TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

IT40LT

Vladislav Zakharenkov 164755

SOLVING THE TARTARUS PROBLEM BY MEANS OF GENETIC ALGORITHMS

Bachelor's Thesis

Supervisor: Margarita Spitšakova
M.Sc.

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

IT40LT

Vladislav Zakharenkov 164755

TARTAROSE PROBLEEMI LAHENDAMINE GENEETILISTE ALGORITMIDEGA

Bakalaureusetöö

Juhendaja: Margarita Spitšakova
M.Sc.

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Vladislav Zakharenkov

22.05.2017

Abstract

Evolutionary Robotics is a methodology inspired by the Darwinian principle of selective reproduction of the fittest. In this approach, robots are treated as autonomous artificial organisms that develop necessary skills in order to survive in their test environment. Genetic Algorithms are one the most popular techniques employed to solve robot controller optimization tasks.

The main objective of this thesis is to develop a search method in the form of a Genetic Algorithm that produces reasonably fit controllers for a path-finding and planning problem called Tartarus. In the course of this work, two separate controller models are designed, implemented and evaluated. Two independent baselines are used to validate the experimental results: one trivial, one from existing literature.

The result of this work is a search algorithm and a standalone Java application with a graphical user interface that allows to reproduce the experiments presented in this thesis as well as perform custom search using proposed models.

This thesis is written in English and is 49 pages long, including 10 chapters, 27 figures and 5 tables.

Kokkuvõte

Tartarose probleemi lahendamine geneetiliste algoritmidega

Evolutsiooniline robootika on metodoloogia, mille aluseks on Darwini selektiivse reproduktsiooni printsiip. See lähenemisviis käsitleb roboteid autonoomsete tehisorganismitena, mis arenguprotsessis omandavad testiümbruses ellujäämiseks vajalikke oskusi. Geneetilised algoritmid on üks levinumaid meetodeid, mida kasutatakse roboti kontrolleri optimeerimiseks.

Käesoleva töö peamiseks eesmärgiks on arendada geneetilisel algoritmil põhinevat otsingumeetodit, mis produtseerib Tartaros-nimeliseks raja leidmise ja planeerimise ülesandeks kõlblikke kontrollereid. Töö käigus on kavandatud, implementeeritud ja testitud kaks erinevat kontrolleri mudelit. Eksperimendi tulemuste valideerimiseks on kasutatud kaks sõltumatut lähtetaset, millest üks on triviaalne ja teine põhineb Tartarose teemat käsitleval kirjandusel.

Töö tulemuseks on otsingu algoritm ja autonoomne Java rakendus graafilise kasutajaliidesega, mis võimaldab reprodutseerida käesolevas töös esitletud eksperimente ning sooritada kohandatud otsingut kasutades pakutud mudeleid.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 49 leheküljel, 10 peatükki, 27 joonist, 5 tabelit.

List of abbreviations and terms

CSV	Comma-Separated Values, file format
EA	Evolutionary Algorithm
FSM	Finite State Machine
GA	Genetic Algorithm
GP	Genetic Programming
GUI	Graphical User Interface

Table of Contents

1 Introduction....12
1.1 Problem overview....12
1.2 Objectives....13
1.3 Outline....13
2 Problem statement....14
3 Theoretical background....17
3.1 Evolutionary Algorithms....17
3.2 Genetic Algorithms....17
3.2.1 Canonical Genetic Algorithm....18
3.2.2 Genetic representation....18
3.3 Genetic operators....19
3.3.1 Selection....19
3.3.2 Crossover....19
3.3.3 Mutation....20
3.4 Finite state machines....20
4 State of the art....21
5 Methodology....22
5.1 Tools....22
5.2 Strategy....23
5.3 Baselines for validation....23
6 Controller models....24
6.1 Baseline string controller....24
6.2 Proposed controller model....25
6.3 Genetic representation....26
6.3.1 Baseline string controller....26
6.3.2 Proposed controller model....26
6.3.3 Possible genetic defects....26
6.3.4 Prevention of genetic defects....27

6.4 Sensory data interpretation....27
6.4.1 Position inference....28
6.4.2 Sensor simplification....29
6.4.3 Left-Right Agnostic....30
6.5 Behaviour around walls and corners....31
6.6 Model A....32
6.7 Model B....33
7 Proposed search method....34
7.1 Evolutionary process....34
7.2 Fitness function....35
7.3 Statistics collection....35
7.4 Search parameters....36
8 Experiments....37
8.1 Random noise generator....37
8.1.1 Pure noise....37
8.1.2 Hard-coded edge and corner behaviour....38
8.2 Baseline string controller....39
8.3 Model A....40
8.4 Model B....41
9 Analysis....43
9.1 Model comparison....43
9.2 Discussion....45
9.2.1 Success....45
9.2.2 Shortcomings....45
9.3 Future work....46
10 Summary....47
References....48

List of Figures

Figure 1: Examples of valid Tartarus starting configurations..14

Figure 2: Sensor readings for configurations from Figure 1..14

Figure 3: Examples of Tartarus configurations with scores...15

Figure 4: An example of an invalid starting configuration...16

Figure 5: Activity diagram of the canonical GA..18

Figure 6: An example of a finite state machine...20

Figure 7: Data flow through the controller...24

Figure 8: Baseline string controller as a single-loop finite state machine......................24

Figure 9: Proposed controller model as a multi-loop finite state machine.....................25

Figure 10: An example of a genome consisting of a single chromosome.......................26

Figure 11: An example of a genome consisting of multiple chromosomes....................26

Figure 12: Examples of impossible sensor system states...28

Figure 13: Sensor usage percentages by successful FSM controllers as reported by Ashlock and Freeman...29

Figure 14: Combination of observable sensor states as a result of sensor system simplification...30

Figure 15: Examples of sensor image symmetry associations: none (top) and pair (bottom)...30

Figure 16: Edge states with prohibited transitions in red...31

Figure 17: Corner states with prohibited transitions in red..32

Figure 18: Activity diagram of a single evolution run..34

Figure 19: An example of two Tartarus configurations with the same set of solutions...35

Figure 20: Results for pure pseudo-random noise generator..38

Figure 21: Results for pseudo-random noise with overridden perimeter behaviour.......38

Figure 22: Results for baseline string controller of size 80..39

Figure 23: Results for controller model A..40

Figure 24: Distribution of raw sensor images observable on the inner grid between the 12 subroutines in model B...41

Figure 25: Results for controller model B..42
Figure 26: Comparison of controller model results..43
Figure 27: Advantages of proposed controller models over the baseline........................44

List of Tables

Table 1: Search parameters..36

Table 2: Global search parameters for conducted experiments.......................................37

Table 3: Evolution parameters for the baseline string controller.....................................39

Table 4: Evolution parameters for controller model A..40

Table 5: Evolution parameters for controller model B..41

1 Introduction

Ever since the principles of cybernetics were formulated by Norbert Wiener in 1948, the interest in robotics has been increasing around the world as various digital technologies became more accessible. Nowadays, robots are developed for commercial, domestic and military purposes and are present in virtually every aspect of human endeavour. One of the methods of creating autonomous robots is Evolutionary Robotics. It is based on the Darwinian principle and employs Evolutionary Algorithms to achieve its goals [1].

Biological evolution is an appealing source of inspiration for addressing problems of creating adaptive and innovative solutions that may be too complex to program by hand [2]. Genetic Algorithms are a subset of Evolutionary Algorithms and are used for both theoretical research and finding solutions with real life applications [3].

Robot controller optimization tasks are usually non-trivial problems that have no known algorithmic solutions and require extensive use of metaheuristics such as Genetic Algorithms. This thesis deals with a robot controller optimization task for a particular problem called Tartarus [4].

1.1 Problem overview

Tartarus is a path-finding and planning problem where a virtual robot agent is tasked with pushing boxes from the centre of a fixed-size grid world towards its perimeter. The robot is equipped with eight sensors to identify adjacent boxes and world boundaries once it comes in contact with them and has no knowledge of its own position in the world. It can only move forward one square at a time or make 90-degree turns while remaining stationary. The robot agent is expected to succeed at solving this partially observable problem in limited simulation time.

The Tartarus problem is categorized as a deterministic NP-hard problem with no known optimal solution [5]. In the original article, Teller focuses not so much on achieving high fitness scores, but on the ability of the robot agents to form mental models of the world using indexed memory [4]. However, because of the many challenging properties of its environment, a large number of equally scoring solutions and a very large search space, the Tartarus problem is now regarded by many as a standard test problem to evaluate techniques in artificial intelligence [5]–[11].

1.2 Objectives

The main goal of this thesis is to develop a Genetic Algorithm-based search method that produces reasonably fit controllers for the Tartarus problem with results that can be related to existing literature. This entails viable controller model design and implementation along with genetic representation as well as provision of baselines for subsequent analysis. The search algorithm is implemented as a standalone Java application that allows to configure and run reproducible experiments using said models and allow for the outcome to be analysed using external tools.

1.3 Outline

The thesis is organized in the following way. Chapter 2 contains the full problem statement. Chapters 3 introduces the basic theoretical concepts behind Genetic Algorithms and finite state machines. Chapter 4 provides a brief overview of existing solutions and applied techniques. Chapter 5 elaborates on the choice of tools, general strategy and validation methods used in the thesis. Chapters 6 and 7 are dedicated to the detailed explanation of the proposed solution – controller models and search method, respectively. Chapter 8 documents the conducted experiments and presents obtained results, while Chapter 9 focuses on outcome analysis and discussion of possible interpretations before leading to conclusions.

2 Problem statement

The classical Tartarus environment [4] is a 6×6 grid with 6 boxes randomly distributed on the inner 4×4 grid. The robot can be located on any square of the inner grid that is not occupied by a box and may face one of four directions: North, East, South and West.

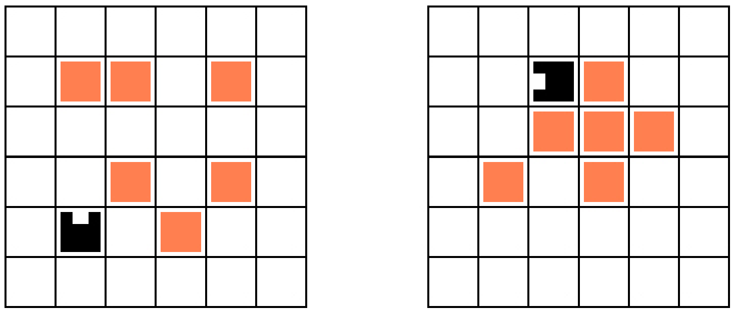
Figure 1: Examples of valid Tartarus starting configurations.

The robot has no knowledge of its position in the world or the direction it is facing. To gather information about its surroundings, it relies solely on a system of eight sensors that are attached directly to its hull. The sensors are aimed at squares adjacent to the robot and can distinguish between boxes, empty squares, and "walls" – positions that lie outside the bounds of the grid.

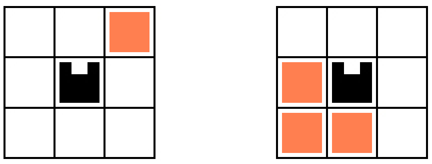
Figure 2: Sensor readings for configurations from Figure 1.

The robot has three actions at its disposal: turn left, turn right (the robot remains on the same square while turning), and move forward. Each action costs 1 time unit to perform. The goal of the robot is to push all of the boxes out of the inner grid and onto the squares that constitute the perimeter within the span of 80 time units. Time is discrete and actions are performed instantaneously, without any transitional states.

There are no restrictions on the robot's ability to turn left or right, so a decision to perform a turn always produces the desired effect. However, the robot's ability to move forward is limited to two scenarios only:

- when there is an empty square right in front of the robot, in which case the robot moves forward onto the empty square,
- where there is a single box right in front of the robot and an empty square directly behind that box, in which case the robot pushes the box onto the empty square and moves onto the square that was previously occupied by the box.

Any attempts to move forward that do not correspond to either of these scenarios – like trying to move into walls or push boxes that have obstacles behind them – produce no effect, but cost 1 time unit nonetheless.

The robot's success is measured by the number of boxes that it has managed to push onto the perimeter by the end of the allotted time: 1 point for each box on the straight edge of the grid, 2 points for each box in the corner. Thus, the maximum score is 10.

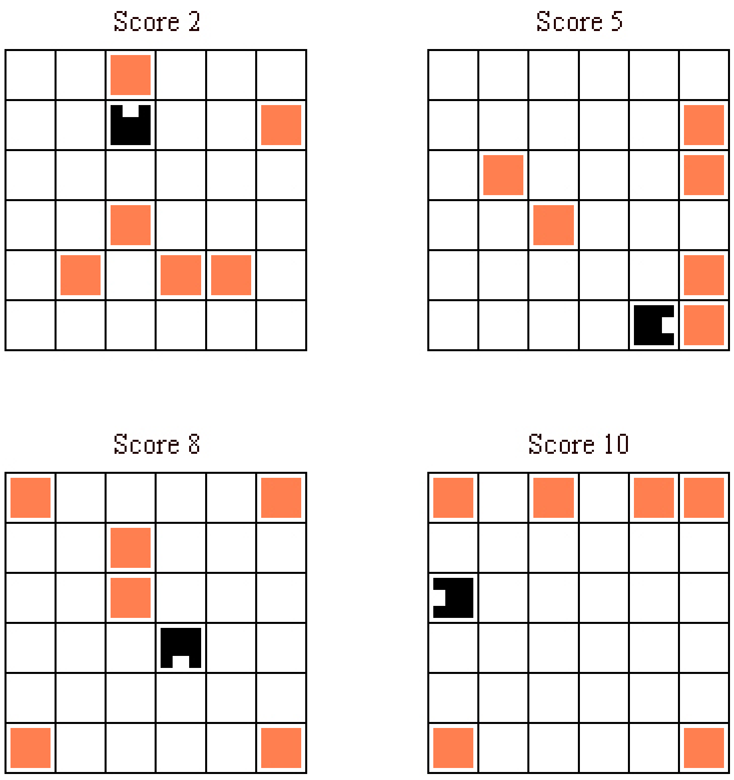

Figure 3: Examples of Tartarus configurations with scores.

Unsolvable starting configurations exist: if four or more adjacent boxes form a rectangular block, it becomes impossible for the robot to push any of these boxes and the maximum achievable score drops to 2 or lower.

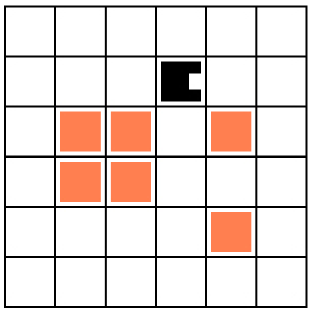
Figure 4: An example of an invalid starting configuration.

Approx. 7.3% of all possible starting configurations exhibit such a trait [9]. This is significant enough to affect the evaluation process, therefore such configurations are considered invalid and are excluded from the trials by consensus.

3 Theoretical background

For optimization problems like Tartarus where multiple solutions exist, any solution that is good enough proves to be sufficient. Since there is no known principled way to find a good solution, but all candidate solutions can be easily verified and graded, metaheuristics are employed. Metaheuristics are techniques that help find solutions when there is little heuristic information to go on [12].

3.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) are an example of metaheuristics. EAs operate on populations of solutions using the theory of evolution as an algorithm [13]. Survival of the fittest serves as the driving force towards better solutions to a problem.

First, a population of data structures that represent candidate solutions is generated. A fitness function is then used to determine how good the individual solutions are. The algorithm exhibits a bias towards better solutions and selects them for further variation and tweaking, while less suitable solutions are killed off and replaced by copies of better ones. Together, survivors and offspring solutions form the next generation and the whole process is repeated until the end condition has been satisfied [13].

3.2 Genetic Algorithms

Genetic Algorithms (GA) are the best-known subset of EAs [13]. GAs are search algorithms based on the mechanics of natural selection and natural genetics. Solutions are encoded as genetic material which then undergoes change over generations, hence the name of the technique. While relying on random processes, GAs efficiently converge on solutions with better performance [14].

Many different variations of GAs exist. Given all of their differences they do not, however, deviate significantly from the canonical form.

3.2.1 Canonical Genetic Algorithm

The canonical GA follows the iterative process of EA with specialized steps: after creating the initial population, fitness of individuals is assessed, parent solutions are selected and offspring is created by manipulating the genetic material [12].

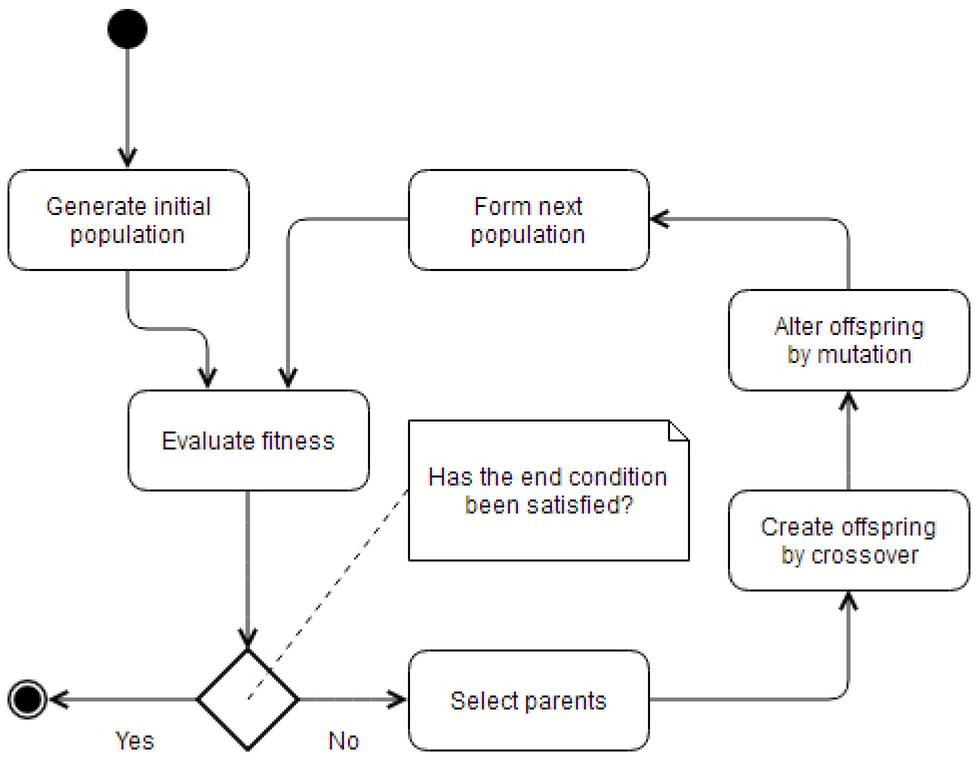
Generate initial population
Form next population
Alter offspring by mutation
Evaluate fitness
Has the end condition been satisfied?
Create offspring by crossover
Select parents
Yes
No

Figure 5: Activity diagram of the canonical GA.

The genetic manipulations in question are crossover and mutation, which will be explained later in the chapter along with methods used for parent selection.

3.2.2 Genetic representation

The classical representation of genetic material is in the form of bit strings. Each bit is a gene, its values 0 and 1 are alleles, each string of bits is a chromosome. Together, the chromosomes make up the genome of an individual solution, i.e. all of its genetic material. The term genotype is used to describe a particular set of genes contained in the genome [2].

Depending on the problem domain and solution encoding scheme, genes can be data types other than bits and allow for more than two possible alleles. They are still represented internally as short blocks of binary data [2].

3.3 Genetic operators

When considering the process of evolution, it is very important to distinguish between pure chance and cumulative selection, as these are two completely different things that yield vastly different results: while the former is merely blink luck with all the accompanying inefficiency, the latter is directed by non-random survival, which makes it a fundamentally non-random process [15].

3.3.1 Selection

The simplest selection method is truncation selection, where only the fittest individuals of the population are eligible for reproduction. This ensures that only the best possible genetic material is used for the next generation, but may cause convergence too early in the evolution [16].

Fitness-proportionate selection, also called roulette-wheel selection, is a method where individuals in the population have an increasingly higher chance of being chosen for reproduction based on their fitness. The better the fitness, the higher the chance. This helps maintain diversity of the population [16].

Tournament selection is a method that takes a group of *n* individuals and chooses the best individual out of that group for reproduction. This procedure may be repeated among the best individuals again depending the size of the population, necessary parent count, and group size *n*.

In many cases, the quality of the solution found by GAs depends on the choice of the selection method [16].

3.3.2 Crossover

Crossover is a process of exchanging subparts of parent chromosomes to produce new offspring, roughly mimicking biological recombination between two organisms [2]. The choice of the crossover method helps avoid premature convergence in GAs and affects the overall performance of the search [17].

Crossover usually involves slicing of chromosomes in one or more places. A result of single-point crossover between two strings 01000110 and 10011111 straight down the middle would be two offspring strings 01001111 and 10010110. Had the single-point crossover been applied at the first gene, the offspring string would be 00011111 and 11000110.

Multi-point crossover slices chromosomes into more parts in multiple locations, while the uniform crossover, as the name suggests, distributes genes from both parents uniformly between the offspring chromosomes.

3.3.3 Mutation

Mutation is a straightforward operation of randomly selecting some gene in a chromosome of an individual and altering it. This introduces random change into the process of evolution and mimics the DNA copying errors that occur in nature. The mutation procedure itself depends on the gene data type, and can vary from a simple binary bit flip to specialized algorithms [12].

3.4 Finite state machines

A finite state machine (FSM) is one of the most powerful and best-known models used to describe behaviour. It uses the concept of state to represent a kind of internal memory, allowing to create sophisticated control models, where events and inputs trigger transitions between states [18].

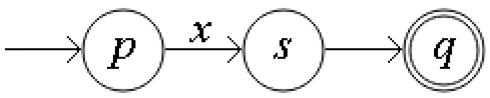

Figure 6: An example of a finite state machine.

Figure 6 shows a simple FSM consisting of 3 states: the initial state *p*, the intermediate state *s*, and the final state *q*. In order to get to state *q*, the machine has to invariably transition from its starting state *p* through state *s* and further – this type of memory is characteristic of FSMs. The machine in the provided example cannot end its operation in states *p* or *s*, as they are not final. Transitions can be triggered by specific events, like *x* for the transition between *p* and *s*, or happen automatically without relying on input – the latter are called epsilon transitions and do not have any associated input markings on the diagrams.

4 State of the art

Since Teller's original article, many different approaches to the problem have been developed and documented. This section aims to provide a very brief overview of existing solutions.

Teller's initial approach uses genetic programming (GP) with indexed memory and introduces the distinction between and reactive and non-reactive controllers [4]. Consequent work by Balakrishnan and Honavar uses neurocontrollers and experiments with the evolution of sensors, slightly deviating from the classical problem specification regarding sensor placement [19].

A considerable amount of work on the Tartarus problem was done by Ashlock. In a number of separate articles, Ashlock and others introduce baselines for future experiments: non-reactive string and reactive FSM controllers [6], [7]. Studies also report good fitness scores for If-Statement-Action table controllers [6].

Controllers that rely on specialized memory and additional heuristics are reported to achieve very high scores [8]. However, the GA and experimental conditions in the related study also play a significant role in the reported success.

Other approaches implement such models as tree state machines [20], fractal gene regulatory networks [10], and recurrent neural networks [21]. At the time of writing, the most recent work published regarding Tartarus is a revision of the GP-based approach [11].

It is also important to mention the research of coevolution in Tartarus (robots being the hosts and boards being parasites) [9] as well as research of diversity sustaining mechanisms to avoid getting stuck in local minima [21] and improve on the standard fitness-based search [22]. These works definitely contribute to the overall understanding of the Tartarus problem.

5 Methodology

Various methods and technologies can be utilized to tackle the Tartarus problem. The ones used in this thesis are chosen with accessibility and efficiency in mind.

5.1 Tools

GAs are processing-intensive techniques, so even though interpreted languages like Python boast a good number of libraries and frameworks for GAs, compiled languages are much more suitable due to their efficiency and speed.

Java was chosen as the programming language for the implementation of models and algorithms described in this thesis due to the author's previous experience and familiarity with it. Programming in Java has the added benefit of the resulting application being platform-independent. Another argument in favour of Java is JavaFX, which allows for fast prototyping and implementation of graphical user interface (GUI) applications [23].

The four most prominent libraries for GAs in Java are ECJ (Evolutionary Computation in Java) [24], JGAP (Java Genetic Algorithms Package) [25], Watchmaker [26] and Jenetics [27]. While the first two are well-established, tested and documented, they target earlier versions of Java and no development is underway to modernize them. Watchmaker is an extensive high-performance evolutionary computation framework with sufficient documentation, but an overabundance of features that are not required for the purposes of this thesis. Jenetics, on the other hand, is an up-to-date actively developed and tested library that focuses specifically on GAs. Its stability, focus on modern Java, active support and solid documentation make it the library of choice for achieving the goals set forth in this thesis.

5.2 Strategy

While it is possible for non-reactive controllers to achieve high fitness scores [4], this is more of an exception and reactive controllers tend to perform significantly better than non-reactive ones [6], [7]. Therefore the controller model proposed in this thesis is reactive.

Existing research indicates that even though certain controller models are able evolve to interpret sensory data in an efficient way, humans are much better at extraction of useful information and sensor interpretation is a task more fit for a human and should not be delegated to GA [8].

Memory is very important in the Tartarus problem – unless the virtual robot agent employs some sort of internal state, it is doomed to fail in scenarios where its actions have no observable effect on the environment. For example, a robot without memory will not be able to recognize a failed attempt at pushing a box forward or perform open-field maneuvers without any boxes in sight [4].

An informed choice is made to avoid GP techniques because of their tendency to impose additional survival problems onto the evolutionary process [28].

5.3 Baselines for validation

Simple and easy-to-analyse controllers are used as baselines. They provide minimal expected performance levels. Two baseline controllers are used to validate experimental results.

The first baseline represents the lowest possible performance, which is expected from a controller driven by a random noise generator that is not affected by the evolutionary process. Anything above the noise level can be considered an improvement, while all performance below the random noise baseline indicates a defective controller.

The second, more advanced, baseline is a non-reactive string controller that was chosen in order to relate the results presented in this thesis to existing studies [6]. Since this thesis focuses on reactive controllers, they should perform noticeably better than non-reactive baselines.

6 Controller models

A controller for a virtual robot agent in Tartarus can be viewed as a monolithic component that takes sensory data as input and outputs a single decision in the form of an action for the robot to perform.

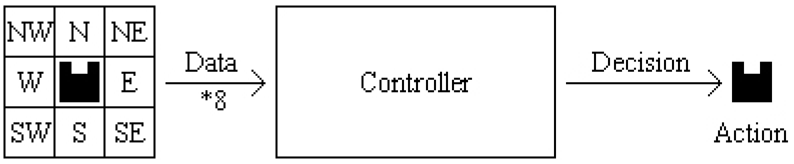

Figure 7: Data flow through the controller.

It is imperative to explicitly define enumerations and encodings in order to avoid any confusion and ambiguity when referring to data sources as well as input and output values.

Individual sensors that supply data to the controller are enumerated in the following way: N for North, NE for North-East, E for East, SE for South-East, S for South, SW for South-West, W for West, and NW for North-West. Each sensor has three states, encoded as EMPTY, BOX, and WALL. Action codes are: L for "turn left", R for "turn right", and F for "move forward." These abbreviations will be used throughout the rest of the document.

6.1 Baseline string controller

A non-reactive fixed-size string controller is a single routine that executes in a loop until the simulation ends. It can be represented as a simple FSM where all states are final.

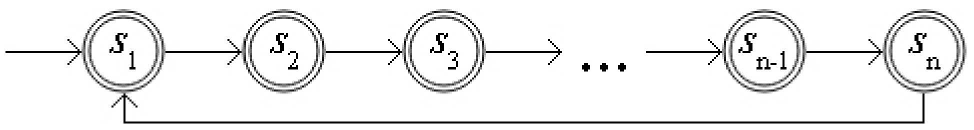

Figure 8: Baseline string controller as a single-loop finite state machine.

There are *n* states, each state produces a single decision. The number and order of states are fixed throughout evolution, the only variables in the model are the action codes produced by each state.

This model represents a controller that develops a general strategy to tackle all possible scenarios.

6.2 Proposed controller model

The decision-making mechanism of the proposed controller model is a finite state machine with multiple subroutines of arbitrary lengths. A special initial state is used to process sensory data and react by executing the corresponding routine. Once the triggered routine has finished executing, the machine immediately returns to its initial state to consult the sensors once again.

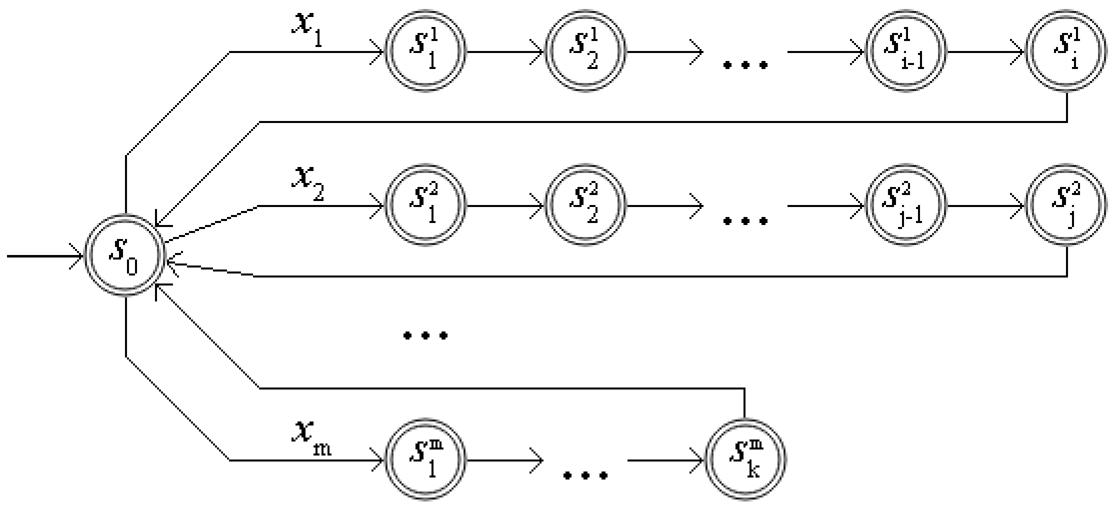

Figure 9: Proposed controller model as a multi-loop finite state machine.

The total number of states depends on the number of subroutines and their respective lengths. Total state count also includes the special state. As with the baseline model, the number and order of states are fixed throughout evolution and the only variables are the action codes produced by states belonging to subroutines.

This model represents a controller that has the ability to develop a more complex strategy with multiple specialized manoeuvres to handle different scenarios. The baseline string controller is, in fact, a slightly altered special case of the proposed model.

6.3 Genetic representation

Genetic representation used for the aforementioned models is very straightforward. Every gene can take on one of three values: L, R, or F. These directly represent the action codes for the states that make up the routine(s). Keeping the genetic material as simple as possible allows for easier and more accurate analysis of the results.

6.3.1 Baseline string controller

Genetic representation for the baseline controller is straightforward: the genome consists of a single chromosome of length *n* (the total number of states). Every gene in the chromosome is an action code that corresponds to the *n*th state.

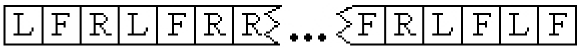

Figure 10: An example of a genome consisting of a single chromosome.

The genome size is equal to the total number of states that produce robot actions.

6.3.2 Proposed controller model

Genetic representation for the proposed controller model differs from the baseline only by the number of chromosomes in the genome. Every subroutine is encoded as a separate chromosome of appropriate length.

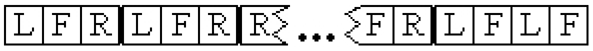

Figure 11: An example of a genome consisting of multiple chromosomes.

The special sensor analysis state from the proposed controller model is not subject to evolution and is therefore not represented in the genome.

6.3.3 Possible genetic defects

There is a number of allele combinations that result in the controller making poor decisions so that actions taken are guaranteed to have no effect [8]. Two examples of the most common defective allele combinations are LR and RL, which have a chance of appearing in any two adjacent genes 2 out of 9 times (provided that genetic material is generated randomly).

Depending on genome size, this results in roughly 11-22% of randomly generated initial genetic material guaranteed to be junk genes. Subsequent mutations may activate these dormant pairs, so for larger chromosomes this is not an issue – they contain enough effective genes to accommodate for the loss. However, for controllers with smaller chromosomes (i.e. shorter routines), such genetic defects may become lethal, rendering entire routines useless. Defects in shorter routines will also have an amplified effect due to being executed multiple times per trial.

Other wasteful defects exist, but they have a lower probability of occurring. These include various series of turns, any consecutive forward moves after five in a row, and other even more unlikely sequences.

6.3.4 Prevention of genetic defects

Proposed controller models will suffer from genetic defects quite significantly and need to be protected. Their initial populations are created free of LR and RL combinations (which has a positive side effect of breaking up some of the longer turn-only sequences). Crossover and mutation are not restricted in any way and will still introduce genetic defects over the course of evolution. This is allowed as a trade-off to avoid creating tight genetic interdependencies that may render specific genes immutable in certain combinations: for example, prohibiting LR and RL combinations to be formed during genetic operations will result in combinations like LFR, where the middle gene cannot be altered unless its neighbours are modified first.

Although possible [8], prevention of genetic defects for string controllers is not implemented in the baseline model in order to conform with examples provided in literature [6].

6.4 Sensory data interpretation

Any and all data that is available to the robot is provided to it by the sensor system. The system comprises of eight tri-state sensors and has $3^8 = 6561$ possible states. However, not all of these states are observable due to the specifics of the Tartarus environment.

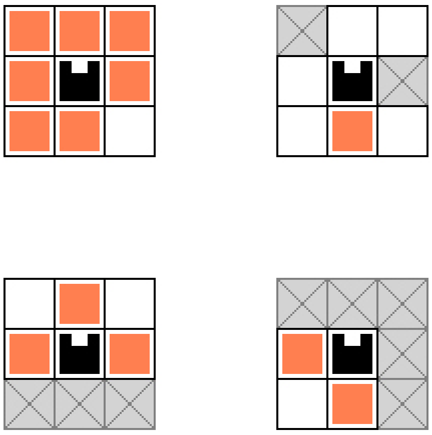

Figure 12: Examples of impossible sensor system states.

While located on the inner grid, the robot can observe not more than 6 boxes and can initially be blocked on all sides. If located on the straight edge of the grid, the robot can observe up to 4 adjacent boxes, but they cannot be blocking it on all sides. When located in a corner, the robot can see up to 2 boxes in a similar non-blocking fashion. The total number of possible observable states can therefore be obtained by calculating:

$$n_{observable}=\sum_{i=0}^{6}\binom{8}{i}+4\left[\sum_{i=0}^{2}\binom{5}{i}+\sum_{i=1}^{2}\binom{3}{i}\binom{2}{3-i}+\binom{3}{2}\right]+4\left[\sum_{i=0}^{1}\binom{3}{i}+\binom{2}{1}\right]$$

Thus, there are only 383 possible observable sensor system states for classical Tartarus. Using the proposed decision-making mechanism, the search space for even the simplest controller with one action per subroutine and one subroutine per sensor system state would shoot up to $3^{383} \approx 5.46 \cdot 10^{182}$. To better appreciate the scale of this number, the volume of the observable universe is approx. $8 \cdot 10^{184}$ Planck volumes [29], which are theoretically considered to be the smallest unit required to quantize space.

Therefore a literal interpretation of sensor data is impossible and it is necessary to drastically reduce the number of scenarios that the controller can distinguish between without leaving any of the possible 383 raw states unaccounted for.

6.4.1 Position inference

An important bit of information present in the raw sensory data is the general position of the robot. Given the nature of the Tartarus board, it can be guaranteed that if a robot's

North sensor detects a wall, then it can be inferred that the robot is located somewhere on the perimeter of the board and is oriented towards a wall. This conclusion may appear trivial, but it provides the robot with very helpful information that was not previously available. The same basic logic can be applied to detect corners and their relative orientation, resulting in 9 possible locations.

All of this new information can be presented through a new virtual position sensor, which can have the following states: NO EDGES, EDGE FRONT, EDGE RIGHT, EDGE BACK, EDGE LEFT, CORNER FRONT LEFT, CORNER BACK LEFT, CORNER BACK RIGHT, CORNER FRONT RIGHT. This method is inspired by heuristics used by Bot, Urquhart and Chisholm [8].

6.4.2 Sensor simplification

A study of sensor preferences in controllers that have evolved sensor interpretation mechanisms shows that the diagonal sensors are used the least by the best-performing controllers [7], so diagonal sensors are discarded to simplify the system.

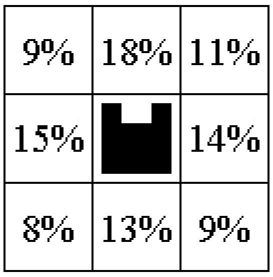

Figure 13: Sensor usage percentages by successful FSM controllers as reported by Ashlock and Freeman.

Once the virtual position sensor has extracted the relevant information, any further need for wall detection disappears. At this stage, the remaining raw sensors can be replaced with binary virtual sensors that detect only two states: BOX or NO BOX, the latter covering the raw states EMPTY and WALL.

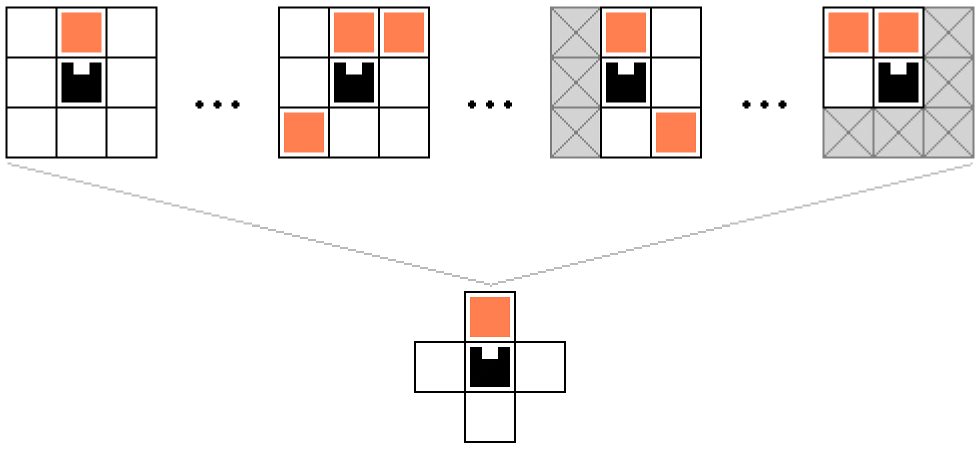
Figure 14: Combination of observable sensor states as a result of sensor system simplification.

The number of possible states becomes significantly lower after simplification.

6.4.3 Left-Right Agnostic

This thesis proposes a way to further reduce the number of sensor system states that the controller can differentiate between and still be able to make effective decisions. This can be achieved by combining pairs of sensor images that are vertically symmetrical to each other. Sensor states that are not a part of any symmetry association are considered original and remain unaffected.

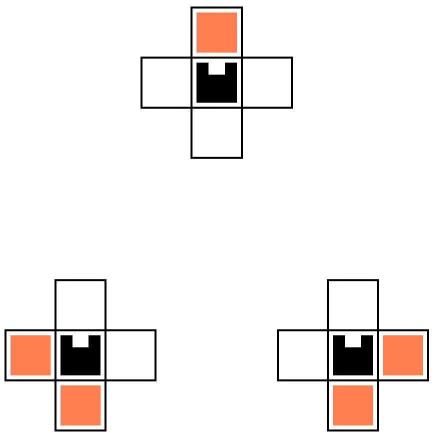
Figure 15: Examples of sensor image symmetry associations: none (top) and pair (bottom).

One of the sensor system states in a pair is designated as the original and second one as the “mirrored” state. A virtual sensor system is responsible for distinguishing between original and mirror states. Its output can serve as an inverter for the controller's resulting decisions: if the original action is a left turn, the mirrored action is a right turn and vice versa. Decisions to move forward remain unaffected.

6.5 Behaviour around walls and corners

There are 16 relative positions on the grid that invariably present increased potential for wasted actions: 4 corners and 12 squares along the perimeter (left, right and upper walls as observed by the robot). 6 of these restrict movement forward, while the other 10 allow the robot to turn towards a wall (and therefore end up in one of the 6 positions with restricted movement options).

While moving forward along the edges is the only available technique for pushing boxes into corners, operating on these squares, although potentially wasteful, is rewarded by evolution. Staying in corners, however, offers zero benefit. The risks are increased since navigating out of corners requires very specific action sequences that are less likely to be found and preserved by GA.

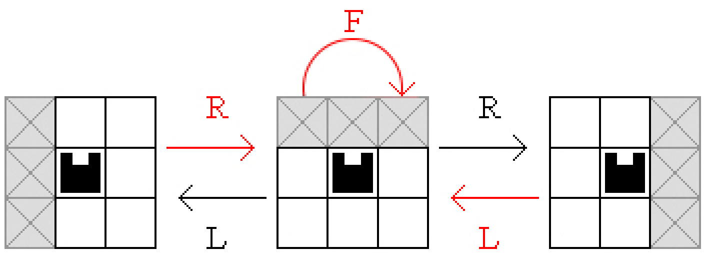

Figure 16: Edge states with prohibited transitions in red.

Moving into a wall has no positive effect and should therefore be avoided. Such behaviour can be circumvented by prohibiting transitions that lead to the robot agent being turned towards the nearby wall. This is not disruptive to behaviour rewarded by evolution.

By prohibiting transitions that result in the robot burrowing further into corners and adding a simple two-step logic it is possible to exit any corner within 2-4 time units, depending on the presence of a blocking box nearby.

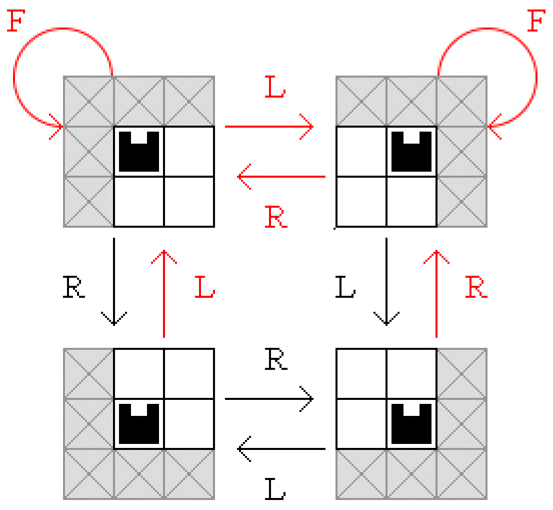

Figure 17: Corner states with prohibited transitions in red.

Transition prohibition guarantees that the robot will assume a position that potentially allows forward movement (ignoring the possibility of an obstructing box) immediately after hitting a closed corner. Once facing away from the corner, the first logical step is to move forward: if after this step the robot finds itself in the same position as it was before, it will turn towards the other open side of the corner and move forward again, going back the way it came from in the first place.

A hard-coded solution that uses the virtual position sensor described earlier can be easily implemented for efficient corner handling and edge behaviour correction instead of relying on the GA to discover and adopt similar behaviours.

6.6 Model A

Controller model A is based on the proposed decision-making mechanism and relies solely on the virtual position sensor for information about its surroundings. It has 5 subroutines that are subject to evolution and 4 hard-coded routines to implement the corner escape logic described in the previous section. The 5 variable subroutines correspond to position sensor states NO EDGES, EDGE FRONT, EDGE RIGHT, EDGE BACK, EDGE LEFT.

Since this model only uses the virtual position sensor, it lacks the ability to detect boxes and bases all of its decisions on its inferred position, remaining unaware of any dynamic obstacles.

6.7 Model B

Controller model B is based on the proposed decision-making mechanism and uses the virtual position sensor alongside the simplified left-right agnostic system of 4 binary sensors. It has 12 subroutines that are subject to evolution – each subroutine corresponds to one of the original images as interpreted by the virtual sensor system.

Simplified sensors in model B are oblivious to walls, so it bases all of its decisions on the perceived configuration of boxes in its proximity. To avoid getting stuck on the perimeter, this model implements hard-coded corner escape logic and corrective behaviour overrides near walls. These hard-coded mechanisms depend on the virtual position sensor.

7 Proposed search method

The proposed search method is implemented as an application with a minimalistic GUI. It is designed with considerations for efficiency, flexibility and usability.

7.1 Evolutionary process

The main search process is a single evolution run and is based on the canonical GA. A single search procedure may consist of multiple consecutive evolution runs.

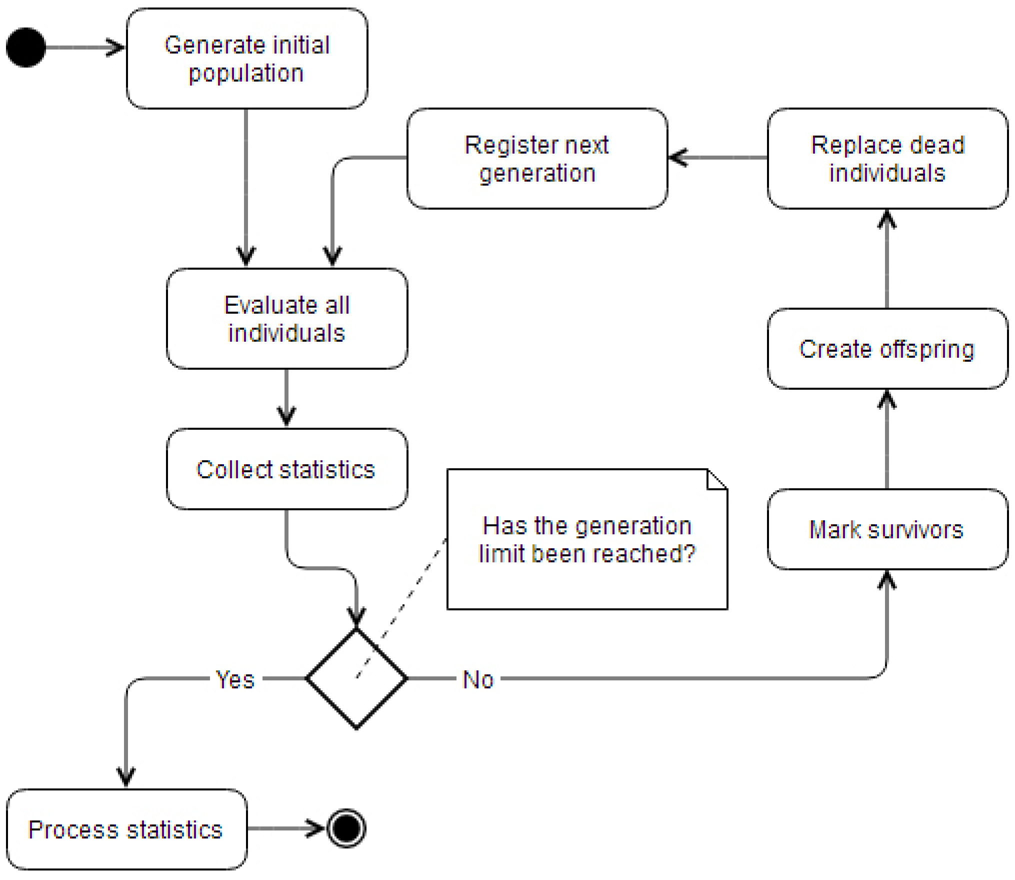

Figure 18: Activity diagram of a single evolution run.

Offspring creation step implies a sequence of selection, crossover and mutation operations as per canonical GA. Statistics collection and processing steps are included for clarity. Fitness evaluation phase is explained in the next section.

7.2 Fitness function

During the fitness evaluation phase, every controller is tested on a set number of random configurations. A new batch of configurations is generated at the beginning of every evaluation phase and all controllers within the population are evaluated using the same collection of trials to provide stable testing conditions.

Each controller is given one attempt per test configuration. Each trial ends with an integer score between 0 and 10. The average result across all trials is then calculated as a real number and assigned to the controller as its fitness score.

Since the absolute orientation of the robot is required strictly for implementation and plays no role in the solution of the configuration, there are rotated configurations that have the same set of solutions and appear absolutely identical from within the simulation.

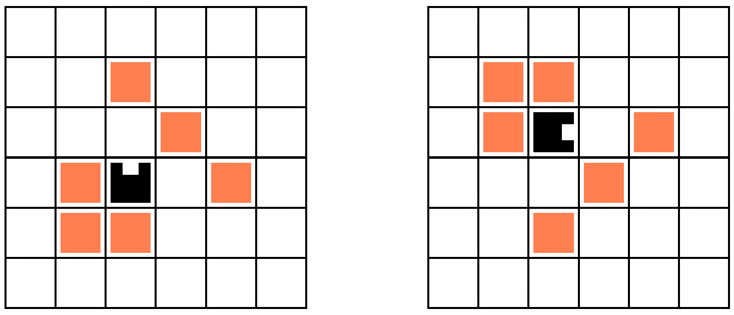
Figure 19: An example of two Tartarus configurations with the same set of solutions.

An experimental option is provided to exclude such duplicate configurations from the trials to ensure a more accurate evaluation process.

7.3 Statistics collection

For every generation, fitness values are collected immediately after the evaluation process. At the end of the evolution run, best and average fitnesses per generation are calculated and stored. If the experiment consists of more than one run, best and average fitnesses per generation are themselves averaged across runs, and the absolute best per generation is calculated using data from all evolution runs.

The application provides an immediate visualization of obtained statistics in the form of a line chart. Collected data can also be saved in CSV file format for a more detailed analysis with external tools.

7.4 Search parameters

Given the non-deterministic trial-and-error nature of GAs, solutions are found by continuous tuning of evolution parameters that requires a great deal of experimentation. This makes capacity for swift reconfiguration a very desirable quality in any GA-based application. An attempt has been made to satisfy this requirement and expose all the necessary parameters to the end user.

Table 1: Search parameters.

Parameter	**Description**	**Allowed values**
Evolution runs	The number of independent evolution runs that constitute a single experiment.	Integer [1..500]
Generation limit	The number of generations after which the evolutionary process ends.	Integer [10..1000]
Population size	The fixed population size sustained throughout the evolutionary process.	Integer [100..5000]
Trials per evaluation	How many random configurations any given individual is tested on during the evaluation phase.	Integer [1..1000]
Exclude duplicate configurations from trials	Self-explanatory. This option is provided for experimental purposes.	Boolean
Offspring fraction	The fraction of individuals to be killed off and replaced with new offspring after each generation.	Real [0..1]
Selection mode	A choice between truncation, roulette wheel and tournament selection modes.	Enumerated options
Tournament size	Applies only to tournament selection mode. Otherwise disabled.	Integer [2..100]
Crossover mode	A choice between single-point, double-point and uniform crossover modes.	Enumerated options
Crossover probability	Recombination probability for the chosen crossover mode.	Real [0..1]
Mutation rate	The fraction of all genes in the offspring that will be affected by random mutations.	Real [0..1]

8 Experiments

The experiments are conducted with fixed global parameters that adhere to the conditions stated in the original article and respected by a good number of existing publications regarding Tartarus [4], [6], [7], [9], [19], [20].

Table 2: Global search parameters for conducted experiments.

Parameter	**Value**
Evolution runs	25
Generation limit	80
Population size	800
Trials per evaluation	40
Exclude duplicate configurations from trials	No
Offspring fraction	0.50

The data for the experiments presented in this thesis was obtained by running the search once and only once per model in order to avoid "cherry-picking" of suitable data sets.

8.1 Random noise generator

Random noise generator is tested in two flavours: the first one produces and executes pure noise made up of L, R and F action codes using a feed from a standard pseudo-random number generator, while the second one passes the same noise through corner escape logic and applies corrective edge behaviour before executing the actions. This model lacks any genetic material and is therefore unaffected by evolution.

8.1.1 Pure noise

Pure random noise generator is expected to produce low fitness scores with a uniform distribution within a certain band.

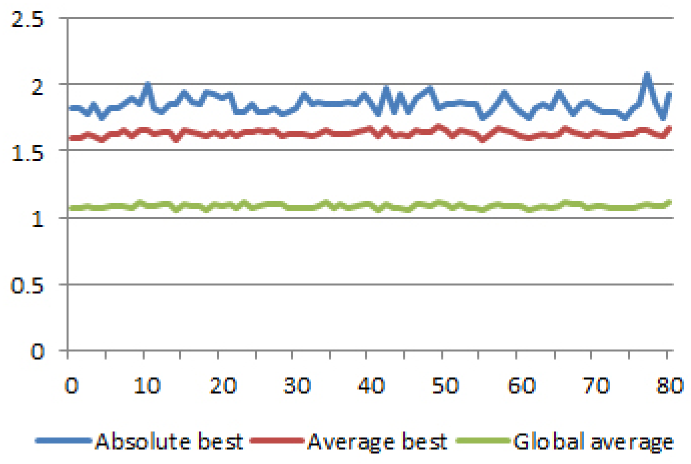

Figure 20: Results for pure pseudo-random noise generator.

The average global fitness baseline is established at 1.08, while the absolute best score achieved by this type of controller is 2.08. It is safe to say that if a controller model cannot achieve a score of 2 using the proposed search method, it is flawed.

8.1.2 Hard-coded edge and corner behaviour

This flavour of random noise generator is expected to perform in a fashion similar to the pure noise model, but with a slightly better overall fitness due to added behaviour.

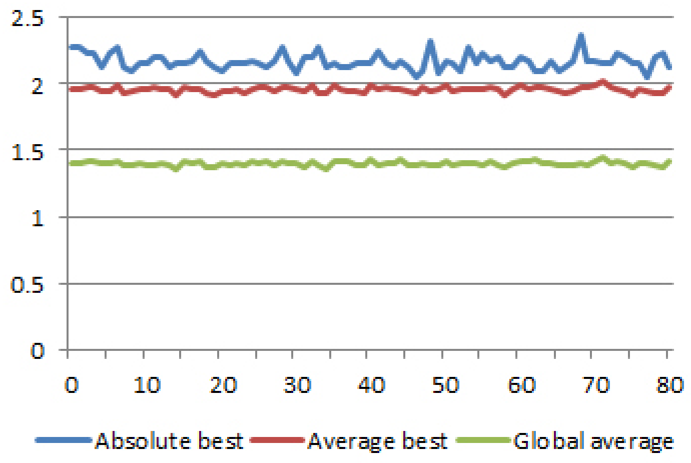

Figure 21: Results for pseudo-random noise with overridden perimeter behaviour.

The added benefit of intelligent handling of corners and walls results in a global average fitness of 1.40 and an absolute best of 2.38. The average best fitness is now 1.96, which is very close to the absolute best that just pure random noise can produce.

8.2 Baseline string controller

Baseline string controllers used in this thesis have the length of 80, as this is the minimum number of actions required to represent one full attempt at solving the Tartarus problem. Evolution parameters are chosen to replicate the original ones as closely as possible [6].

Table 3: Evolution parameters for the baseline string controller.

Parameter	**Value**
Selection mode	Tournament
Tournament size	4
Crossover mode	Single-point
Crossover probability	0.50
Mutation rate	0.40

In the original experiment that introduces this baseline model, controllers with string length of 80 achieve a global average fitness of 1.95 [6]. It is expected that the model implemented in this thesis will get similar scores.

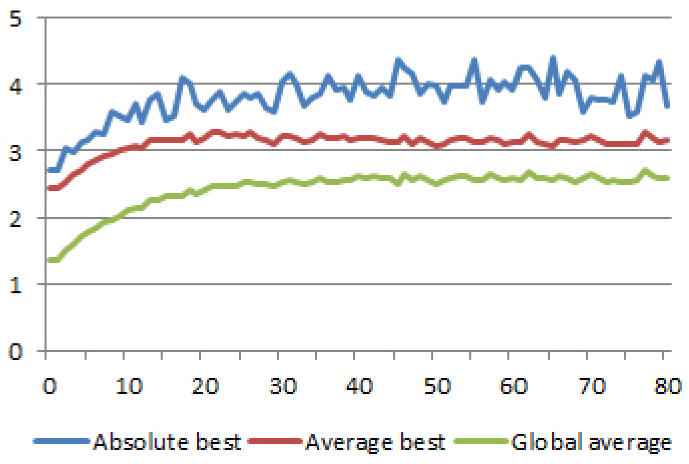

Figure 22: Results for baseline string controller of size 80.

The baseline controller performs better than expected, averaging a score of 2.42 and peaking at 4.40. This can be explained by the differences in implementation.

8.3 Model A

Subroutine lengths for model A were determined experimentally. The lengths 8, 1, 3, 3, and 3 correspond to subroutines triggered by virtual position sensor states NO EDGES, EDGE FRONT, EDGE RIGHT, EDGE BEHIND, and EDGE LEFT respectively.

Table 4: Evolution parameters for controller model A.

Parameter	**Value**
Selection mode	Truncation
Crossover mode	Single-point
Crossover probability	1.00
Mutation rate	0.25

Controller model A has a significantly shorter genome than the baseline controller so the best solution resides in an exponentially smaller search space. The mutation rate is reduced, but compensated by guaranteed crossover, which ensures variety while preventing genetic drift.

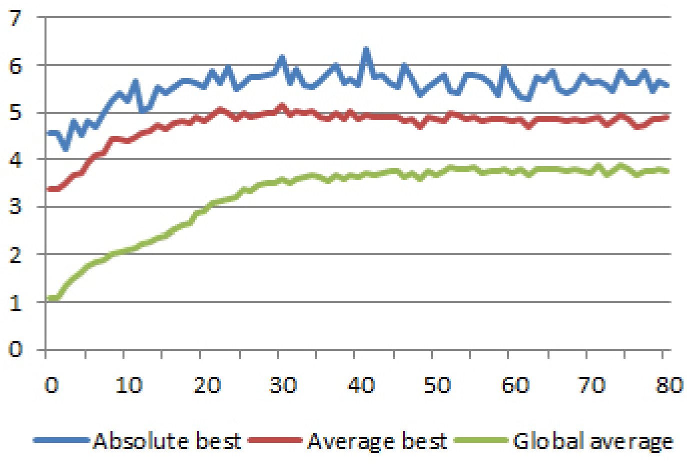

Figure 23: Results for controller model A.

The controller's performance exceeds expectations, given that it only has an arrangement of 18 action codes at its disposal and is bound to suffer from lethal genetic defects. It has an absolute best score of 6.35 and a global average of 3.24.

8.4 Model B

Subroutine lengths for model B were set according to the distribution of 247 raw sensor images that can be observed on the inner grid between the 12 subroutines that get triggered by the respective images once the simplified left-right agnostic sensor system has processed the raw input.

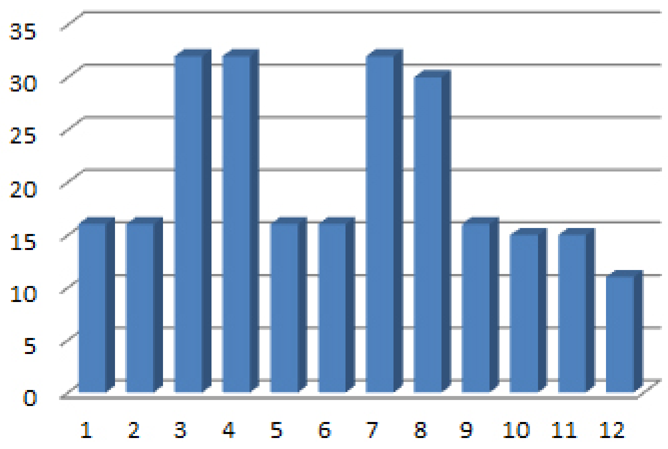

Figure 24: Distribution of raw sensor images observable on the inner grid between the 12 subroutines in model B.

The respective lengths for the subroutines are 4, 4, 8, 8, 4, 4, 8, 8, 4, 4, 4, 4. Only inner grid sensor images were taken into consideration because controller model B relies on hard-coded behaviour for wall encounters and its evolved behaviours are reacting exclusively to boxes.

Table 5: Evolution parameters for controller model B.

Parameter	**Value**
Selection mode	Truncation
Crossover mode	Single-point
Crossover probability	0.50
Mutation rate	0.15

Much less aggressive evolution parameters were chosen for this model to avoid excessive disruption of its many individual routines.

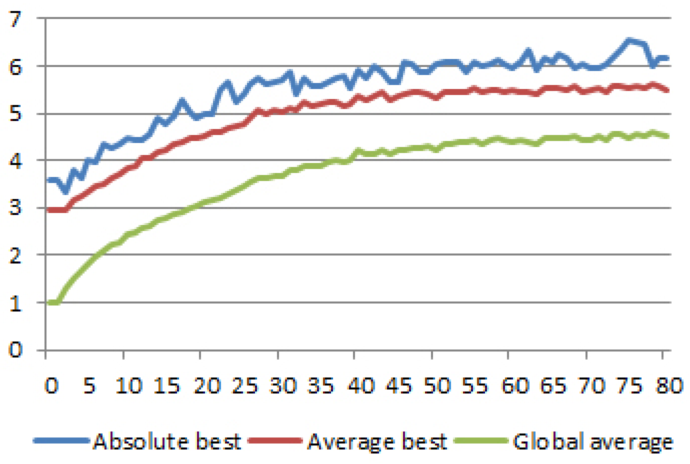

Figure 25: Results for controller model B.

Controller model B performs slightly better than model A and produces the best experimental results, reaching a maximum fitness of 6.55 while the average score across all generations is 3.67.

9 Analysis

The results of random noise generator experiments serve as the lower bound for controller performance. They are not subject to evolution and present no interest in the context of this thesis. The analysis focuses on comparison of the baseline string controller to proposed controller models A and B.

9.1 Model comparison

Both of the proposed controller models have performed significantly better than the baseline. The data obtained in these experiments is in no way an indication of the upper bound of fitness scores that these controllers can attain. Nonetheless, achieved performances demonstrate that the proposed controller models clearly offer an advantage over the baseline.

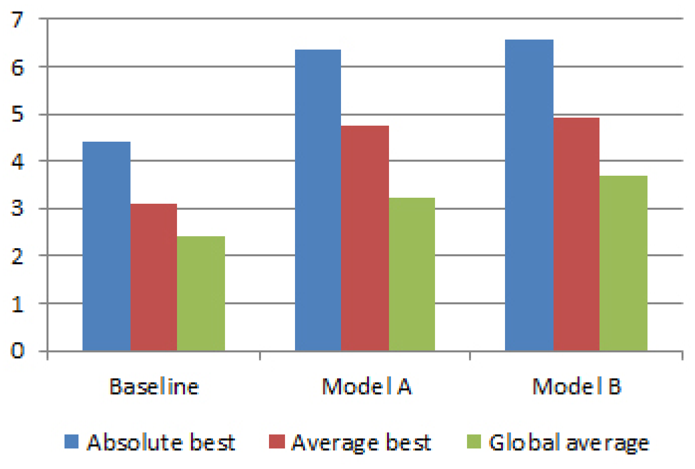

Figure 26: Comparison of controller model results.

Not only is the global average fitness of models A and B higher than that of the baseline, it also surpasses the average best for baseline controllers. Controller models proposed in this thesis operate with average best fitness scores that are 0.33 and 0.52 points higher (for model A and B, respectively) than the absolute best achieved by the baseline. The very best controllers are able to score 2 points above the baseline peak.

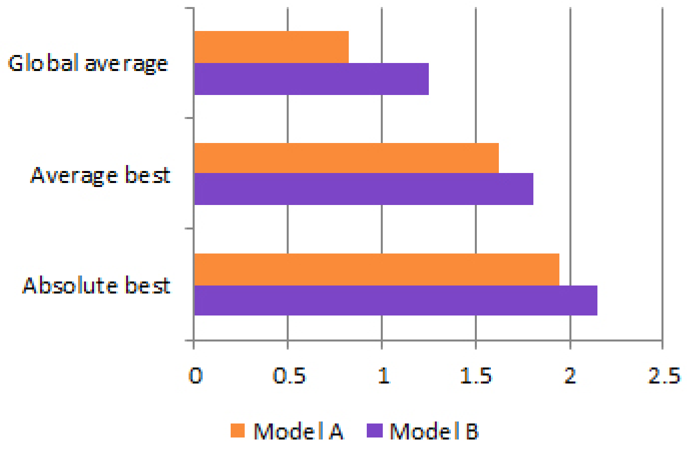

Figure 27: Advantages of proposed controller models over the baseline.

Despite being capable of much more complex behaviour, model B produces results that are only marginally better than those of model A. The only improvement worth mentioning is a 0.43 gain in average fitness across all model B controllers compared to model A.

Considering the vast difference in search space size ($3^{18} \approx 3.87 \cdot 10^{8}$ for model A versus $3^{64} \approx 3.43 \cdot 10^{30}$ for model B), it is very likely that the search space for model B holds a solution that performs exceptionally well, but remains undiscovered and requires a more thorough search procedure. It may also be the case that for this particular problem the position of the robot agent on the grid is a much more important piece of information than a general sense of box locations relative to the robot.

Different subroutine lengths are also a factor. Since shorter routines statistically tend to suffer more from genetic defects and model B has a higher routine count, its controllers will have accumulated more harmful mutations over the course of evolution. While the 5 available routines in model A are likely to be utilized at least once per trial (given that the robot agent can easily end up on any square of the grid during its run), some of the 12 routines in model B have a much lower probability of being needed due to the uneven distribution of sensor system states that trigger their execution. This can result in offspring controllers that inherit multiple inefficient routines at once from otherwise successful parents and fail to survive, taking with them whatever useful genetic information that had accumulated over generations. Model A controllers, however, are less likely to carry genetic defects further than one generation.

Overall, both proposed models display clear advantages over the baseline in every measured aspect. Based on this fact, it can be concluded that the main goal of this thesis has been successfully accomplished.

9.2 Discussion

Based on the initial analysis, the solution strategy chosen for this thesis has proven effective. However, there are certain key areas to consider for further development.

9.2.1 Success

The success of the models proposed in this thesis can be attributed to a number of factors. Naturally, the most important one is the use of state. The proposed controller models have a rigid decision-making mechanism that promotes relatively low state count while operating with a more abstract notion of subroutines rather than individual actions. Subroutines offer a highly functional way of adding state without introducing runaway state transitions encountered by pure FSM controllers [7] or problems that accompany GP-based strategies [28].

The use of sensors plays a big role in Tartarus. Experimental results presented in this thesis may suggest that information regarding the robot's situation on the grid is of great value and reactive controllers that are only aware of their position and relative orientation in the environment perform almost just as well as reactive controllers that evolve strategies that account for boxes only.

It is also important to be able to recognize trivial sub-problems and introduce partial solutions where possible to help reduce the complexity of the total problem that is tackled by the GA. An example of such a sub-problem is the corner escape logic described in this thesis.

9.2.2 Shortcomings

As far as subroutine controllers are concerned, finding a balance between over-generalization and over-specialization of models is necessary. A large number of subroutines ensures versatility and implies high reactivity in general, but requires an increase in the size of the genome for the routines to be effective. The search space grows exponentially with genome size. This makes searching for solutions significantly more challenging.

Multiple routines also present multiple points of failure and further amplify problems caused by unhelpful genetic combinations. Chaining of routines also presents a potential for wasteful action sequences being decided on, regardless of presence of apparent genetic defects in individual chromosomes. Straightforward safeguarding against such problems comes at a cost of imposing limitations that are in discord with the paradigm of GA. Balance and distribution of different alleles across the chromosome are also very important in order to maintain fully functional genetic material.

9.3 Future work

String controller studies report significantly differing results for strings of different lengths and it is concluded that the genome size plays a major role in the success of string controllers [6]. This serves as a motivation to impose genome size restrictions in order to explore strengths and weaknesses of different models within identical search spaces.

Different genetic representation can be introduced to encode fixed-size routines into single genes. This can help get rid of genetic defects present in the direct encoding method as well as outline a set of mini-routines that prove most useful when solving the Tartarus problem.

Some endeavours have benefited from collective human effort [30]. As an avenue for future work in machine learning, one option is developing an online Tartarus application to gather data from human solutions in order to train neural networks.

10 Summary

The aim of this work was to develop a GA-based search method that produces reasonably fit controllers for the Tartarus problem with results that can be related to existing literature. As a part of the main objective, two successful models were designed and later evaluated using the search method based on the canonical GA.

Baselines were provided for validation of experimental results. A trivial pseudo-random noise generator served as a lower bound for expected controller performance, while a baseline string controller found in existing literature was used to relate experimental results to the body of work that already exists on the subject.

The controller models proposed in this thesis both used the same control mechanism, but relied on different sensor interpretation methods and mixed-in hard-coded behaviours to achieve scores that conclusively surpassed both baselines, but did not display a significant gap in performance in relation to each other.

The scores achieved by proposed models arise from a combination of a multitude of factors like search space size, sensor usage, chromosome length, model-specific genetic defects and hard-coded behaviour. These factors differ between models, so the similarity of fitness scores presents grounds for further analysis, comparison and interpretation of results.

In conclusion, the main objective of this thesis has been successfully accomplished. The end result is a search algorithm that allows for further experimentation and development.

References

[1] Nolfi, S., Floreano, D. Evolutionary robotics: the biology, intelligence, and technology of self-organizing machines, 2001.

[2] Mitchell, M. An Introduction to Genetic Algorithms, 1998.

[3] NASA - NASA 'Evolutionary' software automatically designs antenna. [Online] https://www.nasa.gov/mission_pages/st-5/main/04-55AR.html (Accessed 22.05.2017)

[4] Teller, A. The evolution of mental models. – *Advances in Genetic Programming*, 1994, pp. 199-217.

[5] McDermott, J., White, D., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R., De Jong, K., O'Reilly, U. Genetic programming needs better benchmarks. – *Proceedings of the 14th annual conference on Genetic and Evolutionary Computation (GECCO'14)*, 2014, pp. 791-798.

[6] Ashlock, D., Joenks, M. ISAc lists, a different representation for program induction. – *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 1998, pp. 3-10.

[7] Ashlock, D., Freeman, J. A pure finite state baseline for Tartarus. – *Proceedings of the 2000 Congress on Evolutionary Computation (CEC00)*, 2000, vol. 2, pp. 1223-1230.

[8] Bot, M., Urquhart, N., Chisholm, K. Agent motion planning with GAs enhanced by memory models. – *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation (GECCO'01)*, 2001, pp. 227-234.

[9] Ashlock, D., Willson, S., Leahy, N. Coevolution and Tartarus. – Proceedings of the 2004 Congress on Evolutionary Computation (CEC2004), 2004, pp. 1618-1624.

[10] Zahadat, P., Katebi, S. Tartarus and fractal gene regulatory networks with inputs. – *Advances in Complex Systems (ACS)*, 2008, issue 06, pp. 803-829.

[11] Dick, G. An effective parse tree representation for Tartarus. – *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO'13)*, 2013, pp. 909-916.

[12] Luke, S. Essentials of Metaheuristics, Second Edition, 2013. [Online] Available at: http://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf (Accessed 22.05.2017)

[13] Ashlock, D. Evolutionary Computation for Modeling and Optimization, 2006.

[14] Goldberg, D. Genetic algorithms in search, optimization and machine learning, 1989.

[15] Dawkins, R. The blind watchmaker, 1996.

[16] Jebari, K., Madiafi, M. Selection methods for Genetic Algorithms. – *International Journal of Emerging Sciences,* 2013, 3(4), pp. 333-344.

[17] Umbakar, A., Sheth, P. Crossover operators in Genetic Algorithms: a review. – *ICTACT Journal on Soft Computing,* 2015, vol. 06, issue 01, pp. 1083-1092.

[18] Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P. Modeling Software with Finite State Machines, 2006.

[19] Balakrishnan, K., Honavar, V. On sensor evolution in robotics. – *Proceedings of the 1st Annual Conference on Genetic Programming*, 1996, pp. 455-460.

[20] Kim, D. Analyzing sensor states and internal states in the Tartarus problem with tree state machines. – *Parallel Problem Solving from Nature – PPSN VIII*, 2004, pp. 551-560.

[21] Gomez, F. Sustaining diversity using behavioral information distance. – *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09)*, 2009, pp. 113-120.

[22] Cuccu, G., Gomez, F. When novelty is not enough. – *Proceedings of the 2011 International Conference on Applications of Evolutionary Computation (EvoApplications'11)*, 2011, vol. Part I, pp. 234-243.

[23] What is JavaFX? JavaFX2 Tutorials and Documentation. [Online] http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm (Accessed 22.05.2017)

[24] ECJ. [Online] http://cs.gmu.edu/~eclab/projects/ecj/ (Accessed 22.05.2017)

[25] JGAP: Java Genetic Algorithms Package. [Online] http://jgap.sourceforge.net/ (Accessed 22.05.2017)

[26] The Watchmaker Framework for Evolutionary Computation (evolutionary/genetic algorithms for Java). [Online] http://watchmaker.uncommons.org/ (Accessed 22.05.2017)

[27] Jenetics: Java Genetic Algorithm Library. [Online] http://jenetics.io/ (Accessed 22.05.2017)

[28] Woodward, J., Bai, R. Why evolution is not a good paradigm for program induction; a critique of Genetic Programming. – *Proceedings of the first ACM/SIGVEO Summit on Genetic and Evolutionary Computation (GEC'09)*, 2009, pp. 593-600.

[29] Wolfram Alpha. [Online] https://www.wolframalpha.com/input/?i=convert+volume+of+observable+universe+to+planck+volumes (Accessed 22.05.2017)

[30] UW Today. Gamers succeed where scientists fail. [Online] http://www.washington.edu/news/2011/09/19/gamers-succeed-where-scientists-fail/ (Accessed 22.05.2017)