators. One of the best-known is roulette-wheel selection, tournament selection, and rank-based selection.

2. **Reproduction operator**

   The new solution is created with reproduction algorithms combining two or more parents together. Commonly used operators are one-point crossover, uniform crossover.

3. **Mutation operator**

   A mutation changes parts of a solution genotype. The basic mutation operator is for instance bit-flip. The probability of mutation is a typically small number.

1.3.6.2 Niching Methods

The common problem of evolutionary algorithms is premature convergence. It happens when individuals are too similar in population. Although exploitation and exploration can be affected by genetic operators, the result is often insufficient. If the population is not varying enough, then the evolution operators do not work well. For instance when two identical individuals crossover without any mutation, then the offspring are similar to the parents in every detail. Thus classic evolution algorithms are extended with niching methods trying to prevent premature convergence. [26]

One of the popular methods is fitness sharing. The main idea is that too many similar individuals share the same fitness value. A typical sharing function can be expressed by the following formulas:

$$f_i' = \frac{f_i}{m_i} \tag{1.1}$$

$$m_i = \sum_{j=1}^{N} sh(d_{ij}) \tag{1.2}$$

$$sh(d_{ij}) = \begin{cases} 1 - \frac{d_{ij}}{\sigma_s}, & \text{if } d_{ij} < \sigma_s \\ 0, & \text{otherwise} \end{cases} \tag{1.3}$$

In Equation 1.1, f_i represents an actual fitness of i^{th} individual in a population. Similarly f'_i represents a shared fitness. Variable m_i is a niche count measuring with how many individuals the current fitness is shared. In Equation 1.2, N denotes size of a population and d_{ij} refers to distance between the i^{th} individual and the j^{th} individual. In Equation 1.3, the function sh measures the similarity between two population elements. It returns to zero if the elements are different enough. Variable σ_s represents a threshold telling if two individuals are too similar. [27]

1.3.6.3 Applications

Evolution algorithms are more efficient in non-differentiable, discontinuous problems than classic optimizations. Genetic algorithm, in particular, has many real-world applications from robotics to investment strategies. [28]

For instance, the genetic algorithm was used in the FPS (First-person shooter) game *Quake 3*. The genetic algorithm optimized logic controllers of a game bot. The controllers indicate how much the bot "wants" to do something such as 78% liking for picking a gun or 56% liking for picking the armor. [29]

Chapter 2

Analysis and Design

In the beginning, any software project should define its functions and requirements. Requirements often served as an agreement with a client. They describe what the client wants and what needs to be done. Hence developer work more likely meets the client's requests. On the other hand, defining requirements is important regardless of a client. It helps developers to create a general mental picture of a project and clears goals that need to be achieved. Moreover, these definitions are used later as input for the implementation. Analysis of similar projects can help to specify reasonable requirements and functions as well.

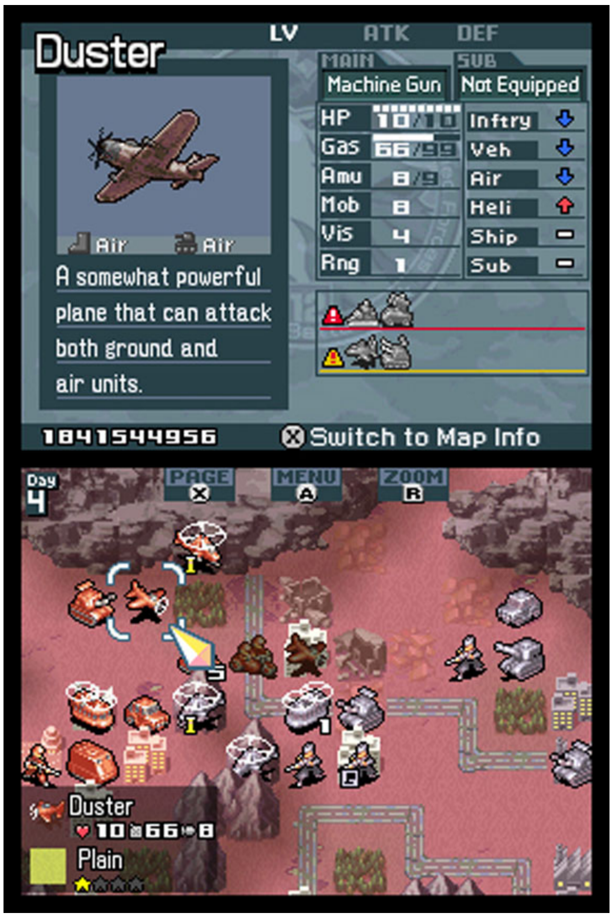
2.1 Examples of Turn-based Strategy Games

Since the thesis focuses on TBS games, the selection is narrowed just to one specific genre. However, only a small sample of games was chosen due to a big amount of produced games of any genre.

This sample reflects the author's opinion. Apart from that, choice take other factors into consideration like popularity and age. This sample contains an open-source game, a newly released game from 2019, and a popular game at its time.

2.1.1 Advance Wars

It is a game created for a pocket console *Game Boy Advance*. The game was developed by a Japanese studio called *Intelligent System*. It was released in 2001. This strategy game is set in a military environment. The aim of the game is to defeat enemy units. It can be done in 2 ways: destroying all enemy units or capturing an enemy base. The game is designed for 2–4 players with an option to compete with artificial intelligence. The project provides more



A somewhat powerful plane that can attack both ground and air units.

Figure 2.1: A screenshot from *Advance Wars: Days Of Ruin* [4]

than 100 game maps, broad and a very user-friendly tutorial. A player learns the basic game mechanics via entertaining and easy missions.[4]

The player can choose from several characters. The characters do not only provide a visual representation of the player as a commander but also they can change game statistics of their game units and introduce special abilities. *Advance Wars* got very positive ratings for its game mechanics, multiplayer and graphic visualization taking full advantage of a game console hardware at that time [30]. The game is presented in Figure 2.1

2.1.2 Wargroove

Wargroove is a commercial project from a British studio called *Chucklefish*. The strategy is situated in a fantasy environment with dragons, knights and

[4]*Advance Wars* was originally sold only in the Japanese market because of disbelief that a complicated game would success outside of the market. Thus developers made a detailed tutorial to make the game more user-friendly and to avoid reading any manual. [31]

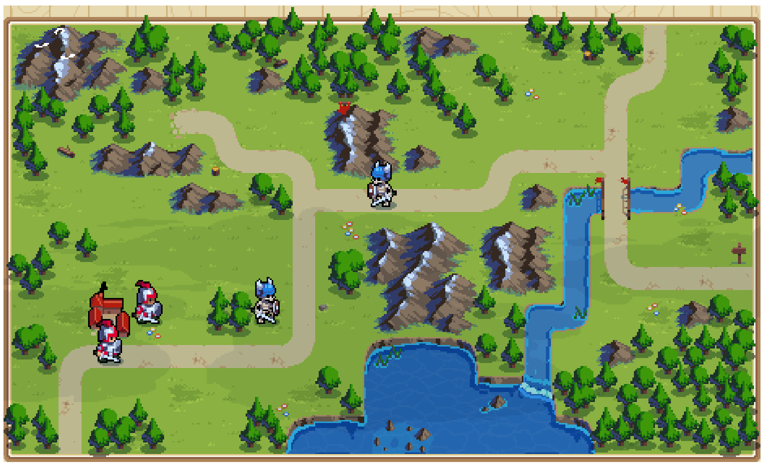


Figure 2.2: Screenshot of the *Wargroove* [5]

catapults. The game was released in 2019. It supports platforms like PC, Nintendo Switch, Xbox One, and PS4. Additionally, it has an online multiplayer for up to 4 players [32].

Besides TBS characteristics, the project takes some of the RPG features. Every player chooses a game hero. The hero is placed in a game map like any regular game unit. The difference from *Advance Wars* is that a chosen character is the only visual representation of commander but does not appear directly on a map in *Advance Wars*. This game world is presented as a 2D pixel art. The structure of game maps has a square grid format. Game battles end under a common condition like destroying all opponent's units, taking over the opponent's base. The additional condition is only destroying the opponent's hero.

Wargroove got mostly positive ratings with the conclusion, that it is a respectable successor to the *Advance Wars* game that some players wanted for a long time [33]. The game has very appealing audiovisual content. There are animated cutscenes that develop the story of the game. The game can be seen in Figure 2.2. Developers claim they wanted to revive the charm and accessibility of games that inspired *Wargroove*. The game is focused on high-resolution pixel art, online playing or modding capability. Developers do not even hide that their game is based on Advance Wars series [32].

2.1.3 Tanks of Freedom

It is an open-source game from Polish studio *p1x*. The game is different from the others with an isometric game map as illustrated by Figure 2.3. It uses old-fashioned graphic visualization using pixel art as well. The project supports operating systems such as Linux, Windows, OSX and Android. Although it is only a free and open-source project with minimum advertising, the game

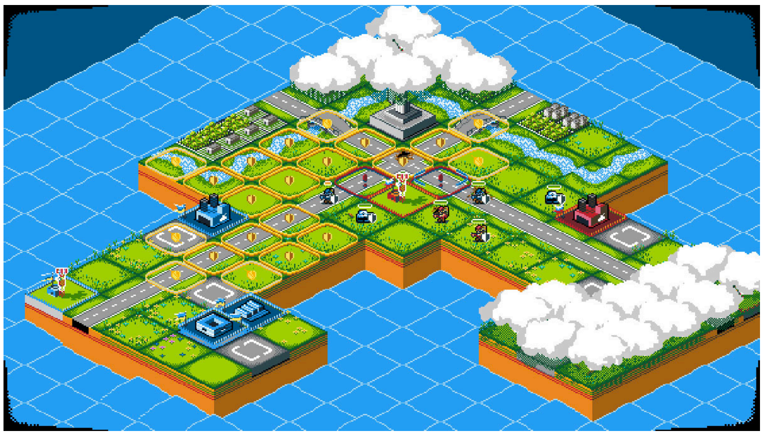


Figure 2.3: Screenshot of the Tanks of Freedom [6]

reached over 30 000 downloads for PC and Android. The game was released in 2015. Since the game is open source, several other ports were made by their community to play the game on other systems like FreeBSD, F-Droid, Pandora or Mageia. It is also possible to play the game through web browsers [34].

The game is designed for 2 players and with an option to compete with artificial intelligence. The game contains a campaign with 14 missions and a map editor. The player wins the moment he captures all of the enemy's strategic buildings [35].

2.2 Examples of Artificial Intelligence Playing Turn-based Strategy

The section introduces two programs playing the TBS game. Each program deals with a different problem. A program called *Blondie24* addresses the problem of learning games with minimal human expertise. The second program *Hero Academy* is solving a great branching factor of a specific TBS game. Both programs make use of an evolution algorithm but each of them applies the algorithm in a very different way.

2.2.1 Blondie24

It is a computer program playing checker as shown in Figure 2.4. *Blondie24*[5] is based on the minimax algorithm with a special evaluation function. The

[5]The first name was David1101 but no one wanted to play with him online. The second name was *ObiwanTheJedi*. People wanted to play with *ObiwanTheJedi*. On the other hand, some players swear to its mother. Last but not least was *Blondie24*. *Blondie24* does not have any problem with swearing or lack of players. *Blondie24* was actually asked to date several times. [36]

function was created using a feed-forward neural network and evolution algorithm. The program was tested in 165 games against human opponents via website `www.zone.com`. *Blondie24* was better than 99.61 percent of all rated zone players. [37]

A control experiment showed that a trained neural network provides an advantage over a classic heuristic depending merely on the checker piece differential. The neural network was tested in 14 games against the classic heuristic. The network won 2 games and the other 12 games ended in a draw. Apart from that, the neural network held more pieces over the classic heuristic most of the time, specifically 10 of 12 draw games. [38]

The neural network was trained via an evolution algorithm with population size 15. Each individual represents a neural network in the population. The entire population was initialized randomly. The structure of the network was fixed and the algorithm optimized only weights and biases. To compute an individual's fitness, it plays 1 game against 5 random opponents from the population. An individual could earn -2 points for a loss, $+1$ point for a win or 0 points for a draw. [38]

The input of the trained neural network is the checkerboard. It is represented as a vector of 32 numbers from a set $\{K, -K, 0, 1, -1\}$. When there is no piece in a board tile, the tile is transformed to 0. A positive value represents a player's piece and a negative value represents an opponent's piece. Variable K refers to a checker king piece. The appropriate value of the king was optimized by the evolution algorithm as well. The initial value was 2.0. The neural network has 3 hidden layers. They consist of 91, 40 and 10 neurons respectively. The output was a single neuron. Bigger the number of the output neuron, the more promising board for a player and more disadvantageous to an opponent. [38]

The evolution algorithm was executed on the *Pentium II* 400 Mhz. Thus the depth of the minimax algorithm was adjusted to 4 in order to get reasonable execution time. The algorithm ran for 8 weeks and iterated over 230 generations. [38]

2.2.2 Hero Academy

The game was created in order to study different approaches to the searching game tree. Hero Academy is a regular TBS game. It tries to solve the huge branching factor of games where a player can move with all its units at one turn. Apart from classic board games like chess with branching factor 30 or Go with branching factor 300, the multi-action games could have branching factor over 1 000 000[6]. [39]

The basic idea for Hero Academy is to use an evolutionary algorithm to evolve sequences of moves. So the game tree is not explored by a standard

[6]If a player handles 6 game units and each unit has 10 possible moves, then there are 10^6 possible moves in a player turn. [39]

Figure 2.4: Graphic visualization of Blondie24 [7]

algorithm like minimax. Unlike *Blondie24* using an evolutionary algorithm for training minimax evaluation function, the evolutionary algorithm is directly used to create player's moves. Creating a sequence of moves directly does not take into consideration opponent's actions. However, the result shows that evolutionary algorithms outperform classic searches. [39]

2.3 Game Requirements

The requirements specify basic features of the game and what it is able to do. Following requirements were created taking into account the analysis of game examples. It uses some common characteristics of all the examples like their game objective.

1. **Functional Requirements**

   F1 Graphics

   - F1.1 Game elements with pixel art style.
   - F1.2 Overall game retro design.

   F2 Multiplayer

   - F2.1 The game can be played by several players sharing a keyboard.
   - F2.2 User can play against computer as well as human player.

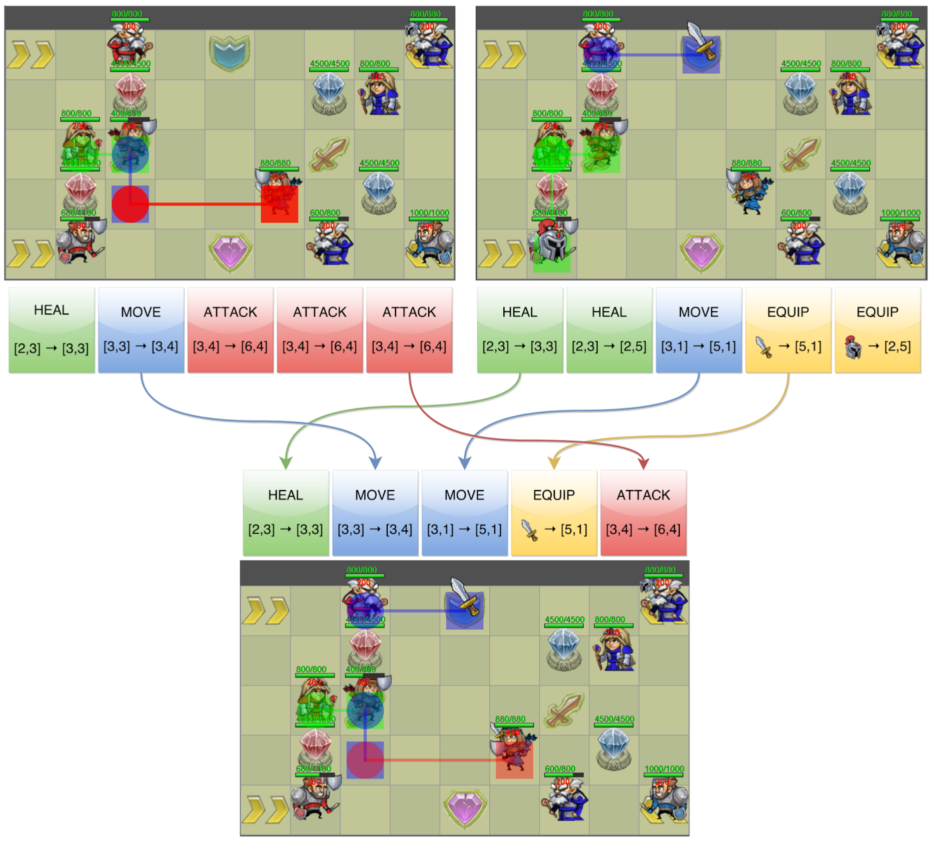


Figure 2.5: Crossover of moves [8]

F2.3 User is able to let computer players play with themselves.

F3 Game maps

F2.1 The game provides at least 4 pre-created maps.

F2.2 Game map has a form of 2D grid.

F4 Game mechanics

F2.1 There are different kinds of units and buildings to enhance tactical planning.

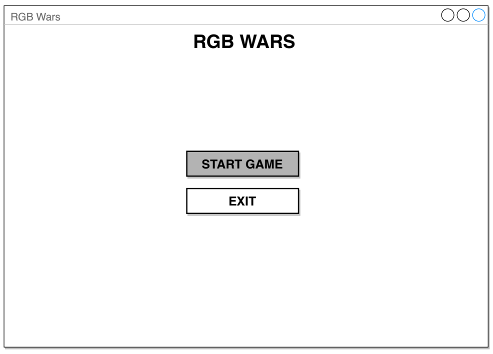
F2.2 The game objective is to destroy enemy base or all enemy units.

2. **Non-functional Requirements**

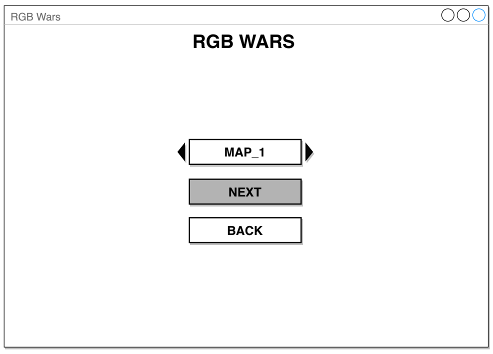
NF1 The game uses the English language.

NF2 The game supports at least Linux operating system.

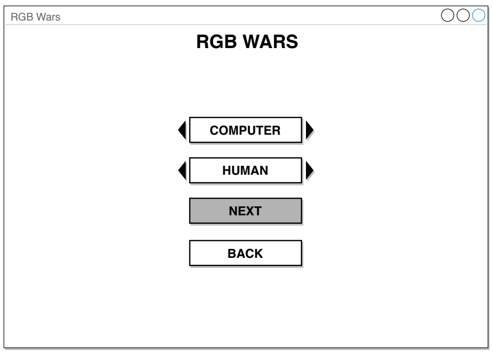
NF3 Game is extensible with minimum changes in the source code.



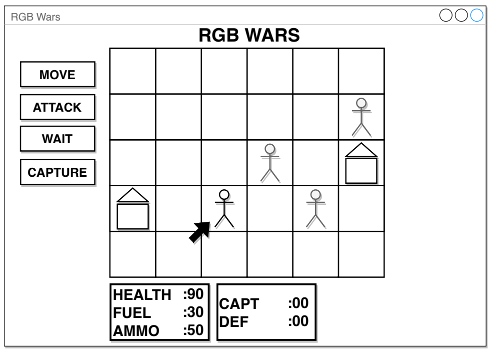
(a) Main menu.



(b) Menu for chosing maps.



(c) Specifying types of players.



(d) Game widget layouts.

Figure 2.6: Wireframes for menus and the actual game.

2.4 Game Wireframes

Figure 2.6 illustrates basic widget layout for 4 different screens. They also show elementary navigation from a main menu to playing the game. At first user has to choose one of the pre-defined maps. Afterwards user sets types of players for the game. In the end it is switched to another screen to play. As can be seen in Figure 2.6d, below widgets shows unit status like health and ammo. The neighbouring widget displays information about a terrain. Both statuses correspond with a tile where a cursor is placed. Menu of possible actions is dynamically changing according to units abilities and nearby objects. So the menu is not showed permanently as indicated in Figure 2.6d

2.5 Game Design

Since the thesis goal is to create functional game prototype, the design is aiming for simplicity. Thus the game does not rely on multithreading architecture avoiding complications such as thread synchronization, concurrency or unpredictable results. It also does not overuse abstraction or polymorphism using hierarchy of classes. Since polymorphism with classes usually involves calling indirect methods via pointers and using operators `new` and `delete` [7], the game tends to not rely on the concepts in critical path like needed computation for AI. On the other hand, abstraction for AI player is needed to provide different implementation of program playing the game without large code repetition. Template programming gives the essential abstraction rather than polymorphism with virtual calls.

2.6 Artificial Intelligence Design

To implement strategy decision-making, minimax is a convenient algorithm. It allows looking arbitrary moves ahead and taking into consideration enemy moves. On the other hand it needs a decent evaluation function in terms of limited search depth. Thus minimax could be a favourable AI component taking into acounts possible enemy moves with good enough evaluation, or heuristic function. In order to create AI with minimum human input, neural networks are one of the possible ways how to achieve the goal as stated in Chapter 1.3.3. They will be used to create the very evaluation function of minimax. The neural network representing evaluation function will be trained via self-play in a similar manner as *Blondie24* program. The algorithm training *Blondie24* ran on *Pentium II* 400 mhz, yet it was possible to create a very good competetive program.

The idea is to use the same approach for more complex games unlike checker. Since nowadays computers are much faster than the old *Pentium*, it might compensate the complexity of the game while training the program. Hopefully, the trained evaluation function will make up for a large branching factor during minimax search just like better heuristic function helps to solve the 8-puzzle with smaller amount of visited states. The algorithm for the neural network training needs to be easily extensible and allowing different tweaks. Also, it should enable being used for different kinds of problems other than training neural network playing a game. A way of dealing with that is to create general genetic algorithm class as a core of the optimization algorithm. Then possible tweaking and extensibility is done by means of different genetic operators as illustrated by Figure 2.7. The algorithm can solve different problems by defining new fitness function.

[7] The operators are used for dynamic memory allocation in C++. However, they consume a big amount of time. Memory can be fragmented when different objects are allocated in random order. Moreover, compiler cannot easily optimize code with pointers. [40]

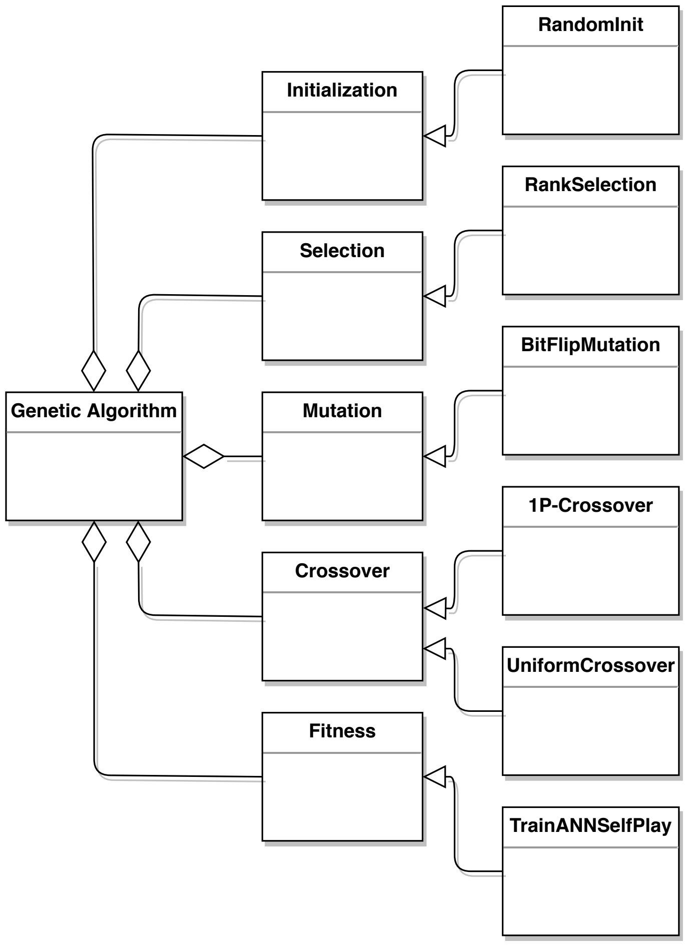


Figure 2.7: Domain model of Genetic algorithm.

2.7 Used Technology

C++ language will be used for the implementation of the game and AI. C++ allows wide variety of programming paradigm like object-oriented programming, generic programming and procedural programming. Moreover C++ compilers provide non-trivial error detection and code optimization. [41]

For graphic input SDL (Simple Direct Media Layer) will be used since it natively works with C++. [42]

Definition of game objects or game maps should not be hardcoded directly inside a source code in order to maintain adding new maps and units easily. Thus definitions will be loaded from a file. The program will use json file format and library JsonCpp for easier manipulation [43]. Unit testing will be done with Catch2 library [44].

Chapter 3

Realisation

In this chapter, concrete implemented classes are described. It looks into the main components of the created game as well as the core of optimization algorithms. The chapter also shows usage of some design patterns like command or data locality and clarifies their correct application.

3.1 Game Architecture

As the implemented game includes dozens of classes, only some key parts of the program will be described.

3.1.1 Game Class

The `Game` class main components are `GameBoard`, `GameBoardGraphics` and vector of `Players` as can be seen in Figure 3.1. It handles an entire game flow such as alternating players, ending game, game animation via its components. The graphic input is represented by `GameBoardGraphics` class. Players can alter gameboard via two different controllers. One controller is for human player and the other for AI. The entire game flow takes place in a simple game loop method. Game loop is a simple pattern. The loop runs continuously processing user input, updating game state and displaying the game [45].

3.1.2 GameBoard Class

`GameBoard` class consists of arrays globally shared with each instances of it. The arrays store information about damages between units or move cost of all unit. Objective of a `GameBoard` is to hold enough information to describe complete individual game state. It holds information about player units and buildings or their positions.

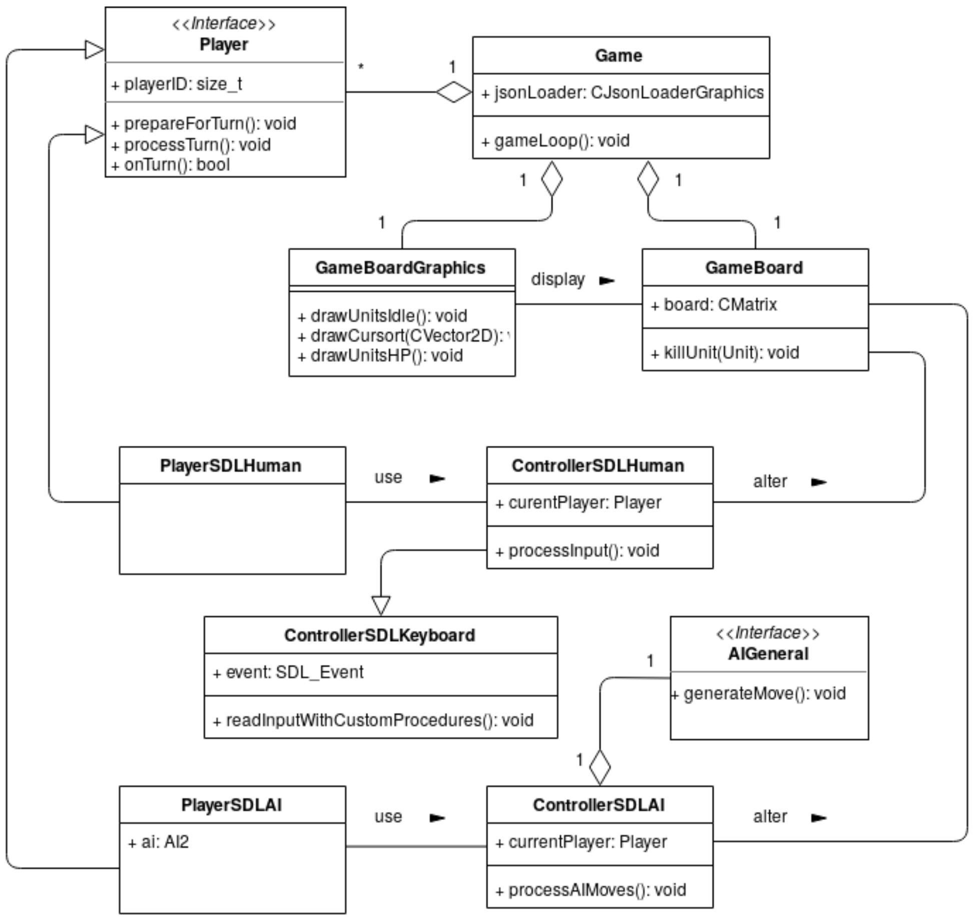


Figure 3.1: Diagram of Game class components

3.1.3 GameBoardOperation Class

Unlike chess or checker a board representation of TBS is a bit more complex. Nor chess nor checker do not have various terrains, different buildings and units with various game states. Creating a copy of the board is more heavy operation because it holds more information.

However the copy is needed for evaluation of each move in minimax algorithm. A straightforward solution is to make each game operation reversible. It enables to execute an operation in gameboard and then takes it back. Therefore copy of a board is not necessary since undoing previous moves turn board to its origin state.

Key technique for implementing it is called command pattern. The necessary information for executing the operation are stored in an object that also execute the operation. The object can also hold enough information to revert the operation [46]. As a matter of fact, similar principle can be seen in graph-

Listing 3.1: Sample code of command pattern

```
class GameBoardOperation
{
public:
        OperationType operation;

        MyInt moveCost;
        //==========================
        //performer
        CVector2D prevCoorPerf;
        CVector2D newCoorPerf;
        CVector2D coorAttackPerf;
        MyInt prevHpPerf, prevAmmoPerf;
        MyInt newHpPerf, newAmmoPerf;
        MyInt prevFuelPerf, newFuelPerf;

        //==========================
        //defender
        CVector2D prevCoorDef;
        CVector2D newCoorDef;
        CVector2D coorAttackDef;
        MyInt prevHpDef, prevAmmoDef;
        MyInt newHpDef, newAmmoDef;

        //==========================

        MyInt prevCapturePoint1, prevCapturePoint2;
        MyInt newCapturePoint1, newCapturePoint2;
        ...
};
```

ical or text editors that allow users to take back unintentional operations [8]. The actual code of `GameBoardOperation` class is given in Listing 3.1

[8]For graphic editor *Photoshop* it is actualy the opposite. Obsolete version of the editor had only one level undo operation. However *Photoshop* developers could not use the command object for technical reasons in order to implement multiple undos. Instead the editor makes entire copy of graphic document between every new user operation. [47]

3.1.4 MLP1 Class

`MLP1` is a class representing feed-forward neural network with one hidden layer and sigmoid activation function. As neural network is heavily used in game tree search, the implementation gave up on more abstract object–oriented design. Instead, it will use simple plain array storing weights and biases rather than creating layers of abstraction such as class for a neuron. In fact nowadays computers benefit of contiguous memory and data locality rather than strict object-oriented design.[9] The implementation of the network is based on small library TINN (Tiny Neural Network) [48]. TINN saves weights and biases in two separates one–dimensional arrays. Unlike TINN, the implementation is using faster static arrays. However size of a static array needs to be known at a compile time, so that an generic programming is used to reduce the disadvantage. A sample of `MLP1` is given in Listing 3.2.

3.1.5 AIGeneral Class

`AIGeneral` is one of the simplest class in the game. It has a single method used to generate moves for computer player. But how it is done is a concern for the next generation of `AIGeneral` classes. Which is why the class is very simple.

One of the possible implementation is provided by `AI2` generic class. It implements minimax algorithm. Although the class does not take responsibility for actual execution of moves. It is done in other class called `AIComponent`. `AIComponent` simulates player moves. At first glance the hierarchy seems to be complicated. On the other hand, it offers better reusability of code. Reason for simulating moves indirectly in bottom classes is more general abstraction. Hierarchy of the classes is shown in Figure 3.2.

For instance smart `AIComponent` can recompute only part of the board since it controls possible game moves. Imperfect yet simple `AIComponent` recomputes an entire game board and not taking individual move into consideration. At last game board is evaluated with `GameBoardHeuristic` class.

3.2 Architecture of the Optimization Algorithms

3.2.1 Classic Optimization Algorithms

Since simulated annealing and hill climbing are at their core almost identical, a shared abstract class `OptimizatorInterface` was created for them. The

[9]Modern processors are much faster in processing data than retrieving that data from memory. Without using caches and loading data ahead processor would wait for needed data most of its time. [49]

Listing 3.2: A sample code of neural network

```
template<class Type, size_t nIn, size_t nHid, size_t nOut>
class MLP1
{
public:
        static constexpr size_t nInputs = nIn;
        static constexpr size_t nHiddens = nHid;
        static constexpr size_t nOutputs = nOut;
        static constexpr size_t nWeights = nHiddens * (
    nInputs + nOutputs);
        static constexpr size_t nBiases = nHiddens+nOutputs;

        static constexpr size_t weightIndexOffset = nHiddens
     * nInputs;
        static constexpr size_t biasIndexOffset = nHiddens;

        using wArray = std::array<Type, nWeights>;
        using bArray = std::array<Type, nBiases>;
        using NumberType = Type;

        std::array<Type, nWeights> weights;
        std::array<Type, nBiases> biases;

        std::array<Type, nHiddens> hiddenNeurons;
        std::array<Type, nOut> outputNeurons;
        ...
};
```

class needs to be generic in order to support any type of neural network from `MLP1` class.

`OptimizatorBase` inherits from `OptimizatorInterface`. The class specifies how to create a neighbour solution in more detail. Finally concrete classes inherit from it and implement in their own way how new solution will be accepted. General diagram of those classes is shown in Figure 3.3.

To allow better abstraction, the class includes a generic type for computing fitness. As a result the optimization algorithms can be used on different kinds of problems by implementing only different kinds of fitness functions.

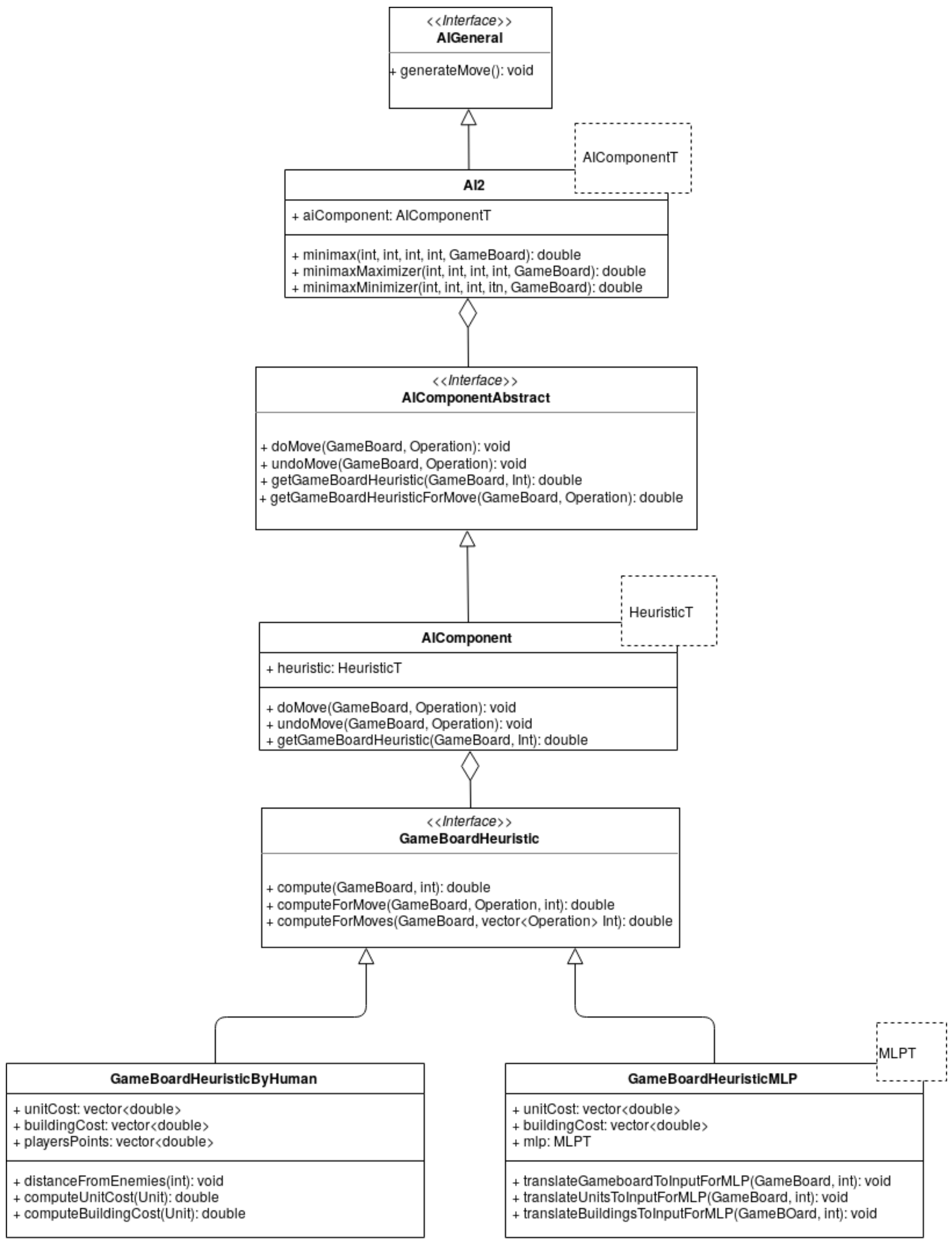

<<Interface>>
AIGeneral
+ generateMove(): void
AIComponentT
AI2
+ aiComponent: AIComponentT
+ minimax(int, int, int, int, GameBoard): double
+ minimaxMaximizer(int, int, int, int, GameBoard): double
+ minimaxMinimizer(int, int, int, itn, GameBoard): double
<<Interface>>
AIComponentAbstract
+ doMove(GameBoard, Operation): void
+ undoMove(GameBoard, Operation): void
+ getGameBoardHeuristic(GameBoard, Int): double
+ getGameBoardHeuristicForMove(GameBoard, Operation): double
HeuristicT
AIComponent
+ heuristic: HeuristicT
+ doMove(GameBoard, Operation): void
+ undoMove(GameBoard, Operation): void
+ getGameBoardHeuristic(GameBoard, Int): double
<<Interface>>
GameBoardHeuristic
+ compute(GameBoard, int): double
+ computeForMove(GameBoard, Operation, int): double
+ computeForMoves(GameBoard, vector<Operation> Int): double
GameBoardHeuristicByHuman
+ unitCost: vector<double>
+ buildingCost: vector<double>
+ playersPoints: vector<double>
+ distanceFromEnemies(int): void
+ computeUnitCost(Unit): double
+ computeBuildingCost(Unit): double
MLPT
GameBoardHeuristicMLP
+ unitCost: vector<double>
+ buildingCost: vector<double>
+ mlp: MLPT
+ translateGameboardToInputForMLP(GameBoard, int): void
+ translateUnitsToInputForMLP(GameBoard, int): void
+ translateBuildingsToInputForMLP(GameBOard, int): void

Figure 3.2: Diagram of AI hierarchy

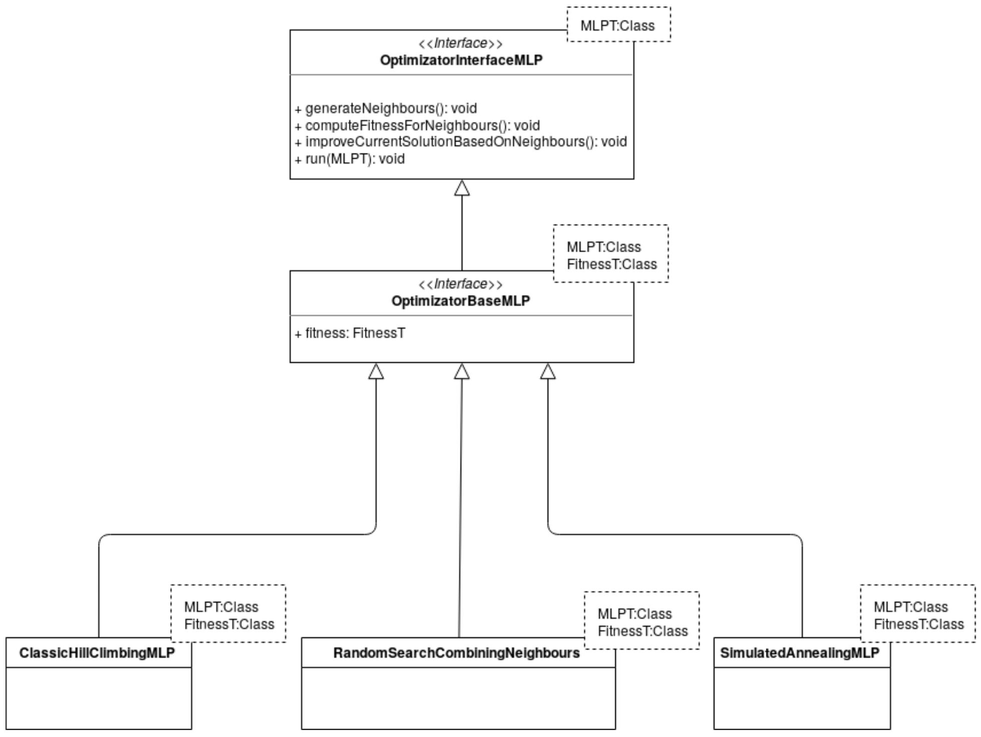


Figure 3.3: Diagram of classes for local optimization algorithms

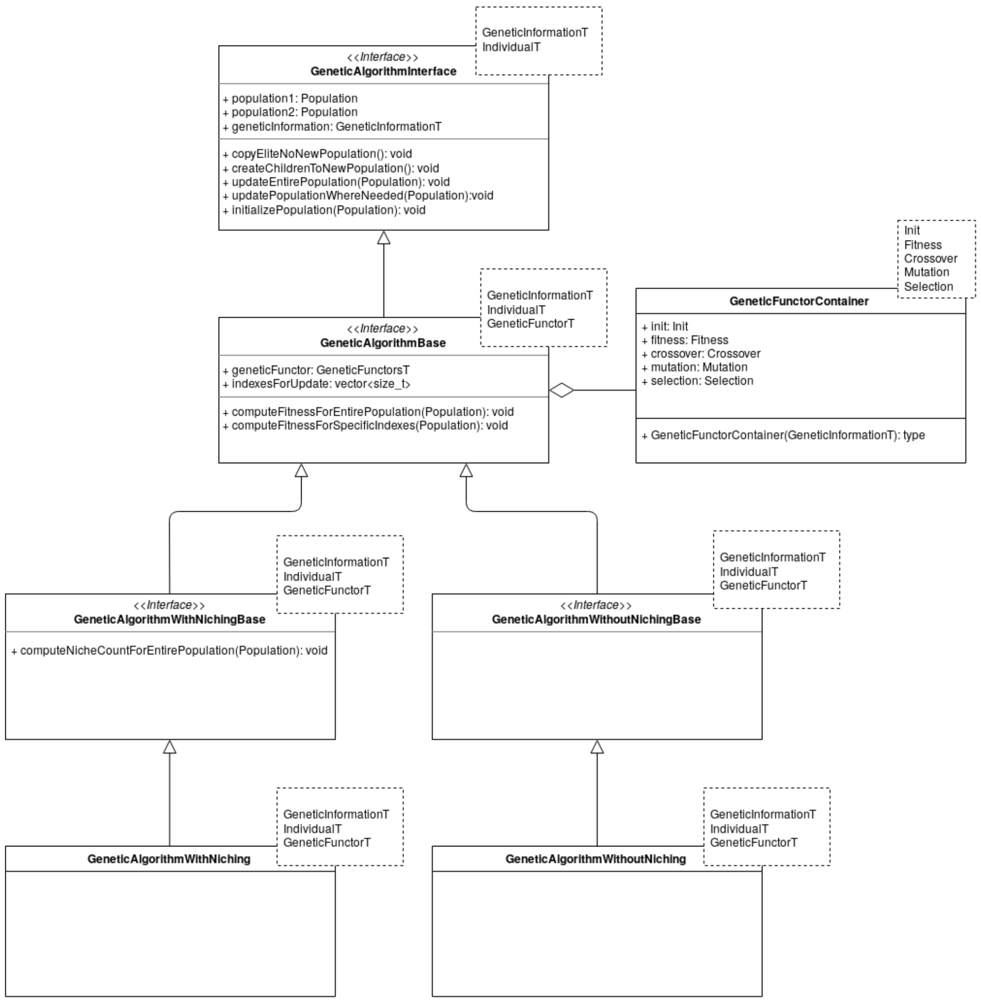

GeneticInformationT
IndividualT
<<Interface>>
GeneticAlgorithmInterface
+ population1: Population
+ population2: Population
+ geneticInformation: GeneticInformationT
+ copyEliteNoNewPopulation(): void
+ createChildrenToNewPopulation(): void
+ updateEntirePopulation(Population): void
+ updatePopulationWhereNeeded(Population):void
+ initializePopulation(Population): void
Init
Fitness
Crossover
Mutation
Selection
GeneticFunctorContainer
+ init: Init
+ fitness: Fitness
+ crossover: Crossover
+ mutation: Mutation
+ selection: Selection
+ GeneticFunctorContainer(GeneticInformationT): type
GeneticInformationT
IndividualT
GeneticFunctorT
<<Interface>>
GeneticAlgorithmBase
+ geneticFunctor: GeneticFunctorsT
+ indexesForUpdate: vector<size_t>
+ computeFitnessForEntirePopulation(Population): void
+ computeFitnessForSpecificIndexes(Population): void
GeneticInformationT
IndividualT
GeneticFunctorT
<<Interface>>
GeneticAlgorithmWithNichingBase
+ computeNicheCountForEntirePopulation(Population): void
GeneticInformationT
IndividualT
GeneticFunctorT
<<Interface>>
GeneticAlgorithmWithoutNichingBase
GeneticInformationT
IndividualT
GeneticFunctorT
GeneticAlgorithmWithNiching
GeneticInformationT
IndividualT
GeneticFunctorT
GeneticAlgorithmWithoutNiching

Figure 3.4: Diagram of classes for genetic algorithms

3.2.2 Genetic Algorithm

Genetic algorithms were implemented in similar manner as hill climbing and simulated annealing. Main idea is to separate genetic operators from genetic algorithm in order to reuse different kinds of genetic operators in different kinds of genetic algorithms. Genetic operators are held in generic class `GeneticFunctorContainer` that enable to take arbitrary genetic operators. Components of the genetic algorithm can be seen in Figure 3.4.

Genetic algorithm and genetic operators are initialized by generic type `GeneticInformationT`. Genetic information contains information about percentage of crossovers and mutations, size of an population and other information necessary for launching genetic algorithm.

Chapter 4

Testing and Measurement

This chapter shows an experimental evaluation of the implemented optimization algorithms. The first measurement serves as a test deciding what algorithm is going to be used for learning the game. The measurements demonstrate differences between various optimization parameters and their impact on measurement score. After that, the best-suited algorithm is selected according to the measurements. To begin with the actual learning algorithm, structure of ANN and fitness function are defined. Lastly, the final result is assessed and the quality of the machine, as well as human created AI, is tested in a real play. All tests run on *Intel Core i3–3220* with 2 cores.

4.1 Testing Optimization Algorithms With MNIST Data Set

MNIST (Modified National Institute of Standards and Technology) data set is a database of handwritten digits. Every digit image corresponds to a label telling what number is represented by the image. The image size is 28×28. The database is split into training set of 60 000 examples and test set containing 10 000 examples. It is a convenient database for testing pattern recognition technique since the data required only minimal effort on processing and formatting. [50]

`MnistReader` class was created so that the data can be loaded properly into feed-forward neural network. All optimizations were tested with the first 8000 digits from training set.

4.1.1 Structure of Neural Network

Every implemented optimization uses feed-forward neural network with immutable structure changing only weights and biases. The structure consists of one hidden layer with 15 neurons. The input layer has $28 \times 28 = 784$

neurons. Output layer consists of 10 neurons. There is an output neuron for every digit. For instance, the first neuron represents number zero, the second neuron represents number one, etc. Output neuron firing the most express neural network prediction. The structure is sufficient to reach 95% accuracy of classifying images with stochastic gradient descent [21]. Reason for not using more layers and neurons is simplicity and easier optimization of parameters. On the other hand there are still $784 \times 15 + 15 \times 10 = 11\,910$ weight parameters to tweak.

4.1.2 Fitness

The algorithms use two different fitnes metrics. The first fitness computes only percentage of correct classified images. The second one uses MSE described in Chapter 1.3.4.4. MSE is able to tell difference how bad is a neural network digit prediction from its expected output. The first fitness cannot do that since it only computes final result for the entire set. Algorithms using the first fitness has the suffix LowSense and algorithms using the second fitness with MSE has the suffix HighSense.

Even though the first fitness is much worse than the second one, it plays a key role for improvement of the algorithm learning the game with minimal human input. The problem is that there is no clear way for creating expected output for evaluation of different gameboard situation. Thus MSE cannot be computed without the expected output.

4.1.3 Genetic Algorithm Testing

Following operators were implemented for the genetic algorithm: rank-based selection, uniform crossover, one-point crossover, mutation with zeroing parameters out of specified bounds, mutation without zeroing and elitism mechanism, initialize function using normal distribution. All implemented genetic algorithms use the uniform crossover since the one-point crossover does not seem to make any difference in results.

As can be seen in graph in Figure 4.1, the genetic algorithm without zeroing quickly converges to a local maxima around 40 percent of correct classified images. There is only a small improvement in the fitness since 20 minutes. The genetic algorithm with zeroing converges a little slowly. On the other hand, it reached worse results showing minimal improvements after 80 minutes. Detailed parameter settings can be seen in Listings 4.1.

4.1.4 Hill Climbing and Simulated Annealing

Due to initial inefficiency of genetic algorithm, hill climbing and simulated annealing were implemented as well. However the local search algorithms perform even worse as ilustrated in graph in Figure 4.2.

Listing 4.1: Parameter Settings 1

```
./GA_MNIST_LowSense_CrossUni
dirName=GA_MNIST_LowSense_CrossUni_NoNiching endIteration
    =5000000 endTimeInMinutes=120 outputFrequency=150
    popSize=10 elitismSize=3 functorMutRate=0.3 crossRate
    =0.15 mutRate=0.3 mutWeightsBound=0.01 mutBiasBound=0.01

./GA_MNIST_LowSense_CrossUniWithReseting
dirName=GA_MNIST_LowSense_CrossUni_NoNiching_NoReseting
    endIteration=5000000 endTimeInMinutes=120
    outputFrequency=150 popSize=10 elitismSize=3
    functorMutRate=0.3 crossRate=0.15 mutRate=0.3
    mutWeightsBound=0.01 mutBiasBound=0.01
```

Listing 4.2: Parameter Settings 2

```
./HillClimb_HighSense dirName=HillClimbTestHighSense
    endIteration=999999 endTimeInMinutes=60 outputFrequency
    =2000

./HillClimb_LowSense dirName=HillClimbTestLowSense02
    endIteration=999999 endTimeInMinutes=60 outputFrequency
    =2000 mutWeightsBound=0.05 mutBiasBound=0.05

./SimulatedAnnealing_LowSense endIteration=999999 dirName=
    SALowSense01 endTimeInMinutes=60 outputFrequency=2000
    mutWeightsBound=0.05 mutBiasBound=0.05
```

Hill climbing with low sense converges even faster than previous genetic algorithms. Even though simulated annealing converged quite slowly, it was not able to reach even 0.4 limit.

See also hill climbing with high sense using MSE fitness in Figure 4.2. It is dramatically better compared to the other algorithms. It reached almost 80% accuracy on trained data after 10 minutes. Moreover it is exactly the same algorithm as hill climbing with low sense. The only difference is the used fitness function. Parameter settings for this measurement can be seen in Listings 4.2.

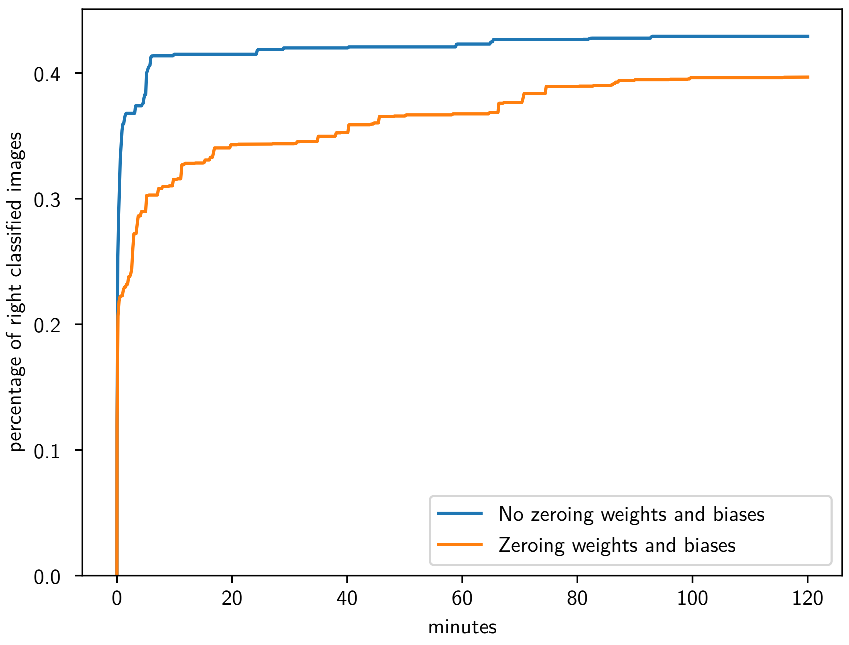


Figure 4.1: Genetic algorithms tested on MNIST dataset.

4.1.5 Genetic Algorithm With Fitness Sharing

The problem with genetic algorithm is premature convergence. It appears that genetic algorithm population differ too little in order to come up with new inividuals via genetic operators.

One possible solution is dynamically change rate of mutation trying to make population more various and possibly escaping from local maxima. The technique would be quite similar to simulated annealing temperature. However the problem is to come up with efficient temperature schedule since in graph in Figure 4.2 simulated annealing was outperformed by even simpler hill climbing. Another possible solution is to use niching method, more specifically fitness sharing described in Chapter 1.3.6.2. To avoid possible problems with defining dynamic mutation schedule a fitness sharing was implemented instead. Euclidean distance for genetic algorithm individuals was used. The value was set to 320 empirically.

Using the fitness sharing method, the genetic algorithm obtained dramatically better result. As illustrated by graph in Figure 4.3, the fitness from genetic algorithm with niching slightly oscilates, unlike hill climbing or genetic algorithm without niching showing quite steady lines. That is the very fitness sharing method causing the oscilation. When there is a drop in a fit-

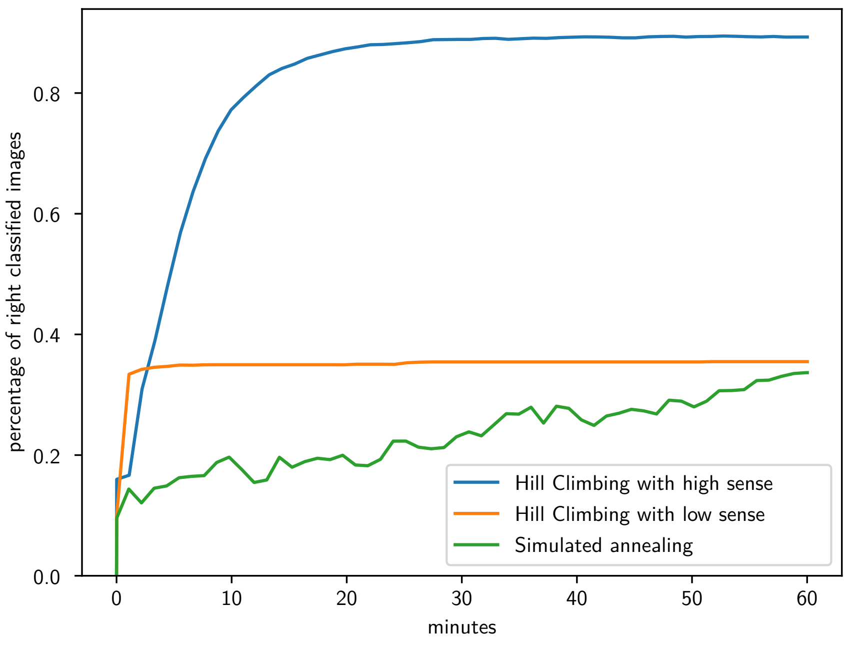


Figure 4.2: Hill climbing and simulated annealing tested on MNIST dataset.

ness, more individuals share the same fitness. Then they can be more replaced by different individuals even with smaller fitness.

After 120 minutes, genetic algorithm with niching was able to reach almost 90 percent of accuracy. On the other hand, hill climbing with MSE fitness still outperforms that. To see parameter settings for this test navigate Listings 4.3.

4.1.6 Results

The results suggest that the optimization algorithms are heavily affected by their fitness functions. With appropriate fitness function, even simple algorithm like hill climbing can be used to train neural networks and gain reasonable results. Genetic algorithm without niching methods works similarly to hill climbing because of premature convergence. It makes genetic population not much diverse, and therefore genetic algorithm searches only narrow local space just like hill climbing does. Using more advance algorithms such as simulated annealing is inefficient without good temperature schedule.

Table 4.1 presents accuracy of the all implemented algorithms on the entire testing and training MNIST data set. The algorithms do not overfit training data since testing and training accuracy differs only in a few percent.

Listing 4.3: Parameter Settings 3

```
./GA_MNIST_LowSense_CrossUni
dirName=GA_MNIST_LowSense_CrossUni_Niching endIteration
    =5000000 endTimeInMinutes=120 outputFrequency=150
    popSize=10 elitismSize=3 functorMutRate=0.3 crossRate
    =0.15 mutRate=0.3 mutWeightsBound=0.01 mutBiasBound=0.01
     niching=320

./GA_MNIST_LowSense_CrossUni
dirName=GA_MNIST_LowSense_CrossUni_NoNiching_1 endIteration
    =5000000 endTimeInMinutes=120 outputFrequency=150
    popSize=10 elitismSize=3 functorMutRate=0.3 crossRate
    =0.15 mutRate=0.3 mutWeightsBound=0.01 mutBiasBound=0.01

./HillClimb_HighSense dirName=HillClimbTestHighSense
    endIteration=999999 endTimeInMinutes=60 outputFrequency
    =2000
```

Algorithm	**Testing Data Set**	**Training Data Set**
Simulated Annealing Low Sense	34.21 %	35.27 %
Hill Climbing Low Sense	35.58 %	35.88 %
Genetic Algorithm With Zeroing	38.78 %	39.68 %
Genetic Algorithm Without Zeroing	40.63 %	42.93%
Genetic Algorithm With Niching	84.64 %	87.63 %
Hill Climbing High Sense	90 %	94.91 %

Table 4.1: Accuracy of the algorithms on the entire MNIST dataset.

4.2 Learning Artificial Inteligence with Minimum Human Input

4.2.1 Fitness

Since there are no expected outputs for an evaluation of a gameboard, the MSE metric cannot be used for fitness function. Thus fitness will be represented by points earning by players in each played game. Player can gain 0, 1, and 2 points for a draw, loss and win respectively. Every player will compete in 4 matches with 4 different players that are randomly selected from population.

4.2.2 Structure of Neural Network

An individual neural network will be trained for every individual new game map. As a result, an ANN does not have to generalize for every possible map,

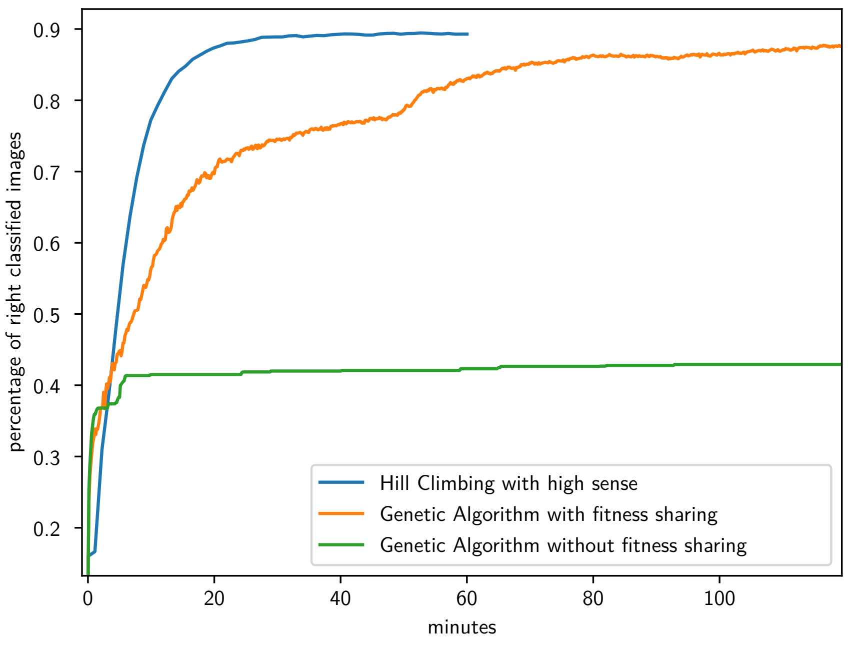


Figure 4.3: Comparison between genetic algorithm with niching and without.

thus it can consist of less neurons. It makes optimization of weights and biases easier as well.

Single neuron input will represent single map tile with unit or single tile with building. Hence we need 2 input neurons for a map tile. Input representing tile with unit will be computed as $unitHealth \times unitCost$. Unit cost is listed in Table 4.2. Input representing tile with building will be computed as $playersCapturePoints \times buildingCost$ playersCapturePoints * buildingCost. Building cost can be seen in Table 4.3. The results will be multiplied with -1 if unit or building belongs to an enemy player. If there are no units and no buildings in a tile, the computed input will be simply zero.

Input layer contains $numberOfMapTiles \times 2$ neurons. Hidden layer has 30 neurons for maps with 150 tiles. Hidden neurons will be connected to single output neuron. The bigger the value of the output neuron, the more promising board for the player. The neural network consists of $300 \times 30 + 30 \times 1 = 9030$ weights to tweak. There are no special reasons for choosing 30 neurons in the hidden layer. The current settings are rather based on intuition. The task seems to be more challenging than training neural network with MNIST, thus more hidden neurons are probably needed. On the other hand, a hidden layer, which is too big, might complicate the optimization by too many parameters.

Unit Type	**Unit Cost**
Infantry	0.1
Mech	0.3
Light Tank	0.7
Artillery	0.6
AntiAirTank	0.8
Battle Copter	0.85

Table 4.2: Unit cost table.

Building Type	**Building Cost**
City	0.3
Airport	0.3
Headquarters	1

Table 4.3: Building cost table.

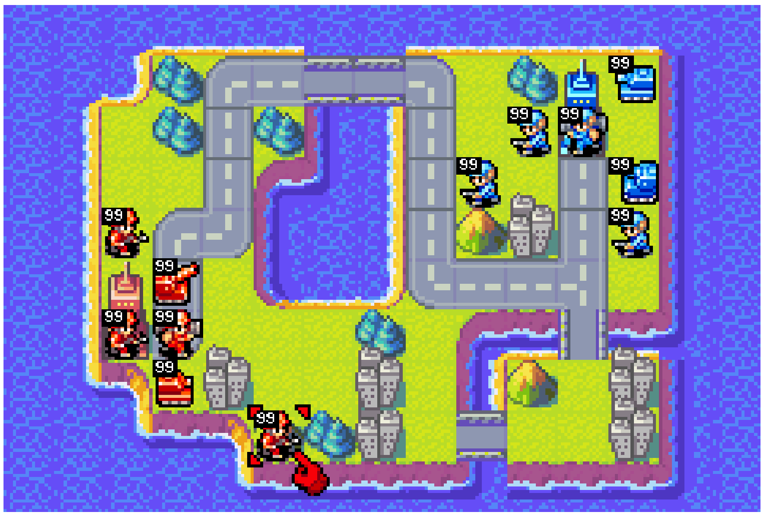


Figure 4.4: The game map used in fitness function.

4.2.3 Genetic Algorithm

Since MSE cannot be applied, genetic algorithm with fitness sharing will be used. Genetic algorithm with niching method proved to gain the best result for MNIST data set without the MSE. It will be set with similar genetic operators and parameters. The only difference will be execution time. Genetic algorithm will be tested on a map given in Figure 4.4.

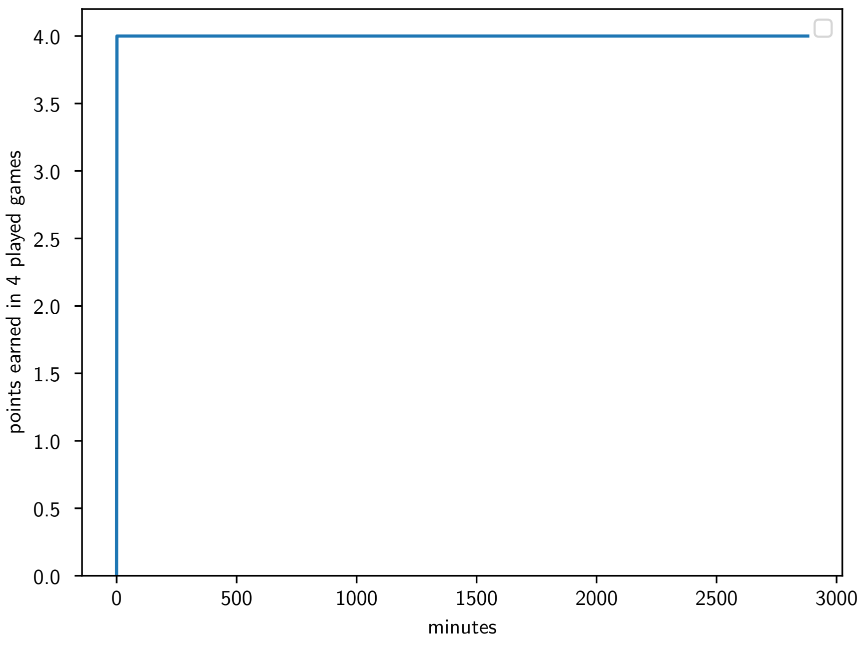


Figure 4.5: Genetic algorithm simulating game tournaments for over 40 hours.

4.2.4 Result

In spite of initial effort to improve genetic algorithm in general, it is unable to make even a small improvement, as shown in graph in Figure 4.5. Every population individual got fitness equal 4 for each iteration. It means all played games end up in draw. Draw is set if any player does not win and number of turns exceed 20.

Changing population size, mutation rate, cross over rate does not seem to make any difference. Moreover setting bigger population or turn game limit slows down an already lengthy algorithm. For instance, genetic algorithm solving MNIST data set go over 3000 iteration, or generation after 3 minutes. On the other hand genetic algorithm solving the game simulation go roughly over 13 iteration after 3 minutes. The second algorithm is too slow because of time consuming fitness computation simulating small tournaments for the entire population.

There could be plenty of reasons why the learning algorithm failed. A straightforward argument is too slow evolution. Another reason could be quality of fitness function that does not satisfy enough the genetic algorithm. The same problem can be seen while training a neural network on MNIST data set. Another reason excluding the genetic algorithm is the structure of a neural network. The structure could be too simple to learn playing the game. Number of neurons could just limit a neural network to play the game reasonably. Even if the number of neurons was sufficient, the representation of the board for neural network might be wrong. Finally training neural network on MNIST data set and learning it to play game are fairly different problems. Parameters like mutation rate, population size or type of genetic operators can work on one problem efficiently but it could be completely inefficient for the other. However trying sets of different genetic operators and parameters is challenging since the genetic algorithm runs quite slowly.

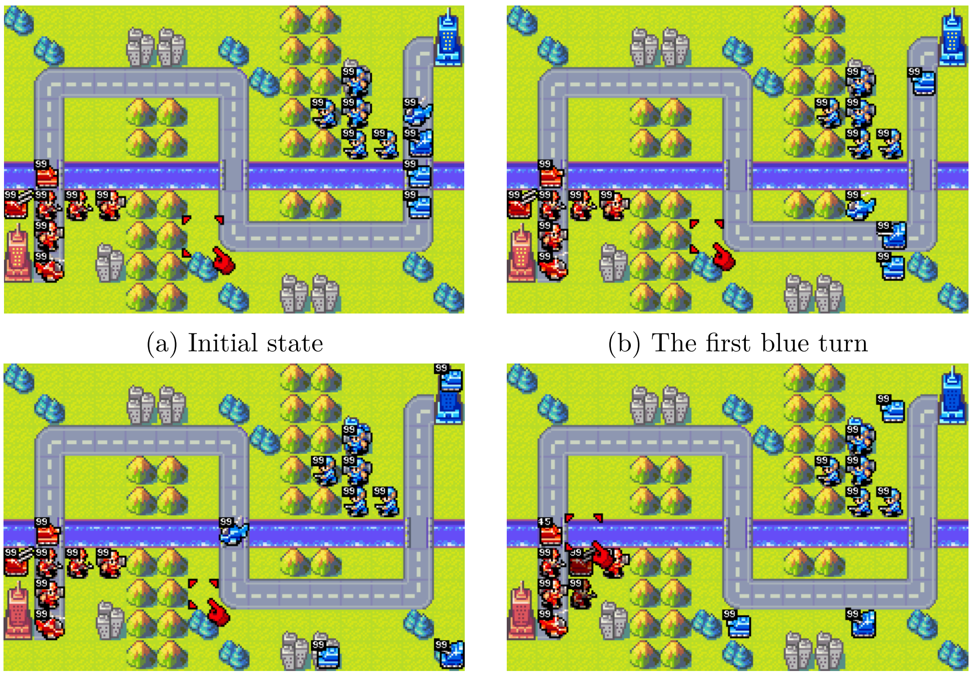
4.3 Quality of the Game Played by Artificial Inteligence

4.3.1 Heuristic Created By Human

The first implemented heuristic takes into consideration only values of individual units and buildings. Due to a big branching factor and shallow minimax, the computer player is not able to move with its units almost anywhere and they are staying still. The player is only keen on moving units with wider range such as helicopter and tanks. The result suggests that minimax with the heuristic has a similar flaw as hill climbing. Since minimax is shallow it searches only a small portion of a map. Which is why only units with wide range are able to take some actions. In order to find a better game position, the algorithm needs to look deeper into the game tree or has more advanced heuristic. Unfortunately the branching factor is too high and it is computationaly very difficult to look more moves ahead. Therefore the effort is going to be put into better heuristic.

The second implemented heuristic uses cost of individual units and buildings as well. In addition, it takes into consideration distances between units. It computes a centroid [10] for all units in a same team. Smaller distance between centroids means more promising game state for a player. This strategy forces player to take at least some actions. It is important that significance of distances does not overrule the rest of the heuristic. For instance, it is usually safer and more beneficial to capture nearby buildings than draw enemy closer at all costs. With the second heuristic computer player makes moves with all its units. Even though the heuristic does not use very advanced technique, it definitely helps improving the minimax result. As can be seen in Figure 4.7, the second heuristic is able to make use of infantry unit to capture nearby buildings. Using centroid metric tends to make unit more closer between each other, which can be convenient rather than attack with individual units and

[10]The centroid is the arithmetic mean of points. For a finite set of k points $x_1, x_2, \dots, x_k$, it is defined as: $\frac{1}{k}\sum_{n=1}^{k} x_n$ [51].



(a) Initial state

(b) The first blue turn

(c) The second blue turn

(d) The third blue turn

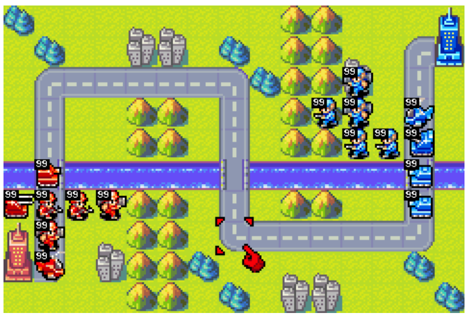
Figure 4.6: Red human player vs. blue computer player considering cost of units and buildings

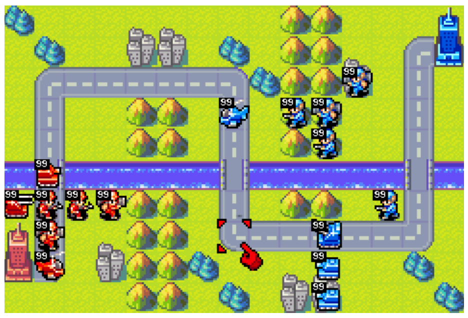
split unit power. The observation is not absolute since the blue player lost unnecessarily the helicopter in Figure 4.6 and Figure 4.7.

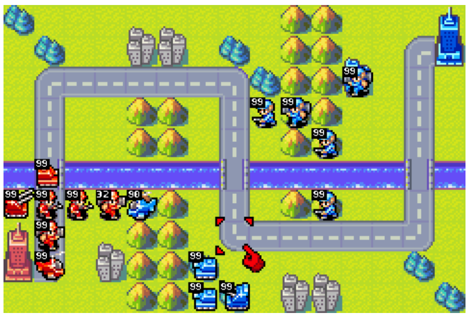
A poor characteristic was discovered in the second heuristic while testing. If the game reaches a certain state where all centroids met in the same coordinates, the second heuristics degrades into the first implemented heuristic. Moreover computer player tends to stay in the position since it is a best option according the metric based on centroids. The situation is illustrated by Figure 4.8.

4.3.2 Heuristic Created By Machine

A computer player using the heuristic seems to play randomly rather than making reasonable moves on the basis of some patterns. The player does make a move toward capturing a building as shown in Figure 4.9. However, it is not able to take it. A game with both neural network players hints a possible reason why the implemented genetic algorithm was inefficient. The players are rearranging their units and after several turns, they are stuck and unwilling to make a different move. Which is probably why every game simulation ends with an equal score in the genetic algorithm.


(a) Initial state


(b) The first blue turn


(c) The second blue turn

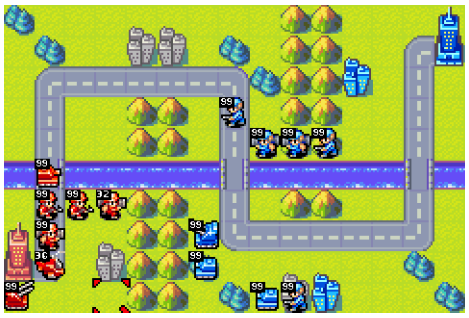

(d) The third blue turn

Figure 4.7: Red human player vs. blue computer player considering distances among units

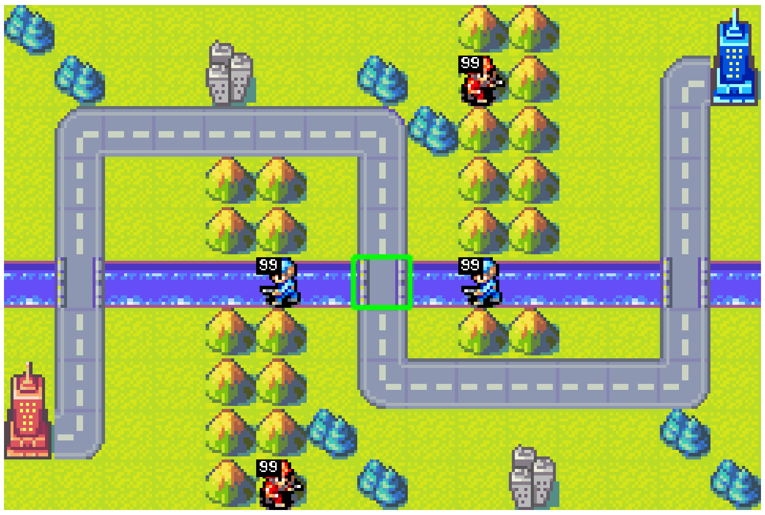
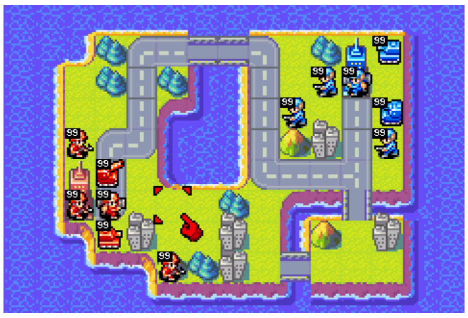
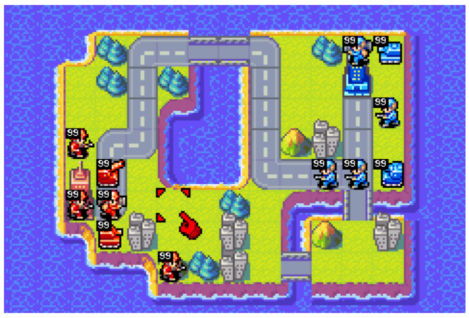
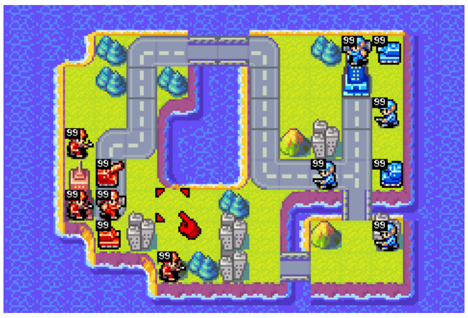


Figure 4.8: Green rectangle represent position of centroid for both players


(a) Initial state


(b) The first blue turn


(c) The second blue turn

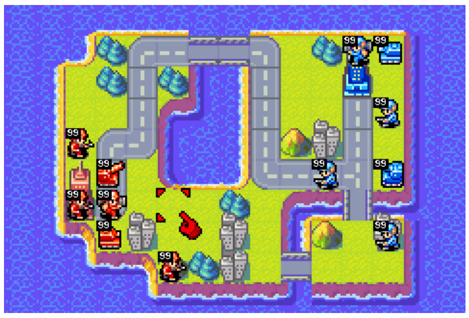

(d) The third blue turn

Figure 4.9: Red human player vs. blue computer player using a neural network.

Conclusion

One of the objectives of the thesis was creating a prototype of a TBS game with some given features. The final implemented prototype met all the requirements. The game can be played by human as well as computer players. There are 4 different maps to be played. Moreover, new maps and even units can be added by modifying several json files rather than writing C++ code. Analysis of several TBS game was made as well to create a game based on common features of strategies. The source code was written with some compromises. The code uses object-oriented design as well as C++ template programming and data-oriented design. Thus code is somewhat trade-off between understandability and performance. Also, it makes use of design patterns such as command, game loop or flyweight. On the other hand, they are not overused and they make sense of their usage.

The second goal was creating an AI playing the game with minimum human input, including a research part of different algorithms. Unfortunately, the goal was fulfilled only partially. The thesis looks into several optimization algorithms and covers the basic theoretical background required for creating the AI. It also studies two different AI programs playing turn-based games making use of evolutionary algorithms. Afterward, some of the algorithms described in the research part were implemented with emphasis on abstraction, reusability, and extension. Then the algorithms were tested with MNIST data set using a mainly poor fitness function. Based on the experimental evaluation the genetic algorithm was chosen to be used as an AI learning algorithm. Furthermore, it was improved using one of the niching methods. The final genetic algorithm was teaching AI to play games via self-play. Apart from that, there are several AI programs, that are manmade. Last but not least, the quality of all AI programs was tested both by man as well as the computer itself. Unfortunately, the program trained via self-play was unable to play games efficiently since its moves look rather randomly.

Future Work

The implemented game prototype can be improved in several ways. There could be some sound effects since the current version lacks them. The GUI (Graphical User Interface) might be more interactive rather than using simple dark boxes. On the other hand, dark boxes serve as a decent retro design. More animation would make a better player experience as well as a mouse controller. From the game mechanics perspective, buildings are currently used for supplying and healing. Expansion using buildings as factories for tanks and infantry could create some new interesting game situations.

There are many approaches on how to fix the learning algorithm. At first, testing the genetic algorithm with another optimization task apart from MNIST data set might help to find a reasonable setting for further genetic algorithm improvement. Also, the genetic algorithm could be improved by implementing new operators. For instance, current crossover operators do not take into consideration the structure of a neural network. It could benefit from creating a genetic operator using weights and biases based on neurons. The current crossover simply selects weights from an array.

Another improvement could be changing the properties of the operators throughout generation rather than using static parameters. To partially solve the problem with the speed of the genetic algorithm caused mainly by the fitness function, it could be implemented using multithreading. As a result, every player would compete against the others in their thread. On the other hand, it would require at least a processor with four cores rather than the processor with two cores in order to get some noticeable speedup. Apart from that, the neural network with different numbers of neurons could be trained. Secondly, changing neural network input might help too. One possible way would be translating only some significant parts of the game map as a neural network input rather than taking it entirely [52].

Also, focussing purely on fitness might work. As shown in Chapter 4.3.2 the fitness simulating the game always ends in a tie. Simulating an entire game seems to be problematic, thus simulations for only some parts of a game could break endings in a tie. One of the examples is shown in Figure 4.10. There is only one way how the red player can win. He needs to capture the enemy base since the blue player possesses more units and a tank. Furthermore, he needs to block the blue tank in front of the bridge. Unlike simulating the entire game, there is only one straightforward way to solve a map in Figure 4.10. On the other hand, this solution would require creating many different game situations in order to evolve a neural network that generalizes from the genetic algorithm training.

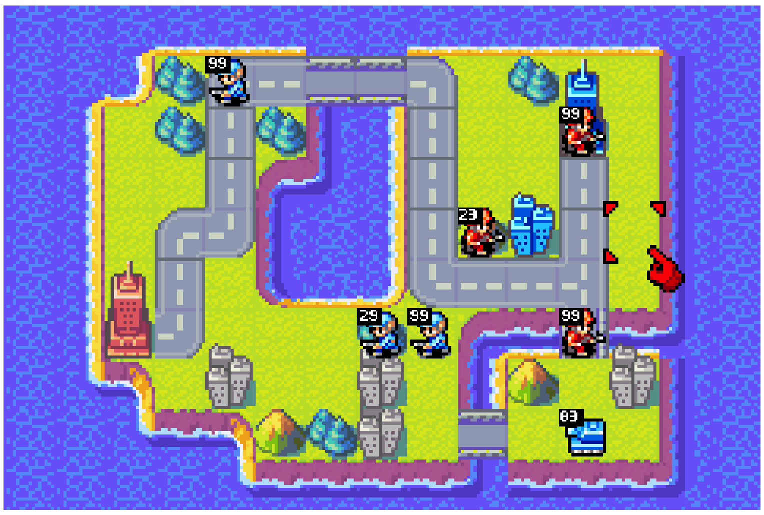


Figure 4.10: Game situation narrowing number of reasonable choices.

Bibliography

[1] The Entertainment Software Association: *2019 Essential Facts About the Computer and Video Game Industry* [online]. 2019, [cit. 2019-11-23]. Available at: `https://www.theesa.com/wp-content/uploads/2019/05/ESA_Essential_facts_2019_final.pdf`

[2] Historie vývoje počítačových her. In: *Root.cz – informace nejen ze světa Linuxu* [online] [cit. 2019-11-23]. `https://www.root.cz/clanky/historie-vyvoje-pocitacovych-her-1-cast-prvni-milniky/`

[3] Shannon, C. E. *Programming a computer for playing chess.* In: *Philosophical Magazine.* 1950, p. 256–275.

[4] Bonanno Giacomo *Game Theory* [online]. California, 2018 [2019-12-09]. `http://faculty.econ.ucdavis.edu/faculty/bonanno/GT_Book.html`

[5] Coon, Dennis, et al. *Introduction to Psychology: Gateways to Mind and Behavior.* Cambridge, Belmont, Calif: Wadsworth Cengage Learning, 2010. ISBN 978-0495599111

[6] David C. Pottinger *Terrain Analysis in Realtime Strategy Games.* In Game Developers Conference Proceedings, 2000.

[7] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford *Introduction to Algorithms.* Cambridge, Mass: MIT Press, 2009. ISBN 978-0-262-03384-8

[8] Russell, Stuart J.; Norvig Peter; Ernest Davis *Artificial intelligence: a modern approach.* Upper Saddle River, New Jersey: Prentice Hall, 2010. ISBN 978-0-13-604259-4

[9] Kenedy, James; Russel, Eberhart C.; Shi, Yuhui *Swarm Intelligence.* San Francisco, Morgan Kaufmann Publishers, 2001. ISBN 1-55860-595-9

[10] Manhattan distance. In: *Dictionary of Algorithms and Data Structures* [online] [cit. 2019-12-14]. `https://xlinux.nist.gov/dads/HTML/manhattanDistance.html`

[11] Řehořek, Tomáš. *Základy umělé inteligence: Automatické plánování* [online]. 2016 [cit. 2019-11-23]. `https://courses.fit.cvut.cz/BI-ZUM/@B172/media/lectures/06-planning-v5.0-anim.pdf` [Accessed from CVUT network after login]

[12] Řehořek, Tomáš. *Základy umělé inteligence: Hry v extenzivní formě, Algoritmus Minimax* [online]. 2016 [cit. 2019-11-23]. `https://courses.fit.cvut.cz/BI-ZUM/@B172/media/lectures/09-minimax-v5.0-noanim.pdf` [Accessed from CVUT network after login]

[13] LeCun Yann; Bottou Leon; Orr, Genevieve B.; Müller Klaus-Robert. *Efficient BackProp.* In: *Neural Networks: tricks of the trade.* Berlin: Springer, 1998, p. 9–50. ISBN 978-3-540-49430-0 `http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf`

[14] Devansh Arpit; Stanisław Jastrzebski; Nicolas Ballas; David Krueger; Emmanuel Bengio; Maxinder S. Kanwal; Tegan Maharaj; Asja Fischer; Aaron Courville; Yoshua Bengio; Simon Lacoste-Julien. *A Closer Look at Memorization in Deep Networks.* 2017 [online] [cit. 2019-12-14] `https://arxiv.org/pdf/1706.05394.pdf`

[15] Chiyuan Zhang; Samy Bengio; Moritz Hardt; Benjamin Recht; Oriol Vinyals. *Understanding Deep Learning Requires Re-thinking Generalization.* 2017 [online] [cit. 2019-12-14] `https://arxiv.org/pdf/1611.03530.pdf`

[16] Understanding Activation Functions in Depth. In: *GeeksForGeeks: A computer science portal for geeks* [online] [cit. 2019-12-14]. `https://www.geeksforgeeks.org/understanding-activation-functions-in-depth/`

[17] The Neural Network Zoo - The Asimov Institute. [online]. [cit. 2019-12-14]. `https://www.asimovinstitute.org/neural-network-zoo/`

[18] Partial derivative. *Wikipedia, The Free Encyclopedia* [online]. 2019 [cit. 2019-12-14]. `https://en.wikipedia.org/wiki/Partial_derivative`

[19] Sanderson Grant. Backpropagation calculus. In: *Youtube* [online] [cit. 2019-12-14]. `https://www.youtube.com/watch?v=tIeHLnjs5U8`. Channel of user 3Blue1Brown.

[20] Goodfellow Ian; Bengio Yoshua; Courville Aaron. *Deep Learning* [online]. 2016 [cit. 2019-12-14]. `http://www.deeplearningbook.org/`

[21] Nielsen, Michael A. *Neural Networks and Deep Learning* [online]. 2016 [cit. 2019-12-14]. http://neuralnetworksanddeeplearning.com/chap1.html

[22] Řehořek, Tomáš. *Základy umělé inteligence: Algoritmy iterativní optimalizace, Populačníı metody* [online]. 2016 [cit. 2019-12-14]. https://courses.fit.cvut.cz/BI-ZUM/@B172/media/lectures/03-optimization-v5.0-anim.pdf [Accessed from CVUT network after login]

[23] Khemani Deepak. Optimization - I (Simulated Annealing). In: *Youtube* [online] [cit. 2019-12-14]. https://www.youtube.com/watch?v=tIeHLnjs5U8. Channel of user nptelhrd.

[24] A Comparison of Cooling Schedules for Simulated Annealing (Artificial Intelligence) In: *what-when-how – In Depth Tutorials and Information* [online] [cit. 2019-12-14]. http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/

[25] Engelbrecht, Andries P. *Computational Intelligence: An Introduction.* Wiley Publishing, 2007. ISBN 978-0470035610

[26] Řehořek, Tomáš. *Základy umělé inteligence: Evoluční výpočetní techniky, Genetický algoritmus, Genetické programování* [online]. 2016 [cit. 2019-12-14]. https://courses.fit.cvut.cz/BI-ZUM/@B172/media/lectures/04-evolution-v5.0.pdf [Accessed from CVUT network after login]

[27] Sareni Bruno; Krähenbühl Laurent. *Fitness Sharing and Niching Methods Revisited.* In: *IEEE Transactions on Evolutionary Computation.* Berlin: Springer, 1998, p. 97–106. https://doi.org/10.1109/4235.735432

[28] 15 Real-World Applications of Genetic Algorithms. In: *Brainz – Learn Something* [online] [cit. 2019-12-14]. https://www.brainz.org/15-real-world-applications-genetic-algorithms/

[29] Buckland, Mat. *AI Techniques For Game Programmers.* Boston: Premier Press, 2002. ISBN 1-931841-08-X

[30] Video Game News, Reviews, and Walkthroughs – IGN.com. [online]. 2001, [cit. 2019-11-23]. https://www.ign.com/articles/2001/09/10/advance-wars/

[31] Advance Wars. In: *Wikipedia, The Free Encyclopedia* [online] [cit. 2019-11-23]. https://en.wikipedia.org/wiki/Advance_Wars

[32] WarGroove – Official Site. [online]. 2018, [cit. 2019-11-23]. `https://wargroove.com/faq/`

[33] Video Game News, Reviews, and Walkthroughs - IGN.com. [online]. 2019, [cit. 2019-11-23]. `https://www.ign.com/articles/2019/01/30/wargroove-review`

[34] Tanks of Freedom. [online]. [cit. 2019-11-23]. `https://tof.p1x.in/`

[35] Tanks of Freedom Manual. [online]. [cit. 2019-11-23]. `https://github.com/w84death/Tanks-of-Freedom/wiki`

[36] Blondie24: the full story In: *Youtube* [online] [cit. 2019-12-14]. `https://www.youtube.com/watch?v=QSNs-PYv7co`. Channel of user JacobsSchoolNews.

[37] Blondie24. *Wikipedia, The Free Encyclopedia* [online]. 2019 [cit. 2019-12-14]. `https://en.wikipedia.org/wiki/Blondie24`

[38] Chellapilla K.; Fogel D. B. *Evolution, Neural Networks, Games, and Intelligence.*. In: *Proceedings of the IEEE.* Berlin: Springer, 1999, p. 1471–1496. `https://doi.org/10.1109/4235.735432`

[39] A Way to Deal With Enormous Branching Factors. In: *Togelius – Better Playing Through Algorithms* [online] [cit. 2019-12-14]. `http://togelius.blogspot.com/2016/03/a-way-to-deal-with-enormous-branching.html`

[40] Agner, Fog. *Optimizing software in C++* [online]. 2019 [cit. 2019-12-14]. `https://www.agner.org/optimize/optimizing_cpp.pdf`

[41] Stroustrup, Bjarne *The C++ programming language.* Upper Saddle River, Addison–Wesley, 2013. ISBN 978-0-321-56384-2.

[42] Simple DirectMedia Layer - Homepage [online]. 2019 [cit. 2019-12-27]. `https://www.libsdl.org/`

[43] JsonCpp. *The world's leading software development platform GitHub* [online]. 2019 [cit. 2019-12-27]. `https://github.com/open-source-parsers/jsoncpp`

[44] Catch2. *The world's leading software development platform GitHub* [online]. 2019 [cit. 2019-12-27]. `https://github.com/catchorg/Catch2`

[45] Game Loop. In: *Game Programming Patterns* [online] [cit. 2019-12-14]. `https://gameprogrammingpatterns.com/game-loop.html`

[46] Command. In: *Game Programming Patterns* [online] [cit. 2019-12-14]. `https://gameprogrammingpatterns.com/command.html`

[47] Sean Parent. Better Code: Runtime Polymorphism – Sean Parent In: *Youtube* [online] [cit. 2019-12-14]. `https://www.youtube.com/watch?v=QGcVXgEVMJg`. Channel of user NDC Conferences.

[48] The tiny neural network library. *The world's leading software development platform GitHub* [online]. 2019 [cit. 2019-12-27]. `https://github.com/glouw/tinn`

[49] Data Locality. In: *Game Programming Patterns* [online] [cit. 2019-12-14]. `https://gameprogrammingpatterns.com/data-locality.html`

[50] MNIST handwritten digit database [online]. [cit. 2019-12-27] `http://yann.lecun.com/exdb/mnist/`

[51] Centroid. In: *Wikipedia, The Free Encyclopedia* [online] [cit. 2019-11-23]. `https://en.wikipedia.org/wiki/Centroid`

[52] Fogel David; Hays Timothy; Hahn Sarah; Quon James *A Self-learning Evolutionary Chess Program.* In: *Proceedings of the IEEE.* 2004, p. 1947–1954. `https://doi.org/10.1109/JPROC.2004.837633`

Figure References

[1] *OXO running in Classis in Mac OS X* [online] [cit. 2019-12-23]. Available at: https://en.wikipedia.org/wiki/OXO#/media/File:OXO_emulated_screenshot.png

[2] Williams, Aaron. *Full Frontal Nerdity* [online] [cit. 2019-12-23]. Available at: http://ffn.nodwick.com/ffnstrips/2011-11-15.png

[3] *Rock Paper Scissors* [online] [cit. 2019-12-23]. Available at: https://en.wikipedia.org/wiki/Rock_paper_scissors#/media/File:Rock-paper-scissors.svg

[4] *Advance Wars* [online] [cit. 2019-12-23]. Available at: https://giantbomb1.cbsistatic.com/uploads/original/0/3699/418220-154908_3_2.jpg

[5] *Wargroove* [online] [cit. 2019-12-23]. Available at: https://wargroovewiki.com/mediawiki/images/5/58/Thumb_The_Breach.png

[6] *Tanks of Freedom* [online] [cit. 2019-12-23]. Available at: https://github.com/w84death/Tanks-of-Freedom

[7] *Blondie24* [online] [cit. 2019-12-23]. Available at: https://youtu.be/TS8QlL-3NXk?t=26

[8] *Hero Academy Crossover* [online] [cit. 2019-12-23]. Available at: http://1.bp.blogspot.com/-K2vOorKLMzk/VvThlEh-OoI/AAAAAAAAC8Q/ZxtBaJjtNpcaOk5GpIKrMaxfrIWWcrkXg/s1600/crossover.png

Appendix A

Acronyms

2D	Two-dimensional
AI	Artificial intelligence
EDSAC	Electronic delay storage automatic calculator
ESA	Entertainment Software Association
FPS	First-person shooter
GUI	Graphical user interface
MNIST	Modified National Institute of Standards and Technology
MSE	Mean squared error
NPC	Non-player character
RTS	Real-time strategy
RPG	Role-playing game
SDL	Simple Direct Media Layer
TINN	Tiny Neural Network
TBS	Turn-based strategy

Appendix **B**

Contents of Enclosed SD Card

```
│ readme.txt ........................ the file with SD contents description
├── evaluated_results ....... the directory with executed tests with graphs
├── src ................................... the directory of source codes
│   ├── impl .................................... implementation sources
│   └── thesis ............. the directory of LaTeX source codes of the thesis
└── text ........................................ the thesis text directory
    └── BP_Stepanek_Ivan_2020.pdf ......... the thesis text in PDF format
```

Appendix C

Game Manual

C.1 Software Requirements

Creating usable executable files require C++17 features supported by compilers such as *g++ 8* or higher, *clang 7* or higher. For easier building *Cmake 3.13* or higher is needed. In addition user must install graphic libraries such as *SDL 2*, *SDL_image 2* for loading textures and *SDL_ttf 2* for loading fonts. Recommended operation system is *Linux* since the program was tested on *Debian* distribution.

C.2 Getting Required Software

For Debian and Ubuntu users, SDL libraries can be easily obtained via a package manager using command-line interface. Commands are showed in Listing C.1. Package managers may differ between Linux distributions and the commands might be invalid in a different operating system. For successful installation `root` priviliges might be needed.

Listing C.1: Terminal Installation

```
apt install libsdl2-dev
apt install libsdl2-image-dev
apt install libsdl2-ttf-def

apt install g++
apt install cmake
```

Listing C.2: Building Source Code

```
cd RGB_Wars
mkdir Release
cd Release
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=/usr/
    bin/g++ ..
make all
```

C.3 Building the Source Code

To create executables it is needed to create a directory using an arbitrary name inside the `RGB_Wars`. In the created directory `cmake` command will be executed. Then it will generate platform specific build tool, like *Make* for *Linux*. After that it is needed to execute the generated build tool. Finally the executables are created from the source code. Listings C.2 shows all neccesary commands.

C.4 Launching the Game

After the successful building there will be `RGBWarsApp` executable in the `RGB_Wars` directory. After launching the game from terminal a screen with simple menu is displayed. The entire game uses only 6 buttons. Arrow keyboards for navigating the cursor, button `a` for accepting or chosing and button `s` for deselecting or going back. These keyboards are used for both navigating in menu and actual playing.

C.5 Game

Objective of the game is to destroy all enemy units or to capture enemy base. Capturing can be done only by infantry units. The smaller the unit health the longer time to capture. Buildings can refuel and heal units, if player places his units at his captured building. Player has to think of unit positions since different terrains provide different levels of defence. Defence status of a terrain is shown when player places cursor at the terrain. If player moves unit at some position, the move can be reverted by pressing button `s`. However the move cannot be taken back when it has been already confirmed. Move for specific unit is confirmed when unit cannot make another moves and is gray.