BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

EVOLUTIONARY DESIGN USING GRAMMATICAL EVOLUTION

EVOLUČNÍ NÁVRH VYUŽÍVAJÍCÍ GRAMATICKOU EVOLUCI

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **TOMÁŠ REPÍK**
AUTOR PRÁCE

SUPERVISOR **Ing. MICHAL BIDLO, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Brno University of Technology - Faculty of Information Technology

Department of Computer Systems Academic year 2016/2017

Master's Thesis Specification

For: **Repík Tomáš, Bc.**
Branch of study: Bioinformatics and biocomputing
Title: **Evolutionary Design Using Grammatical Evolution**
Category: Artificial Intelligence

Instructions for project work:

1. Perform a study of the topic regarding evolutionary algorithms (EAs).
2. Take up with basic principles of computational development in EAs. Make a focus on developmental models based on rewriting systems (e.g. grammars, Lindenmayer systems).
3. Choose a suitale type of grammar whose symbols can be interpreted as instructions, functional blocks or other entities related to a problem to be solved.
4. Implement a system for the design of grammar rules by means of EA.
5. Choose a problem (e.g. from the area of algorithm design, circuit design, decision making etc.) for which perform a set of experiments in order to demonstrate an ability of the evolutionary system implemented in item 4 to solve this problem.
6. Evaluate the results obtained and discuss potential possibilities for the future work.

Basic references:

- According to the recommendation of the project supervisor.

Requirements for the semestral defense:

- Fulfilling items 1 to 3 of the assignment, demonstration of a prototype system from item 4.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Bidlo Michal, Ing., Ph.D.**, DCSY FIT BUT
Beginning of work: November 1, 2016
Date of delivery: May 24, 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

Lukáš Sekanina
Professor and Head of Department

Abstract

Natural evolution serves as a source of inspiration for this thesis. The basic algorithm utilizes generational power of grammars in combination with evolutionary approach. The search for behavior strategies in different environments draws from evolutionary methods. Behavior trees are the model generally used to control decision making of some artificial intelligence. This thesis seeks for behavior trees which would control individuals solving the following two problems: an adjusted version of knight's tour problem and playing the game Liar's dice. When searching for a strategy of a player in a game, a competitive coevolution was implemented to mitigate the difficulty of designing a good fitness function.

Abstrakt

Evoluce v přírodě slouží jako zdroj inspirace pro tuto práci. Základní myšlenkou je využití generativní síly gramatik v kombinaci s evolučním přístupem. Nabyté znalosti jsou aplikovány na hledání strategií chování v rozmanitých prostředích. Stromy chování jsou modelem, který bývá běžně použit na řízení rozhodování různých umělých inteligencí. Tato práce se zabývá hledáním stromů chování, které budou řídit jedince řešící nasledující dva problémy: upravenou verzi problému cesty koněm šachovnicí a hraní hry Pirátské kostky. Při hledání strategie hráče kostek, byla použita konkurenční koevoluce. Důvodem je obtížnost návrhu spravedlivé fitness funkce hodnotící výkony hráčů.

Keywords

grammatical evolution, genetic algorithm, competitive coevolution, behavior trees, AI control, knight's tour, Liar's dice

Klíčová slova

gramatická evoluce, genetický algoritmus, konkurenční koevoluce, stromy chovaní, umělá inteligence, cesta koněm šachovnicí, Pirátské kostky

Reference

REPÍK, Tomáš. *Evolutionary Design Using Grammatical Evolution.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Bidlo Michal.

Evolutionary Design Using Grammatical Evolution

Declaration

Hereby, I declare; this thesis is my authorial work that have been created under supervision of Ing. Michal Bidlo, Ph.D. All sources used during elaboration of this thesis are properly cited in complete reference to the source.

. . . .
Tomáš Repík
May 24, 2017

Acknowledgements

I would like to thank Michal Bidlo, my thesis supervisor, for his willingness and friendly approach during our collaboration. His professional advices always led me the right way and helped me with reaching my goal. Special credits to Marek Kukan for development of Liar's dice simulator, which I used at the beginning and for prompt answers to all queries I had about the game. I really appreciate the help with wording, grammar and composition, from my brother Matej. Big thanks goes also to my wife, family and friends, who supported me all the way through.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 4
- 1.2 Organization . . . 6

2 Applications **7**
- 2.1 Knight's tour . . . 7
- 2.2 Liar's dice . . . 8
  - 2.2.1 Basic rules . . . 8
  - 2.2.2 Wild ones . . . 9
- 2.3 Common solution design . . . 10
  - 2.3.1 Behavior trees . . . 11

3 Evolution **13**
- 3.1 Evolution in biology . . . 13
  - 3.1.1 Reproduction in biology . . . 13
  - 3.1.2 Adaptiation in biology . . . 15
  - 3.1.3 Competition and cooperation in biology . . . 15
- 3.2 From biology to the algorithm . . . 16
  - 3.2.1 The ultimate goal . . . 16
  - 3.2.2 Population . . . 16
  - 3.2.3 Communication . . . 17
  - 3.2.4 Adaptivness . . . 17
- 3.3 Basic evolutionary algorithm . . . 18
- 3.4 Parameters of evolutionary algorithms . . . 18
  - 3.4.1 Population initialization . . . 18
  - 3.4.2 Quality of individuals . . . 19
  - 3.4.3 Mates selection strategy . . . 19
  - 3.4.4 How offsprings are born . . . 20
  - 3.4.5 Game of survival . . . 21
  - 3.4.6 Terminating conditions . . . 21
- 3.5 Evolutionary algorithm design process . . . 22
- 3.6 Coevolutionary approach . . . 22
  - 3.6.1 Multiple fitness functions . . . 23

4 Grammatical evolution **24**
- 4.1 Grammatical and developmental computing . . . 24
  - 4.1.1 Grammars . . . 24
  - 4.1.2 Parse tree . . . 25

4.2 Grammatical evolution algorithm 27
4.2.1 Genotype mapping 27

5 Implementing gramatical evolution **29**
5.1 Implementation in general 29
5.1.1 Programming language 29
5.1.2 Initial grammar design 30
5.1.3 Behavior tree activaton 31
5.1.4 Generating simplified behavior trees 32
5.1.5 Tree structure 33
5.1.6 Common evolution parameters 33
5.2 Specifics of knight's tour 34
5.2.1 Knight's grammar 34
5.2.2 Knight's code 35
5.2.3 Knight's tour simulation 35
5.2.4 Knight's fitness 36
5.2.5 Knight's evolution 36
5.3 Specifics of Liar's dice player 36
5.3.1 Liar's grammar 36
5.3.2 Liar's code 37
5.3.3 Liar's evolution 40
5.3.4 Liar's competition 40
5.3.5 Liar's fitness 41

6 Experiments **43**
6.1 Solving knight's tour 43
6.1.1 Size of population 43
6.2 Playing the Liar's dice 44
6.2.1 Are populations improving? 45
6.2.2 Comparing fitness functions 45

7 Conclusion **49**

Bibliography **51**

A List of attachements **53**

Chapter 1

Introduction

„The systematic method has the disadvantage that there may be an enormous block without any solutions in the region which has to be investigated first. Now the learning process may be regarded as a search for a form of behaviour which will satisfy the teacher (or some other criterion). Since there is probably a very large number of satisfactory solutions the random method seems to be better than the systematic. It should be noticed that it is used in the analogous process of evolution. But there the systematic method is not possible. How could one keep track of the different genetical combinations that had been tried, so as to avoid trying them again?"

Alan Turing, *Computing Machinery and Intelligence*, 1950 [14]

Engineers and scientists are scratching their heads about computers for less than a century. It was only eighty years ago when Turing started all this describing a principle in his paper *On Computable Numbers* [13]. The very principle of computers is still being used nowadays. Almost everything except the principle has changed since then.

In the early years the most valued engineers were those who knew the math well, and they led the development in the field. However, it did not took long for scientists to realize that one day, the methods based on pure math would hit their limits, and some problems would not be feasible to solve by those methods. The realization made some of them think differently. They could have continued to optimize their algorithms, search for some new methods or models as their colleagues. Instead, they started looking for inspiration elsewhere.

The obvious choice was drawing inspiration from phenomena of natural world. Scientists have always been inspired by nature, so why not this time. Even the process happaning around computers since the fifties till now is labeled with a term from biology *evolution*. The difference, between the evolution of computers and the evolution that could be observed in nature, is that one is spontanoius. The other is catalyzed by engineers and scientists. Would not it be nice, if the evolution in computers was spontanious and after a small imput form scientists continued on its own? The first one to come up with evolution in computer science was unsuprisingly Turing in his very influential *Computing Machinery and Intelligence* [14]. Nowadays, there is a huge field in computer science inspired by biological evolution, it is known by term *evolutionary computation*. The evolutionary algorithms take some starting parameters, provided by engineers. Then the algorithms work on their own producing results that are good enough, close to ideal solution. What is actually happening

behind the scenes, how the algorithms work, will be discussed later. For now the answer *evolution* will do just fine.

Keeping in mind the other half of scientists, all of whom were still developing the complicated math. One of them was Chomsky. His grammar hierarchy [3] is a pretty robust mathematical model being used in many fields. Grammars, as some rules for aligning symbols to form sentences in a language, provide some kind of surety, control or stability. They allow patterns to repeat and, with regards to the previous paragraphs, they also have something to do with evolution. In the literature these properties are hidden under the term *generational power*.

Aside from grammars, mathematicians came up with another, more practical model. To represent grammars, sentences and syntactic structures in a presentable form, they adopted a structure that is also inspired by nature. Tree is one of the basic components of natural ecosystems and it plays vital role in the computer ecosystem as well. Here is the analogy between the two trees. In the nature, tree has to provide support for its leaves. Unsurprisingly, it is done very efficiently. At the bottom there is a stable trunk (based on roots). As one follows the trunk up to the leaves, the trunk is divided into branches, which then also divide into smaller branches, and so on, all the way up to the tiniest branches, which then hold the leaves themselves. The branches are organized in a hierarchy. When following branches, there is only one path from the trunk to each of the leaves. It is a very cleverly organized structure. In the world of computers, when it comes to organizing something into a hierarchy, the structure used for that is also called a tree. Similarly, to the ones found in forests, the 'computer' tree has a stable base, usually called root. Then there is a thicket of branches connecting the leaves to the root. The leaves represent the objects being organized. The tree model is a very useful structure, unfortunately it is not clear how long has it been used nor who to admire for the brilliant idea.

1.1 Motivation

„Owing to this struggle for life, variations, however slight and from whatever cause proceeding, if they be in any degree profitable to the individuals of a species, in their infinitely complex relations to other organic beings and to their physical conditions of life, will tend to the preservation of such individuals, and will generally be inherited by the offspring. The offspring, also, will thus have a better chance of surviving, for, of the many individuals of any species which are periodically born, but a small number can survive. I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection.“

Charles Darwin, *On the origin of species*, 1859 [4]

This quote of Darwin is very powerful and relates to almost any field of science. When monarch butterflies learned to be toxic, leaving a bad taste in mouths of their avian predators, it saved their species. Their were not eaten unlike other butterflies. However, viceroy butterflies topped their fellow monarchs. They learned that they didn't have to be toxic to stay alive. Simply appearing like monarchs was enough to fool the birds and easier to evolve. These are both examples of natural selection in biology. When a mathematician discovers a new theory, a slightly better one than the previous, and the public adopts it (other mathematicians start to cite it, teachers start to teach it, engineers start to build new technology based on it), that is a natural selection in mathematics. When a chef invents

a new recipe to cook pork and customers find it more delicious than the pork from the restaurant across the street, they stop eating the pork from across the street. Want it or not the chef from across the street buys the new pork recipe from his innovative competitor and starts serving the same pork, in order to keep customers coming to his restaurant. However, his old recipe is not used any more, it died and that is the natural selection in gastronomy. Applying the principle of natural selection in computer science resulted in evolutionary algorithms.

It looks very promising indeed, but there is one big downside of the evolution: it takes time. The earth is 4543 billion years old and it took it a while to evolve to the current state. The monarchs and viceroys did not evolve their tricks for survivor in a year. It took them more than hundred million years. Billions of individuals had to die, before their genetic sequence was successfully altered and evolved their life saving features. New mathematical theory also takes weeks or even months to be adopted by public. Similarly, a word about a new delicious meal has to spread for more than a week to reach all the possible customers. People do not have that luxury of time in computer world. If one wanted to sell an awesome algorithm that solves a problem brilliantly, but it would take a week to evolve the solution, nobody would be interested in buying. Luckily computers are able to accelerate the evolution, thus the evolutionary algorithms are feasible.

So there is an algorithm to solve problems, but how should one interpret these problems to the algorithm? The evolutionary algorithms are a method of solving problems, but the problems need to be specified in a way that a computer understands. Some sort of problem abstraction is needed. The algorithm could be tightly bound with the problem it is solving. Instead a method that could solve many different problems is preferred. An algorithm that takes a problem representation as an input and returns a solution to the problem as a result of its work.

When one explains his problem to his friend or his health troubles to his GP[1], he uses a language. To communicate with a computer programmers also use languages, although the programming ones. Thus to explain some problems to the algorithm, some sort of a language is going to be used as well. A handy tool for language specification is the formalism presented by Chomsky *grammars*.

What kind of result is expected from the algorithm? It probably depends on the problem itself, so a restriction has to be made in here. This thesis focuses on problems, with some courses of action as solutions. Once again, there is a need to understand the results from the algorithm, and a need to easily use the results to ones advance. This issue will remain unanswered for now and instead the goal of the thesis is introduced.

Close examination of available methods of evolutionary computing, development and grammatical evolution, investigation of properties of grammars and study of the ways grammars are constructed is crucial. Based on the gathered knowledge an algorithm for solving problems is going to be designed. This algorithm should leverage both Darwin's *natural selection* and Chomsky's *grammars*. By applying this algorithm to some problems of real world, the theoretical considerations would like to be proven. The very common problem of knight's tour is slightly adjusted to be the first lab rat. If the rat survives, meaning the algorithm works, a true real world problem is going to be tackled. The evolution will be let to handle development of an AI player for the game *Liar's dice*.

[1]GP - general practitioner

1.2 Organization

The work is split into chapters, which step by step apprise the reader with new ideas being applied to the solution. The paper starts with an introduction to problems 2 it is trying to solve. A very high level outline of the solution is presented in section 2.3, together with a model that the solutions are based on 2.3.1. The means to help solving the problems follow in chapters 3 and 4. Firstly evolution is examined and the key points for defining the basic evolutionary algorithm 3.3 are discussed. To better understand all the parameters of evolutionary algorithms see section 3.4. There are descriptions and also reasoning for the choices made later during experiments. In adition to the straight evolution, the coevolutionary approach is introduced in section 3.6. Basic knowledge about grammars, needed for the purpose of this work, is presented in sections 4.1.1 and 4.1.2. Answers regarding the last missing part of the algorithm are in section 4.2.1. The grammatical evolution algorithm itself 2 can be found in the section 4.2. Digging deeper into the details of the solution is chapter 5. It starts showing the parts that are common in both problems 5.1 and explains the issues found during the implementation and experimenting. Specifics of the implementation like the design of grammars or implementation of the operations is described in sections 5.2 and 5.3. In chapter 6 the most interesting experiments are presented, with some conclusions being made. There is one appendix A to this work. It is a description of files attached on the digital part of this thesis.[2]

[2]These files could be also found on the following url https://github.com/trepik/ge

Chapter 2

Applications

Coming up with a new idea, method or algorithm is very satisfying, but using it to solve some real problems makes one even more pleased. Choosing the right application domain was not easy. When opting for an evolutionary approach the problems need to be complex enough. In case the solutions could be easily computed with the conventional methods, evolution is not needed. Problems can not be too complex either, because of the computational power that the evolution needs. In order to develop the algorithm and conduct some experiments, on a mediocre personal computer, the solution of the problem has to be easily evaluated.

Navigating a knight over the chessboard and not stepping on the same square twice, is the first application. The environment of the chessboard is pretty simple and the behavior of the knight is straightforward. This problem was also chosen for the ease of its implementation. The second application, behavior of a dice player is definitely more complex. In a game with dice, where probability plays one of the main roles, finding the right strategy is not a piece of cake. Leaving it to the evolution is a lazy man's approach, but also a good proof of concept.

In the literature, there have been many different applications of grammatical evolution. This theses tries something yet new, but similar. The inspiration came from *Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution* [7], where the authors control Mario with evolutionary developed behavior trees. This work aims to achieve something similar with the two scenarios. In the following sections 2.1 and 2.2, the principles and the rules of both problems are explained, while the last section 2.3 outlines a common strategy for solving the problems.

2.1 Knight's tour

In order to know how to catch a big fish, one needs to know how to catch a small one first. In this section a simple problem is presented, so that a more complex problem could be solved later. The original knight's tour is a chess problem. Chess is played on a board of 64 squares organized into a square pattern (8x8). A knight (also known as horse) is one type of pieces, chess is played with. Knight's specialty is his L-shaped movement over the board as shown on figure 2.1.

Definition 2.1.1 (Knight's tour[1])
Knight's tour is a chess problem in which a knight makes a circuit of the board touching each square once.

A restriction of knight ending one move from the starting position makes it a *closed* knight's tour. This closed version is actually an instance of more general mathematical problem *Hamilton circuit*[2]. Computing knight's tour for any initial position on the board is considered solving the knight's tour problem. One could even question, whether there is a solution to the problem, but in fact there are several billion solutions.

This thesis does not seek for a sequence of moves, but rather a behavior tree that controls knight's movements on the board. So that the one sole result (a behavior tree) is able to maneuver the knight over the board according to the rules of knight's tour. A term behavior tree has been mentioned several times without any further explanation. Detailed elaboration on the topic is in section 2.3.1.

There is no need to restrain the knight's tour problem to 8x8 chessboard. Experimenting with boards of different sizes may be interesting. The smallest board on which the knight is able to finish his tour is 5x5 borad, and experiments on smaller boards would not make much sense.

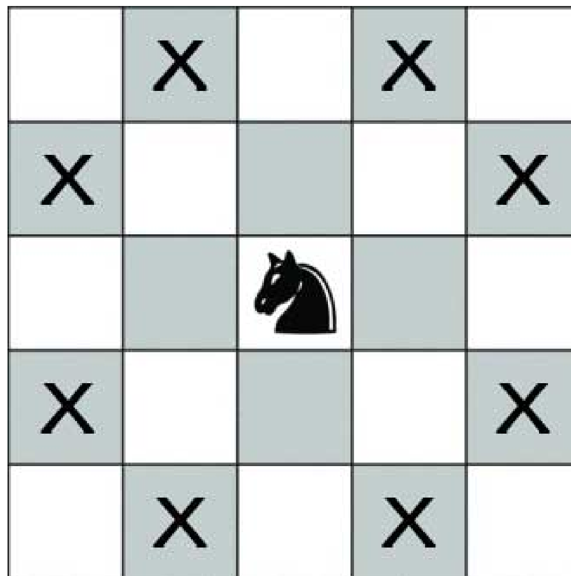

Figure 2.1: All possible knight moves.

2.2 Liar's dice

In this section a game of dice is presented. There are many different variations and also names for this game. Only one set of rules is chosen to base the solution on. The *Single Hand* version of rules from dicegamedepot.com [5] is implemented. To reference the game a name *Liar's dice* is used from now on. A detailed explanation of the rules follows.

2.2.1 Basic rules

The game is played by two or more players ($p \in \mathbb{N} \land p \geq 2$). Each player starts the game with the same number of 6-sided dice d. The game is further divided into smaller closed

[1]Definition from http://www.merriam-webster.com/dictionary

[2]More about Hamilton circuit at http://mathworld.wolfram.com/HamiltonianCycle.html

segments called rounds. The order of the players in the first round is random. Each round begins with each player rolling his or her dice at the same time. Everybody keeps his or her roll to him or herself. The player to start the round states his or her bid consisting of a value v and quantity b. The bid represents the player's belief that there are at least b dice of v value rolled by all players together. Each subsequent player has three options. He or she can:

- make a higher bid, or
- re-roll, and then make a higher bid, or
- challenge the previous bid.

Making a higher bid can be done in two ways:

- either increasing the value of the previous bid and not lowering the quantity ($v_{new} > v_{old} \wedge b_{new} \geq b_{old}$), or
- raising quantity of the previous bid and choosing any value ($b_{new} > b_{old}$).

If a player is not sure about making a higher bid based on dice he or she sees, he or she can uncover at least one of his or her dice and roll the rest again. Then he or she has to make a higher bid. The bidding, or re-rolling and bidding goes on until one of the players decides to challenge the previous bid. With a challenge comes the end of a round and everybody uncovers his or her dice. The challenged bid and the actual state on the table are compared, giving three possible outcomes. ($r = b - a$ where a is the actual quantity of v value on the table)

- $r < 0$ - There are more dice of the bidden value on the table than the actual bid, and therefore the challenger looses. He or she has to put aside (for the rest of the game) $-r$ of his dice.
- $r = 0$ - Both the actual quantity on the table and the bidden quantity are equal, which makes once again the challenger loose. He or she has to give one of his dice to the bid caller.
- $r > 0$ - There are less dice of the bidden value on the table than the actual bidden quantity, making the bid caller loose. He or she has to put aside (for the rest of the game) r of his dice.

The loser of a round always begins the following one[3]. The game is played in an elimination fashion, once a player has no dice to roll, he or she is eliminated. A winner is the player who stays last in the game, having at least one dice to play with.

2.2.2 Wild ones

There is one extra rule added to spice up the game a bit. The value of 1 is considered as wild, always counting as the value of the current bid. When a bid caller announces that the bid value is fives, this rule of wild ones turns all of the ones rolled into fives. The ones always match the bid value. The value of ones can also be bidden, and their quantity counts

[3]Very likely situation to happen is that the looser of a round is eliminated and therefore can not start the next round. A player right after the eliminated one in the playing order is to begin the new round.

double. To top a bid of normal values with a bid of wild values, the new quantity times two needs to be higher than the previously bidden one. Here is an example of a bid order: $5\ sixes < 3\ ones < 6\ sixes$. With this new rule the game becomes a bit more complex, bringing more possible strategies into the frame.

The rules of the game were explained, but to make sure the reader understands them well an example of few rounds of the game is presented.

Example 1

There are three players in the game and Mike is to start the first round. Everybody rolls his or her dice, but keeps it hidden from the others. Mike rolled 1 1 2 5 3 5 and he bids 4 ones. Then both Emily and Josh increase the bid to 5 and 6 ones. Mike is suspicious, so he calls a challenge. Emily and Josh reveal their dice to Mike and make him sad. Both Emily and Josh had 2 ones each which makes it total of 6. That is exactly as the Josh's bid said ($r = 0$) and Mike has to give one of his dice to Josh. The second round is here and Mike starts it because he had lost the last one. He rolls 3 5 2 1 6 and bids 4 sixes. Both of his opponents once again increase the bid quantity by one up to 6 sixes. Mike does not want to end up in the same trap as before so he decides to reveal 1 and 6. He re-rolls his remaining three dice and gets 3 5 6. Luckily for him he rolled one extra six and with satisfaction calls a new bid 7 sixes. Emily decides to challenge Mikes bid only to find out that there were exactly 7 sixes rolled on the table. As in the first round $r = 0$ and Emily passes one of her dice to Mike. She also starts the next round. All the dice are rolled and she announces a bid of 1 six. Josh calls 2 sixes and Mike does the same and increases the bid to 3 sixes. This continues in a similar fashion until the bid is 7 sixes and Mike decides to call a challenge on Emily's bid. Everyone reveals their dice showing:

- Mike: 6 1 4 5 3 2
- Josh: 1 2 2 6 3 2 4
- Emily: 5 4 3 3 5

There are only 2 sixes which would mean Emily had to give away all of her 5 dice. Luckily for her, there is the rule of wild ones and the 2 ones on the table count as sixes. $r = 7 - 4 = 3$ and she loses only three of her dice leaving her with two. There is no need to continue with the game as most of the rules should be clear by now. Who is going to win the game, the reader would never know.

2.3 Common solution design

Both of previously outlined behaviors, either a knight traversing a chessboard, or a player trying to win a game of dice, have some similarities. They need to stay within some boundaries or rules. The same approach will be used trying to solve both problems. To make them act according to the rules, grammars will be used. To make them improve and search for better strategies, evolution will be applied. And finally, to represent and evaluate evolved solutions, behavior trees are employed. Close analysis of these problems follows.

What is a solution to the knight's tour problem? It is a sequence of moves that maneuvers the knight over the board. Similarly, when playing Liar's dice the algorithm should be looking for a sequence of moves that lead to a victory. Looking for a single sequence, could lead to a proper solution of the knight's tour from one place on the board. However, changing the initial position of the knight, would probably require a new, different sequence

of moves. The same with the Liar's dice, one strategy leading to victory in one game, might be terrible in another. So instead of a single sequence of moves the goal is to look for multiple sequences. They do not necessarily need to be unique sequences and could differ from one another only slightly, they may overlap. To avoid redundancy, the overlapping parts of the sequences can be merged and organized into trees. These trees should form a hierarchy of strategies. By following a certain branch of a tree, player would be applying a certain strategy. To follow a certain branch means deciding, which branch to follow next at each branch division. These decisions should be based on some observations of the environment. In case of the knight he should be considering his position and free slots he can move to. A player of the Liar's dice would observe the dice on the table he can see, as well as the dice that are hidden to him. Similarly he should choose his next move based on the current bid.

So the trees will not consist of actions only, but conditions for testing the state of the environment should be incorporated as well. In general, this tree is describing behavior of a player. *Behavior trees*, as they are called, are widely used in computer games, to define behavior and actions taken by artificial players. More on behavior trees follows in next section.

2.3.1 Behavior trees

There are many different types of behavior trees all over the internet. This work is inspired by the behavior trees described in *Natural Computing Algorithms*[2]. As mentioned before, behavior trees are organizing behaviors or player strategies into hierarchical structures. Higher-level behaviors appear closer to the roots, while the primitives stay in the leaves. There are two types of leaf nodes in the behavior trees. Action nodes include primitive behaviors and condition nodes represent the states of environment. Usually there is a condition node followed by some action nodes, corresponding to the behavior of a player in a game. At first the player examines the state of the game (condition nodes) and then he or she takes some actions.

Aside from leaf nodes, there are composite nodes. They are connecting leaf nodes to form the structure of a tree. According to the terminology of the tree data structure, composite nodes are parenting leaf nodes. Having the structure connected, it is time to explain how the structure is used to control a player in a game.

Each of the nodes of a behavior tree has some code attached to it. This code differs with the type of the node. Aside the code each node can be in one of four different states: *unknown*, *active*, *succeeded*, *failed*. At the beginning each of the nodes is in the *unknown* state. The decision making starts by moving the root node to the *active* state. Activating a node means executing its code. By executing the code a node can move from the *active* state. The action nodes directly interact with the environment. If the code of an action node successfully alters the environment, the node moves to the *succeeded* state. If something goes wrong and the action is not accepted by the environment, the resulting state is *failed*. The condition nodes, upon activation, change their states according to a test on the state of environment. Each of the condition nodes tries to get some information about the current state of the environment and tests a condition, which is either true or false. If a condition is evaluated as true the condition node moves to *succeeded*, and if the condition is not true in the environment, the state changes to *failed*.

To be able to model more complex structures, two types of composite nodes are going to be implemented. A sequence composite node is going to activate all of its child nodes one by one, always in the same order, while they move to the *succeeded* state. As soon as one of

the nodes moves to the *failed* state the remaining child nodes are not activated. It is similar to the lazy evaluation of conditions in programming. The sequence node represents the *and* operation. The second type of composite nodes is a selector, activating all of its child nodes until one moves to the *succeeded* state. The behavior of selector node is equivalent to the *or* operation in lazy evaluation of conditions. One last note is that the root is also a composite node and its type depends on the application.

There are many other types of nodes in the literature like decorators, invertors, succeeders, repeaters or limiters[9]. However, this work does not attempt to make use of them and it implements only the types presented in detail in this section.

The construction of behavior trees is one very demanding and time consuming task. Very good knowledge of the environment and scenarios is required. Instead of a deep analysis and observations on the environment, the evolution is used to handle the problem. More on how the trees are generated and used to solve the problems is in section 4.2.

Example 2

Here is an example of a behavior tree with both selector and sequence composite nodes. With regards to the leaf nodes, each condition precedes an action.

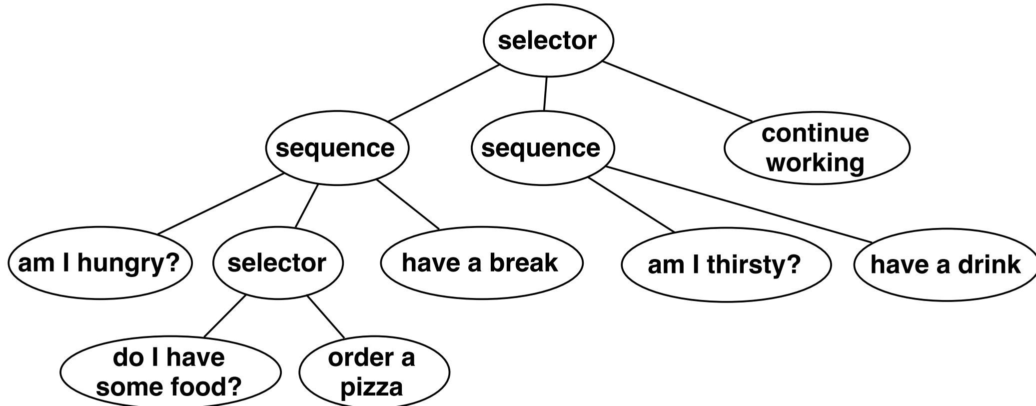

Figure 2.2: Behavior tree example.

Chapter 3

Evolution

When one wants to draw inspiration from natural evolution, firstly he or she has to know, how does it actually work. The basic vocabulary and concepts of evolution are introduced. Some of the most important terms are defined solely for the purpose of this work. They may or may not overlap with definitions in literature. Only key ideas are highlighted and developed step by step making the text easy to follow. The focus is on methods used in the experiments closely described in chapter 6.

3.1 Evolution in biology

When speaking about evolution, there is always a subject. The subject could be one individual or a group of individuals. This work focuses on more individuals interacting with one another calling the group of individuals *population*.

Definition 3.1.1 (Population)
Population is a community of individuals among whose members interbreeding occurs.

When population is static, there is not much to observe, because there si no evolution. This implies the changes in a population are going to be investigated. In particular there are two changes of interest and that are births and deaths. It is possible to cover these two changes with one term. The father of the evolution theory Darwin wrote about *natural selection* in his *On the Origin of Species* [4]. Shortly after reading this paper Spencer implied the famous *"Survival of the fittest"* quote. *Survival* is the term encompassing births and deaths, and itself it is a very challenging problem. For successful survival one needs to have available resources, such as food, water, shelter and mates. Certain mechanisms or skills like senses, communication, cognition and mobility evolved to help individuals survive. Ability to reproduce is undoubtedly a key factor in survival as well. These are the properties of natural evolution that the algorithms presented in this thesis are inspired by.

3.1.1 Reproduction in biology

This section has a closer look at the process of reproduction. Because the process is pretty complex, it is explained in brief. The emphasis is on the key parts that the algorithms are leveraging.

Definition 3.1.2 (Reproduction)
Reproduction is a biological process of new individuals being produced.

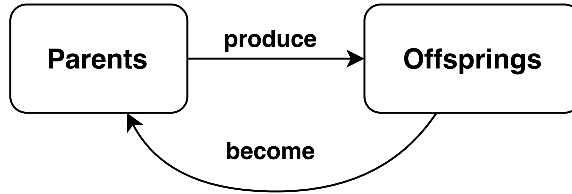

Figure 3.1: Reproduction.

Producers are called *parents* and the product is called an *offspring*, as shown on figure 3.1. From the algorithm development point of view, the most important concept in reproduction is the transfer of genetic information. This information is stored in each and every cell of an individual as a sequence of nucleotides. A nucleotide is an organic molecule serving as the basic building block for genes. One can imagine the genetic information as a recipe or a manual for a specific feature. For example there is a genetic information saying what eye color a person has, or would have. The information is called genotype and it is directly linked with a similar term phenotype. Definitions of both terms follow.

Definition 3.1.3 (Genotype)
Genotype is a genetic information responsible for a particular characteristic.

Definition 3.1.4 (Phenotype)
Phenotype is a particular characteristic or physical expression defined by some genotype.

Within the same species genotypes differ only slightly. As the species phenotypes differ more the same is with their genotypes. On the other hand there are species having very similar genotypes but their phenotypes differ a lot. That is possible because genotype is not necessarily the entire genetic information of an individual, genotype is usually only some proportion.

Definition 3.1.5 (Genome)
Genome is the complete set of genes or genetic material that an individual poses.

In the process of reproduction parents duplicate their genome and pass it to their posterity. There are two forms of reproduction. The first one is called asexual. Only one individual is involved. The whole genome of a single parent is duplicated and its offspring gets identical or almost identical genetic information. Although the population tend to grow in numbers exponentially, the genetic evolution in asexual reproduction is making very small steps. The only changes between parental and offspring genomes are *mutations*.

Definition 3.1.6 (Mutation)
Mutation is a permanent alteration in the nucleotide sequence.

Individuals reproducing asexually tend to adapt slower in terms of the number of individuals. More individuals need to reproduce, in order to ratify a change in genotype. Vulnerabilities of asexually reproducing individuals usually stay longer within the population, by traversing from parent to offspring. Types of asexual reproduction observed in nature are for example budding or binary fission.

The second form of reproduction is sexual. There are two individuals required to sexually interact in the process of reproduction. Genetic information of both parents is combined to create completely new genome of their offspring.

Definition 3.1.7 (Crossover)
Crossover is an exchange of genetic information between two homologous genetic sequences.

The crossover brings a new variety of genomes to the population. Individuals can therefore survive in changing conditions, by adapting to the environment faster. Less individuals need to reproduce in order to ratify a change in the genome. In contrast to the asexual reproduction, sexual makes much larger steps in evolution. Of course there is no guarantee that all the steps are in the right direction. The natural selection decides whether the change in genotype was right or not.

The evolutionary algorithms implement both forms of reproduction, each one for a different purpose.

3.1.2 Adaptiation in biology

The evolution and reproduction are processes helping populations survive. The other process happening behind the scenes is *adaptation*. It occurs in multiple levels and timescales. Good illustration is the POE model presented by Sipper et al. in 1997 [11]. Their model distinguishes between three levels of adaptation. *Phylogenesis* would be the adaptation of genomes as mentioned in section 3.1.1. This is actually the slowest level of adaptation speaking from the number of generations perspective.

Definition 3.1.8 (Generation)
Generation is a collection of all individuals of a certain population living at a certain moment.

Ontogenesis is the second level of adaptation and covers the development of an individual during its maturation (differentiation, becoming an adult, etc.). It can be viewed as a process of genotype to phenotype mapping. The genome stays the same while the individual still adapts according to its surroundings. It is possible because the genetic information is not being used all at once. The process responsible for this behaviour in nature is one of gene expression regulation processes, known as methylation[1].

Epigenesis might seem very similar to the Ontogenesis. It is also a process of adaptation or development of an individual, and it is also hugely affected by the environment. However, the development is not engraved in the genome of the individual. The name epigenesis reflects the fact, meaning something beyond genetics. One can imagine this level of adaptation as a lifetime learning of an organism. Epigenesis type of adaptation relies on experience gathered in sort of a memory.

It is true that in nature all these three levels of adaptation are closely interlinked, but when searching for inspiration in the process of algorithm design, there is no problem in separating them.

3.1.3 Competition and cooperation in biology

In the natural environment among many species of animals or plants there is a continuous interplay, encounters or interactions of different kinds. Two different species are either cooperating in order to survive in the environment or there is a competition for resources between the species. The cooperative behavior is called *symbiosis*. An example is the symbiosis of the Nile crocodile and the Egyptian plover, a bird that feeds on any leeches attached

[1] More on methylation could be found in the work of Bell et al. [1]

to the crocodile's gums, thus keeping them clean. The competitive behavior resembles an *arms race*, when each competitor evolves to overcome the latest tricks of its opponent. The pray-predator conflict between lions and antelopes, is a perfect example. The environment, too, is continuously changing and preparing new challenges for the species, so they have to adopt to it as well. This observation suggest there is more to be investigated on the two behaviors in evolution: *cooperation* and *competition*.

3.2 From biology to the algorithm

Section 3.1 introduced evolution from the biological point of view. Now it is time to use this knowledge to construct an algorithm. This section is going to prepare the ground by simplifying and slightly tweaking some concepts. The evolution algorithm itself is presented in the following section 3.3. Together with the presentation of the algorithm building blocks, this section contains reasoning for each of the choices made. The rest of this chapter is inspired by the first chapter from Springer's *Natural Computing Algorithms* [2].

3.2.1 The ultimate goal

Before getting started with preparing the building blocks for an algorithm, one needs to know, what is he or she trying to achieve. What is the ultimate goal of this thesis? An algorithm in general is a step by step procedure designed to solve a problem. So evolutionary algorithm should be also able to solve problems. The important here is the plural of the last word *problems*. Yes, this work aims to generalize as much as possible, so that the algorithm could be used to solve not only one but many different problems. It needs to be universal. If the algorithm would take ages to finish, it would not be very convenient either. Therefore, a good performance is also one of the targets. It all looks very promising, but there is one small problem. No one is going to get all this for free. In order to achieve the listed goals, one needs to give something up, and that something is precision. Why? Nothing is really perfect in the real world, so there is no need to have perfect solutions to the problems either. All in all the algorithm or a problem solving mechanism should be both universal and effective, and should produce solutions that are close approximations of the ideal solutions.

3.2.2 Population

The term population was already defined and it has also been said that the focus will be on evolution of a population. But why? Section 3.1.1 partially answered the question, when it spoke about parents and their offsprings. Making a connection to the ultimate goal of solving problems, population is one of the key components of the algorithm. Instead of working on one solution to a problem, the evolutionary algorithm works with multiple solutions at a time. These potential solutions make up a population of solutions. It is a huge improvement in contrast to the conventional methods. By working with a population one imposes parallelism. It is not necessarily a speed up gain, but this approach brings many other advantages.

With a population of potential solutions, one is trying to avoid convergence to a local optima of a problem. Then there is no need to restart the algorithm over and over again, because of an 'unlucky' initialization. There is much higher probability that one of the potential solutions in population gets close to the global optima. The vital point is to

disperse the initial population evenly to cover the entire solution space. In fact it is beneficial to maintain the dispersion during the entire evolution. The algorithm is then able to adopt to changing conditions of the environment. The global optima might also be changing in larger distances over time and a dispersed population has a higher chance of capturing these changes. The population mainly allows the algorithm to avoid local optima and achieve a global search characteristics through dispersion of its individuals.

3.2.3 Communication

Communication, or exchange of information, between individuals within the population is crucial. Working in isolation would not be much different from the conventional methods. Yet interacting with peers, who are trying to achieve the very same goal, is profitable. The exchange of genetic information between parents and offsprings is the actual way of communication, individuals are able to do. When some individuals are closer to the target, advertising it to the others is vital. Being the evolution in changing environment, communication can be beneficial as well, but only in well dispersed population. One can spot a conflict of interests in here. On one hand the algorithm should guide individuals to better places, solutions, on the other hand it should keep the population dispersed all over the solution space. Finding the right balance is key here. Wise communication can help bias the algorithm towards better solutions, while maintaining dispersion.

3.2.4 Adaptivness

Section 3.1.2 introduced the POE model [11], looking at the process of adaptation from three perspectives. The algorithm is going to apply principles coming from phylogenesis. Individuals are going to carry the same genetic information during their entire life. All of it is going to be used at all time, that means no methylation and thus no change in behavior. Individuals will have no memory, so they will not be able to apply the principle of epigenesis either. The population is going to adopt by passing the genetic information from one generation to another. Each new generation should be at least as good as the previous one, but it is not a necessity.

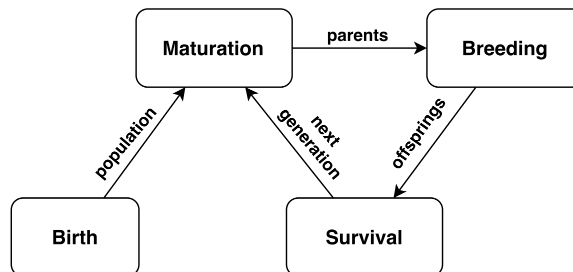

Figure 3.2: Evolution cycle.

Before the algorithm is built up, a brief recapitulation of the key characteristics of evolution follows. At the beginning a population of individuals is born. Some of them mature, find their mates and become parents. After breeding, new individuals or offsprings are born. Because of new individuals joined the population, there is not enough resources for all of the individuals. Darwin's *Natural Selection* [4] comes into play and a game of survival is played to form the next generation. As figure 3.2 shows, the cycle is already closed. The process repeats over and over again, which makes it the evolution.

3.3 Basic evolutionary algorithm

All the necessary building blocks to define a basic evolutionary algorithm were provided. The algorithm simulates the process of biological evolution (phylogenesis), in order to find a satisfying solution to a problem. At the beginning initial population of potential solutions is generated, so that it is evenly dispersed over the solution space. The solutions are iteratively modified and preferably improved from one generation to another. Some solutions are selected to become parents, exchange information, and thus create new solutions. Next generation is formed from both parents and their offsprings, using variety of strategies. Most of the strategies are based on the quality of individuals. Better solutions are preferably selected for survival. However, it is important to keep in mind the goal of solution space coverage. The algorithm ends when one of the solutions meets a predefined criteria. In other words the solution is good enough to survive in the environment and solve the problem.

Algorithm 1: BASIC EVOLUTION [2]

```
1 population.init()
2 repeat
3     parents = population.getParents()
4     offsprings = parents.breed()
5     population.update(offsprings)
6 until terminating condition
```

As the reader can see each individual step of the basic evolutionary algorithm is very abstract and can be implemented in many different ways. That implies existence of multiple different evolutionary algorithms, all of which have the same starting point the *basic evolutionary algorithm*. Section 3.4 describes each step of the basic evolutionary algorithm in more detail. Different options for each step are presented with highlighting their pros and cons.

3.4 Parameters of evolutionary algorithms

Without a proper implementation details the previously described procedure is just a rephrasing of the biological process. In order to make the evolutionary algorithm complete and ready to use, sections 3.4.1, 3.4.2, 3.4.3, 3.4.4, 3.4.5 and 3.4.6 speak about each step in detail. The most common approaches are presented, emphasizing the ones used in this work. The algorithm is almost ready, only waiting to asign missing values to its variables. Now it is time to examine the potential values for each of the variables. This section is inspired by chapter 3 of the *Natural Computing Algorithms* [2].

3.4.1 Population initialization

If one already knows some solutions to the problem he or she is trying to solve with evolution, these information can be without any harm used to initialize the population. Usually the engineers end up on the other side of the river, where they do not know any good solutions. The population is then in most of the cases initialized at random. Speaking about initialization of the population, one might be asking how many individuals should there be? There is no universal answer to the question. It is one of the parameters of each

evolution algorithm. The size depends on many aspects like computational power of the hardware, specifics of the problem being solved and other parameters of evolution as well. Usually the size of the population stays the same the entire evolution, but there might be cases where growing or shrinking population is useful. Size of the initial population is most often chosen experimentally.

3.4.2 Quality of individuals

After an individual is born, it is vital to get some information about its qualities. Without it the natural selection could not compare the individuals with each other and would be helpless. To enable the natural selection a measure of fitness is established.

Definition 3.4.1 (Fitness)
Fitness is measure of capability to solve a certain problem.

From now on, when comparing two individuals, one is going to be fitter. In other words one is going to have a better fitness, than the other. The actual fitness measurement or computation is problem specific. Designing a good measure of fitness, or so called fitness function, is also a crucial part in the algorithm specification. It is the second parameter. Evaluating individuals is usually the most resource consuming part of the evolutionary algorithm. That is why one needs to keep the fitness function as simple as possible. At the same time the fitness function must well distinguish the individuals, to drive the evolution forward. To keep the algorithm efficient, one does not want to be evaluating the same individuals multiple times, and rather use the fitness stored somewhere. However, it is not always possible and in some cases the fitness of an individual needs to be calculated at each generation.

3.4.3 Mates selection strategy

Motivation behind breeding is simple, newly generated individuals should be fitter, better suited for the environment. Therefore tending to choose better solutions to mate is intuitive. However, it is not always the best policy. Selection strategy, dictating which individuals are going to provide their genetic information, in order to breed an offspring, determines the *selection pressure*.

Definition 3.4.2 (Selection pressure)
Selection pressure is amount of bias towards selecting fitter individuals for breeding.

There are two extremes, one of which is pressure being too low. In that case the evolution is usually too slow and inefficient. The other extreme of pressure being too high, results in an evolution not maintaining the population dispersion, thus possibly converging to a local optima. Finding the right balance is key here. The original method for mates selection is *fitness proportionate selection*. A probability of an individual being selected is directly related to its fitness. The selection itself is based on a roulette principle, where the probability of selection is transformed to a space on a roulette wheel. The method is very intuitive, but it embeds a high selection pressure. Generally, in the early stages of the algorithm, there are many different solutions with some being noticeably better. The better ones are selected by roulette more frequently and thus push the population to a premature convergence. There is also a risk of low selection pressure occurring. In the later stages of the algorithm the individuals have more or less the same probability for selection, and

slightly better solutions are not able to strongly influence future populations. To overcome the selection pressure problems, another approach was introduced *ordinal selection*.

In ordinal selection, the fitness measure is computed in two steps. Firstly all solutions are ranked according to their original fitness. Fitter solutions get higher rank. Then the rescaled fitness values are computed using the ranks. That should lessen the risk of high selection pressure in the early stages of the algorithm. The roulette principle of selection could be applied as before but there are other more sophisticated options. *Top rank selection* method looks only at the top n ranked individuals and chooses mates from those. The most commonly used method of ordinal selection approach is a *tournament selection*. From k individuals, chosen randomly without replacement, the fittest one always 'wins the tournament' and is selected for mating. The k parameter is very interesting. With low values of k the selection pressure is low, while high values of k result in higher pressure.

Mates selection utilizes only phenotypes (fitness of individuals) and thus it is dependent neither on genetic information of an individual nor on the representation. The mates selection strategy is the third parameter of evolutionary algorithms.

3.4.4 How offsprings are born

In contrast to mates selection, breeding a new offspring is genotype dependent. There are two genetic operations generating variety and both were already defined in section 3.1.1. With mutation in play the evolution process makes smaller steps, but never stops. On the other hand by applying crossover evolution can make bigger steps but once the population converges to the same genotype, crossover cease to produce novelty.

In the natural evolution *mutations* occur randomly. To represent this randomness in the algorithm, a probability for a mutation is set. As by definition 3.1.6, mutation alters genetic sequence in some parts. To represent the true nature of mutation a different probability should be set for each part of the sequence, as different parts are more or less likely to mutate. In the algorithm a simplification, of setting the same probability of mutation for the entire sequence, is applied. The probability does not have to stay the same and can also evolve. Setting up the right probability is once again crucial. Setting up high probability of mutations could resemble a random search method, but in combination with high selection pressure, premature convergence of the population could be prevented.

The *crossover* operation allows for good genetic material to be inherited by next generations. This encourages more intensive search in the space of better solutions. There are two common types of crossover operations. A *n-point crossover* distributes n points across sequences, splitting them into parts, which are numbered. The points have to be at the same positions for both sequences being crossed over. Otherwise, the length of the resulting sequences would differ. To form a new sequence odd parts from one parent are merged with even parts from the other parent and vice versa. The other approach is *uniform crossover* where the combination is basically a random selection of genetic information from either of two parents. There is also some probability involved in crossover operation determining whether the crossover is applied. If the crossover is not applied, then the offsprings are only clones of their parents.

So far it has not been discussed how does the actual genetic information look like. It is also very problem specific, but there are some common representations like binary, integer or real-valued sequences. It was mentioned earlier that both genetic operations depend on genotype and therefore on its representation. Design of these operations is another parameter of evolutionary algorithms.

3.4.5 Game of survival

Once offsprings are born, there is one last important step to be done. Individuals surviving into next generation have to be chosen. Again, there are many strategies of doing so and again, the survival selection strategy is one of the evolutionary algorithm parameters. These are some of the most common strategies:

- *direct replacement* - all offsprings replace their parents
- *random replacement* - individuals advancing to a new generation are selected at random from both parents and offsprings
- *fitness based replacement* - both parents and offsprings are selected to form a new generation based on their fitness (roulette, top rank, tournament)
- *steady state replacement* - only small number of offsprings is created and replace usually the least fit parents
- *elitism strategy* - keeps always the fittest parents in the population
- *crowding operators* - maintain the population dispersion, allowing new individuals to join the population, only by replacing the most similar individual

3.4.6 Terminating conditions

The natural evolution never stops and it is possible that also evolutionary algorithm has no end. But most of the evolutionary algorithm scenarios have an end. The algorithm is computing a fitness of each individual during its run. It is common to set a threshold for fitness. When an individual's fitness reaches the threshold, the algorithm stops and take that solution as the result. To be honest, it is likely that the threshold would never be reached. Lowering the threshold throughout the evolution is one solution, the other is to terminate the evolution after some number of generations, and taking the fittest individual as the result. *Fitness threshold* and *number of genrations* are the last two parameters of evolutionary algorithms, which need to be set up beforehand.

3.5 Evolutionary algorithm design process

- population initialization strategy
- population size
- fitness function
- mates selection strategy
- mutation operation
- probability of mutation
- crossover operation
- probability of crossover
- replacement strategy
- fitness threshold
- number of generations

In section 3.4 these parameters of evolutionary algorithm were discussed separately. However, it is possible that all parameters mentioned before affect each other. This knowledge has to be taken into account, when designing an algorithm. Generally, fitness-based selection is implemented either for mates selection or replacement selection, not both. When one starts to dig deeper into the details of the evolutionary algorithm, many other questions arise. How many individuals are chosen to become parents? How many offsprings should be created? These two are closely related with each other, as with the selection and replacement strategies. Regarding the probabilities of genetic operations, mutation is much less likely to happen (usually around 0,1) than crossover (usually from 0,6 to 0,9). If an algorithm implements both of these operations, crossover is usually applied first. Setting up a fitness threshold always depends on the fitness function itself, but also on the computational power or time one has.

Huge part of setting up and practicing evolutionary algorithms is experimenting. Comparing the results is fundamental to fine tune the parameters. There is never a perfect setup of the parameters. Adjusting the parameters is an ongoing process an evolution itself. More about this can be found in the experimental part of this work in sections 6.1 and 6.2.

3.6 Coevolutionary approach

The most of evolutionary algorithm applications work with a single population. As mentioned in section 3.1.3, in natural ecosystems more species, populations, interact and evolve together. Splitting individuals into more populations and evolving them simultanously is the so called *Coevolutionary approach*. Populations could be competing against each other to get more of the recources that the environment offers. A mutual cooperation is not out of the scope either, when each population could be dealing with a distinct part of a problem.

Tomassini et al.[12] divided the coevolutionary approaches into two categories:

- *Cooperative coevolution*, is a system in which a number of different species, populations cooperate in order to find partial solutions, which are then combined in some way to solve the global problem.
- Competitive coevolution, is a system in which the different species, populations prosper or decay at the expense of each other.

When is using this coevolutionary strategy appropriate or even beneficial? Chapter 14 in *How to Solve It: Modern Heuristics* [8] talks about using coevolution in the search for optimal strategies for playing games. When there is no easy way to measure performance beyond counting wins or loses. Defining a suitable fitness function is difficult or even impossible. All these symptoms exactly fit the case of the Liar's dice player. Therefore a competitive coevolution is implemented, rather than the straight evolution. More details can be found in section 5.3.3.

3.6.1 Multiple fitness functions

When dealing with a single population, there was only one fitness function employed at once. With the coevolutionary approach and more populations evolving simultaneously, a new dimension of experiments opens. Instead of having a common fitness function for all populations, each population can have a different fitness function. It is possible because an individual is compared only with individuals from his population. No inter-population comparisons occur. Having different fitness functions for each population better resembles the coevolution happening in nature. Experimenting with different fitness functions in solving the Liar's dice problem is described in section 6.2.2.

Chapter 4

Grammatical evolution

'In natural computing algorithms grammars are mainly utilized to construct syntactical structures.' 'This generative nature is useful with a developmental approach.' 'Some developmental algorithms take advantage of grammatical encoding.' 'In genetic programming, grammars take control of evolving executable structures.' These statements from *Natural Computing Algorithms* [2] suggest that using grammatical evolution in evolutionary design is not a waste of time.

4.1 Grammatical and developmental computing

This section introduces the other adaptation level, discussed in section 3.1.2, into the play. The process of ontogenesis inspires a field of natural computing called *developmental computing*. The ontogenesis also fits into the frame of basic evolution algorithm 1, where it is responsible for mapping genotype to its phenotype. One can look at the mapping process, as if genotype was holding some parameters, and development would be decoding those into a solution, a phenotype. The original process is usually pretty simple, but with grammars in hand, the algorithm gains power. *Grammatical computing* are methods powered by a generative engine, *a grammar*. These methods have one big advantage over the other approaches in natural computing. They are capable of, not only generating the appropriate parameters for a model, but also generating the model itself. Most importantly both grammatical and developmental computing tend to go hand-in-hand. While the development ensures a stable connection between a model and the environment forming so called *feedback loops*, grammars tend to produce parameters for the model, and are capable of altering the model itself, if needed. In sections 4.1.1 and 4.1.2 the generative engine and its representation are introduced, borrowing all definitions from [10].

4.1.1 Grammars

The basic component of grammatical computing algorithms are *grammars*. To represent grammars in this work a very common notation technique for grammars, the *Backus-Naur form*, is used. It contains two kinds of symbols:

- terminal symbols, *terminals*, denoting lexemes and
- nonterminal symbols, *nonterminals*, representing syntactic structures.

These symbols are formed into *productions* (p), with a nonterminal on the left hand side $lhs(p)$ and a sequence of terminals and nonterminals on right hand side $rhs(p)$. Nonterminals are usually words in pointy brackets, terminals are usually same as the lexeme they represent. The name productions indicates that something is produced. How? By replacing a left hand side of a production with the respective right hand side. The replacement is called a *derivation step*. By doing derivation steps, one performs a derivation. The derivation usually starts from a special start nonterminal symbol and it ends when there are no nonterminals left to replace. The sequence of terminals one gets at the end belongs to a set called *generated language*. So far the descriptions perfectly matches a description of *context-free grammars*, an equivalent to the Backus-Naur form. There are other types of grammars, but for this thesis the context-free ones will suffice the case. Formal definitions of context-free grammar 4.1.1, direct derivation 4.1.2, leftmost derivation 4.1.3 and generated language 4.1.4 follow.

Definition 4.1.1 (Context-Free Grammar)
A context-free grammar is a quadruple $G = (N, T, P, S)$, where

- N is a set of nonterminals
- T is a set of terminals, $N \cap T = \emptyset$
- P is a finite set of productions of the form $A \rightarrow x$, where $A \in N$, $x \in (N \cup T)^*$
- $S \in N$ is the start nonterminal

Definition 4.1.2 (Direct Derivation)
Let $G = (N, T, P, S)$ be a context-free grammar, $p \in P$, and $x, y \in (N \cup T)^*$. Then, $x\ lhs(p)\ y$ directly derives $x\ rhs(p)\ y$ according to p in G, denoted by

$$x\ lhs(p)\ y \Rightarrow x\ rhs(p)\ y \quad [p]$$

Definition 4.1.3 (Leftmost Derivation)
Let $G = (N, T, P, S)$ be a context-free grammar, $u \in T^*$, $v \in (N \cup T)^*$ and $p = A \rightarrow x \in P$. Then, uAv directly derives uxv in the leftmost way according to p in G, written as

$$uAv \Rightarrow_{lm} uxv[p]$$

Definition 4.1.4 (Language)
Let $G = (N, T, P, S)$ be a context-free grammar. If $S \Rightarrow^* w$ in G, then w is a word of G. A word w, such that $w \in T^*$ is a word generated by G. The language generated by G, $L(G)$, is the set of all words that G generates:

$$L(G) = \{w : w \in T^*, S \Rightarrow^* w\}$$

4.1.2 Parse tree

One might easily get disoriented in derivation steps, lose track of which symbols were derived from which. Therefore, it is a good habit to represent the derivation steps visually. For that scientists use parse trees. By constructing and later examining a parse tree, one can easily trace derivation steps. To formally define a parse tree a small subtrees called *production trees* are used.

Definition 4.1.5 (Production Tree)
Let $G = (N, T, P, S)$ be a context-free grammar, and $p \in P$. The production tree $pt(p)$, corresponding to p is a labeled elementary tree, such that $lhs(p)$ labels $root(pt)$ node and frontier nodes, $fr(pt(p))$ consist of $|rhs(p)|$ nodes labeled with the symbols appearing in $rhs(p)$ from left to right

Now, based on the definition above, one is able to construct a production tree for each production of a grammar. As a derivation is a sequence of applied productions, a *parse tree* is a set of production trees joint together.

Definition 4.1.6 (Parse Tree)
Let $G = (N, T, P, S)$ be a context-free grammar. A parse tree of G is a labeled tree t satisfying two conditions:

- $root(t)$ is labeled with a start symbol S
- each elementary subtree t' appearing in t represents the production tree $pt(p)$ corresponding to a production $p \in P$.

The following example 3 should demonstrate all new structures presented in last two sections 4.1.1 and 4.1.2 and resolve any potential misunderstandings.

Example 3
Consider a grammar with these two productions,

$$\begin{aligned} 1: \langle expression \rangle &\rightarrow \langle expression \rangle + \langle expression \rangle \\ 2: \langle expression \rangle &\rightarrow \langle term \rangle \end{aligned}$$

- the left hand side of a production, is denoted as: $lhs(1) = \langle expression \rangle$
- the right hand side of a production, is denoted as: $rhs(1) = \langle term \rangle + \langle expression \rangle$
- nonterminals in this example are: $\langle expression \rangle, \langle term \rangle$
- the only terminal is $+$

A production tree, representing production 1, is on figure 4.1 and a parse tree, representing the following derivation is on figure 4.2.

$$\begin{aligned} \langle expression \rangle &\Rightarrow \langle expression \rangle + \langle expression \rangle & [1] \\ &\Rightarrow \langle term \rangle + \langle expression \rangle & [2] \\ &\Rightarrow \langle term \rangle + \langle term \rangle & [2] \end{aligned}$$

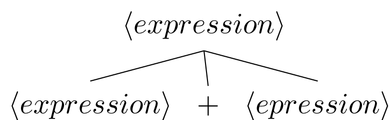

Figure 4.1: Production tree.

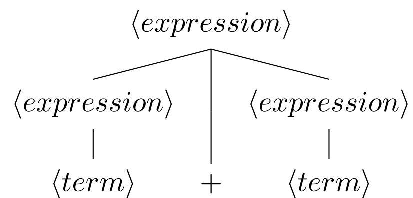

Figure 4.2: Parse tree.

4.2 Grammatical evolution algorithm

The algorithm for grammatical evolution is presented and described in this section. There is one last piece of the puzzle that needs to be explained in advance. Therefore section 4.2.1 reveals all secrets of this process.

4.2.1 Genotype mapping

Mapping a genome to a phenotype is the thing that is unique for grammatical evolution. This section explains how is the issue solved in this thesis. The process of mapping depends on representations of both genotype and phenotype. While the genotype is kept simple the phenotype can be as complex as one wishes. The advantage of having a simple genotype is in the simplicity of genetic operations. One would implement these rather over a sequence of bits, than over behavior trees. For the purpose of this work a sequence of integers was selected as the genotype representation. Any programming language can easily store and work with sequence of integers, operations with integers are simple and fast. Moreover the genotype is easy to read by humans.

How does the algorithm map a sequence of integers to phenotype? Grammar, a generative engine described in detail in section 4.1.1, is going to transform the integers of genotype, to a behavior tree, phenotype. Derivation starts with the starting symbol, as usual. Always the left most nonterminal symbol is expanded, applying the leftmost derivation. Each derivation step is controlled by integers from genotype. This assures that the derivation is deterministic and never ambiguous. The choice of which production should be applied, is computed with a simple equation: $p = c \ \% \ o$ where p is a production option to be used, c is the next integer value from the genome, % is the modulo operation and o represents the number of all production options. This simple evaluation, hand in hand with the respective derivation step, continues on, until there is no production to apply. The final phenotype is reached. One could ask, what if there was not enough integer values in a genome? A very simple solution of taking values once again from the beginning of the genome could be applied. It is called a circular genome.[1]

There is one specialty in the genotype mapping. A distinct nonterminal symbol, $\langle gene \rangle$, may be used in a grammar. When this symbol is to be expanded, the next integer value from the genome is stored directly in the behavior tree instead of the $\langle gene \rangle$ symbol. How this value is used during the decision making, depends on the application, but the number usually represents some kind of threshold.

The whole process of mapping has to be deterministic. So that reproducing the process on the same genotype would produce the same result. Thanks to the determinism of the process, mapping happens only once in a lifetime of an individual, in a similar way to the original process in nature.

[1] Circular genomes or DNA can be found in nature as well for example among bacteria, but also plasmids, mitochondrial DNA and chloroplast DNA form circular structures.

Algorithm 2: GRAMATICAL EVOLUTION [2]

```
1  grammar.init()
2  fitnessFunction.define()
3  parameters.set()
4  population.init()
5  population.develop()
6  population.evaluate()
7  repeat
8      parents = population.getParents()
9      offsprings = parents.breed()
10     offsprings.develop()
11     offsprings.evaluate()
12     population.update(offsprings)
13 until terminating condition
```

All steps of the algorithm were discussed before, but here is a short recapitulation. In the initialization phase a problem specific grammar is created. Fitness function is also problem specific and it must be defined beforehand. One is trying to keep it as simple as possible, because computing the fitness is usually the bottleneck of the algorithm. Other parameters of the algorithm were thoroughly described in section 3.4. Those are the parts of the algorithm requiring experimenting. Population is then initialized, perhaps randomly at the first experiments, but with some gathered knowledge applied latter in the day. The first development takes place even before the evolution starts. It was mentioned in section 4.2.1 as genotype to phenotype mapping. Once all individuals have their phenotype, the fitness can be computed. This is the step where knowledge about behavior trees, from section 2.3.1, comes in. Evaluating an individual means applying its phenotype, a behavior tree, and solving the problem with it. Quality of the solution is computed as an individuals fitness. Inside of the evolutionary loop there are only operations already described in the section 3.3 or here.

Chapter 5

Implementing gramatical evolution

After reading this chapter, the reader should be able to reproduce the solution and implement all described algorithms by him or herself. The following paragraphs elaborate on the most intimate details of the implementation. Nothing is left for later explanation and all of potential questions should be answered. If not do not hesitate to contact the author of this thesis. Section 5.1 speaks about common features of the algorithm, those that are the same for both problems. The problem specific parts are discussed in sections 5.2 and 5.3.

5.1 Implementation in general

As mentioned in section 2.3, both problems have some similarities and the same mechanism is used to solve them. Having the theoretical algorithm all prepared and ready is one thing, but when it comes to writing the code, many hidden unanswered questions arise. The following sections 5.1.1, 5.1.2, 5.1.3, 5.1.4, 5.1.5 and 5.1.6 speak about these little issues. How were the questions answered and what effect did the answers have on the final solution.

5.1.1 Programming language

The choice of the implementation language is a bit controversial, but this section tries to reason for the choice. A mediocre computer science engineer would suggest the use of compiled languages over the interpreted ones. He would reason that the compiled languages produce small optimized binaries, they have tight memory control, so they are fast. It is a perfectly acceptable and valid reason to choose a compiled language like C++ or Java. In spite of this knowledge Python was chosen, for many different reasons:

- The implementation is smooth with fast turnaround time[1] and easy debugging.
- The source code is easy to read and manage.
- Genetic operations (selection, mutation and crossover), which are the core of the evolutionary approach, do not take much time to compute in whichever programming language. The bottleneck of the algorithm is fitness evaluation.
- The code is constantly changing during the development of the algorithm.

[1] Period for completing a process cycle, commonly expressed as an average of previous such periods. (http://www.businessdictionary.com)

- To show that even a popular scripting language can successfully apply evolutionary approaches.
- The game simulator used for evaluating a player of Liar's dice was written in Python.

If one wanted to use the algorithm in production, coding it in a compiled language would be necessary.

5.1.2 Initial grammar design

The grammar has to be designed, so that it would generate a behavior tree. There has to be a root, sequence and selector nodes, and of course condition and action nodes. Productions of the initial grammar were constructed to allow for nested behavior trees. No restriction regarding the order of action and condition nodes in selectors or sequences were made. These common productions were used in the first run of the experiments:

$$\langle start \rangle \rightarrow \text{sequence } \langle behaviorTree \rangle \mid \text{selector } \langle behaviorTree \rangle$$

$$\langle behaviorTree \rangle \rightarrow \langle behaviorTree \rangle \ \langle node \rangle \mid \langle node \rangle$$

$$\langle node \rangle \rightarrow \langle condition \rangle \mid \langle action \rangle \mid \langle start \rangle$$

The productions with $\langle condition \rangle$ and $\langle action \rangle$ on its left hand side are problem specific, and therefore, the implementations are presented separately in the respective sections 5.2.1 and 5.3.1. The use of circular genome, explained in section 4.2.1, opened a possibility for constructing an infinite behavior tree. Obviously it is not possible to work with an infinite behavior tree on a finite hardware, so a limitation had to be made. The number of generated subtrees was restricted. After a certain amount of usages of the production $\langle node \rangle \rightarrow \langle start \rangle$, the production is removed from the grammar. This forbids generating more subtrees than the parameterized threshold.

Figure 5.1 shows an example of a parse tree generated by the common productions presented in this section. Although this tree does not resemble the tree from figure 2.2, it is a valid behavior tree and it was used according to the outline from section 2.3.1.

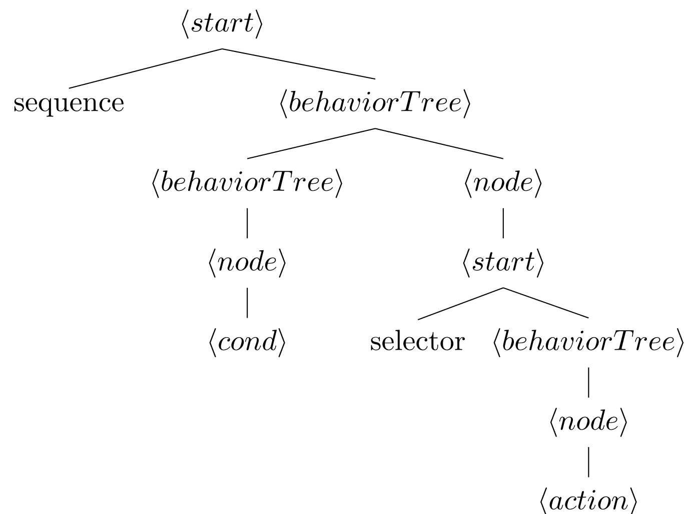

Figure 5.1: Behavior tree generated by the initial common productions.

5.1.3 Behavior tree activaton

After a tree was built, it can be used to solve the targeted problem. A decision making process starts here. The root node of the tree is activated and its code is executed, as does section 2.3.1 suggest. Now it is the right time to explain, what the code actually contains.

To traverse a behavior tree recursive descent is implemented. When a composite node activates one of its children, code of the activated node is executed, while the original execution is paused. This is where the recursive behavior transpires. Recursion is an elegant solution, offloading the complicated context switching tasks to a compiler or an interpreter and keeping the code clean. However, it is a quid pro quo for performance.

A bright reader could object here, that the behavior tree from section 5.1.2 had some extra nodes that were not described in section 2.3.1, nor was their code explained here. The objection would be valid, but the code of *start*, *behaviorTree* and *node* will not be mentioned even here. The reason is found in the following paragraph.

As the first round of experiments was run, using the initial grammar, the results were not very satisfying. It took too many generations and too much processor time for evolution to produce some reasonable, not necessarily good, solutions. The first experiments are not mentioned in more detail either, as they produced poor results. A review of the algorithm followed and decision to simplify the structure of behavior tree was made.

As figure 5.2 shows, the initial behavior tree presented on figure 5.1 can be reduced to a very simple behavior tree with only four nodes. A three-fold reduction was made in this case, as the original tree had twelve nodes. Less nodes mean less recursion, less memory and better performance. The number of nodes was not the only problem of the initial design. The grammar allowed for creation of trees that had some dead nodes. These nodes could have never been activated or their activation was of no use. For example a condition node after an action node is of no use and a tree without any action nodes likewise. Creating such trees made no sense in the first place.

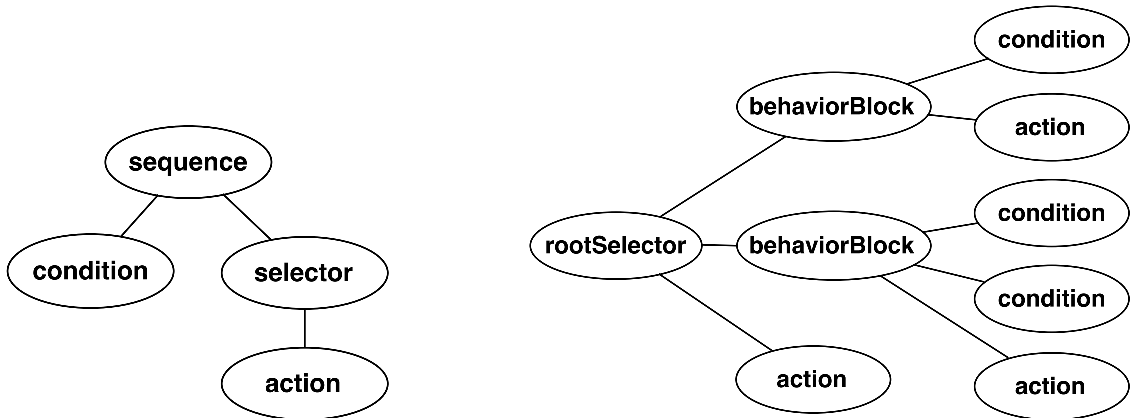

Figure 5.2: Simplified behavior tree from figure 5.1.

Figure 5.3: Behavior tree generated by enhanced method.

There were two possible ways to overcome the revealed obstacle. Implementing an algorithm that would simplify the parse tree generated by the grammar, by removing the unnecessary nodes, was the longer way around. It would not be a trivial algorithm, but more importantly, additional computation would be needed, decreasing the overall performance. The second and shorter way was removing the obstacle completely and creating a new enhanced grammar. Constructing a simplified behavior tree right away saved some

execution time and increased performance. Figure 5.3 shows an example of a behavior tree generated by enhanced method from section 5.1.4.

5.1.4 Generating simplified behavior trees

A new grammar was designed to generate more appealing behavior trees. The new design is inspired by the work of Nicolau et al.[7]. At first new productions are presented and then a description of each symbol and the respective implementation follows.

$$\langle rootSelector\rangle \rightarrow \langle behaviorBlock\rangle\langle rootSelector\rangle|\langle actionSequence\rangle$$

$$\langle behaviorBlock\rangle \rightarrow \langle condition\rangle\langle behaviorBlock\rangle|\langle condition\rangle\langle actionSequence\rangle$$

$$\langle actionSequence\rangle \rightarrow \langle action\rangle\langle actionSequence\rangle|\langle action\rangle$$

The logic behind the new grammar goes like this. The root node is a selector node (*rootSelector*) with zero or more sequence nodes (*behaviorBlock*) as its children. The last and the only mandatory child of the root node is also a sequence node (*actionSequence*). A *behaviorBlock* consists of one or more condition nodes followed by an *actionSequence*. There is no mystery behind the *actionSequence*, as it consists of one or more action nodes. A decision making based on the new type of behavior tree is pretty straightforward. The leftmost *behaviorBlock* is activated and the conditions associated with the block are tested. If all of the conditions are satisfied, the actions following are activated, one by one. Otherwise the conditions of the following block are tested. In case all of the blocks fail to satisfy their conditions, the default actions are activated. For the convenience of the implementation the default actions are also covered by a *behaviorBlock*. The default sequence solves the problem of individuals failing to do any move, as a result of not meeting any condition combinations.

One extra trick is implemented in addition to the new grammar. In case there are two selector nodes in the tree, and one is a child of the other, a simplification can be made. The simplification comes directly from the theory of behavior trees presented in section 2.3.1. The definition says that a selector node activates its children one by one, always in the same order, until one of the children succeeds. In that case the selector node itself succeeds, otherwise it fails. So without breaking the rules of this definition a hierarchy of nested selectors can be transformed to a single selector. It does not matter on which level of the nested selectors a node succeeds. Once it does, its parent does the same and the success is escalated to the topmost level. The same works with sequence nodes, so a hierarchy of nested sequence nodes can be transformed to a single sequence. As soon as only one node in the sequence hierarchy fails, the failure is escalated to the topmost sequence, which fails as well. These two properties are used, as the so called trick, to simplify each behavior tree. Each nested selector or sequence node is attached to its parent only temporarily. When the temporary node is expanded, its children are attached to the tree at the exact spot, where the temporary node was residing. An example to demonstrate the trick follows.

Example 4

Let $g = [14, 5, 2, 47, ...]$ be a genome of an individual, who is about to generate its behavior tree, using the productions from section 5.1.4. Derivation starts with the starting symbol and that is $\langle rootSelector\rangle$. Using the equation from section 4.2.1 ($p = 14 \ \% \ 2 = 0$)[2], the first alternative of the production is applied. Figure 5.4 captures the state of the behavior tree after the first expansion. The fourth derivation step expands the other $\langle rootSelector\rangle$

node. Here comes the trick. The expanded node is removed and all of its children are attached to its parent. The state of the tree before the expansion is captured on figure 5.6, the state after the trick is shown on figure 5.7. In fact the trick was applied also in step three, where the $\langle actionSequence\rangle$ node was replaced by the $\langle action\rangle$ node. In all four figures shortcuts for the nonterminals are used in the following manner: $RS = \langle rootSelector\rangle$, $BB = \langle behaviorBlock\rangle$, $AS = \langle actionSequence\rangle$, $C = \langle condition\rangle$ and $A = \langle action\rangle$.

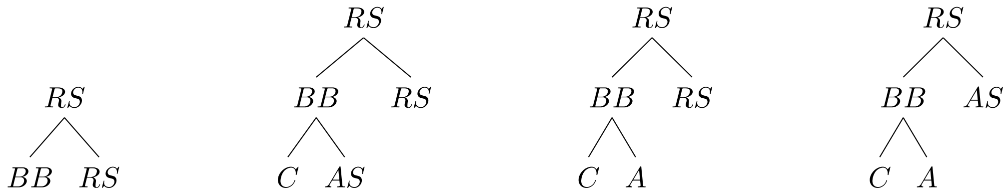

Figure 5.4: Step 1. Figure 5.5: Step 2. Figure 5.6: Step 3. Figure 5.7: Step 4.

5.1.5 Tree structure

To store a behavior tree the program leverages a `Node` class. Each `Node` has three attributes: `data` used for a type of a node, `children` containing list of child nodes, `size` as the total number of nodes in the sub-tree. Constructing a behavior tree, means adding `Node` objects to the lists of children of the *rootSelector* `Node` object.

5.1.6 Common evolution parameters

Some parameters of evolution are common for both algorithms. At the beginning of the evolution the population is initialized randomly. It is done in hope for a well dispersed population, but there is no guarantee. The size of the population is specified by parameter `POPULATION_SIZE`. Only positive multiples of 4 are valid population sizes. Each individual is initialized with a genome, consisting of random integer values. The maximum value is set by `MAX_GENE_VALUE` parameter and the number of values is set by `GENOME_LENGTH` parameter. Each individual is given a random name, to have a friendly reference to him or her. For generating the random names the program uses a python package `names`[3]. The range of values for both fitness functions is: $\langle 0, 1\rangle$. After each new generation is formed, a new fitness value is computed for each individual. The reason is the ever changing environment that the individuals interact with.

When it comes to creating a new generation, the number of individuals selected for mating is set by `MATES_NUMBER` parameter. The parents are always selected in pairs and an individual can mate only once per generation. A crossover operation is continually applied to both provided genomes and two new individuals are born. The operation splits both genomes at random but same place and combines these parts to form new genomes. After the single point crossover a mutation operation is applied to both genomes. The traditional approach of specifying some probability for mutation is not used. The operation uses two parameters which can be specified for each evolution: `MUTATION_SIZE` tells how many genes are mutated and `MUTATION_INTENSITY` is the maximum value to be added to the mutated gene. The locations of mutation are chosen randomly. Also the value added to the mutated

[2] Result of the modulo operation is zero and the list of production alternatives is also indexed from zero. That means the first alternative (on index zero) of the production is to be applied.

[3] the package was obtained from: https://pypi.python.org/pypi/names/

gene is chosen randomly from 0 up to the maximum value specified by the parameter. This implementation ensures that the evolution does not end, because there is always some change in the genome, thanks to mutation.

After a certain number of generations the evolution is terminated. The number of generations is one of the parameters of the evolution run. More on setting a proper value, could be found in section 6.1. All the other parameters that were not mentioned are problem specific and will be described for each problem separately in sections 5.2.5 and 5.3.3.

Although there are some common parts of the algorithm two separate scripts are used to solve each of the problems. There are no configuration files, no parameters, no arguments, no environment variables. Every parameter is set directly at the beginning of each script. Python `logging` module is used to manage the data flowing from the evolution. Methods for formatting the data, like behavior trees, board status, rolled dice, genomes, fitness values and many other, are implemented to help with experimenting.

5.2 Specifics of knight's tour

To solve the problem of the knight's tour as specified in section 2.1, the behavior tree gives decisions on where to move the knight next on the board. A specific grammar is used for the generation of a behavior tree. A mechanism which takes a behavior tree and simulates a knight's tour on the board is part of the algorithm. To be able to compare two individuals a pretty simple fitness function is used. Parameters of evolution specific to the knight's tour are discussed as well as some additional features included to help with experimenting.

5.2.1 Knight's grammar

The following grammar was implemented in the grammatical evolution of the knight's tour problem. $G_k = (N_k, T_k, P_k, S_k)$, where:

$N_k = \{\langle rootSelector\rangle\ ,\ \langle behaviorBlock\rangle\ ,\ \langle actionSequence\rangle\ ,\ \langle condition\rangle\ ,\ \langle action\rangle\ ,\ \langle op\rangle\ ,\ \langle gene\rangle\}$

$T_k = \{$ possibleMoves, movesLeft, inCorner, onEdge, $=, >, <, \geq, \leq$, NW, NE, EN, ES, SE, SW, WS, WN $\}$

$S_k = \langle rootSelector\rangle$

$P_k =$

$\langle rootSelector\rangle \rightarrow \langle behaviorBlock\rangle\ \langle rootSelector\rangle \mid \langle actionSequence\rangle$

$\langle behaviorBlock\rangle \rightarrow \langle condition\rangle\ \langle behaviorBlock\rangle \mid \langle condition\rangle\ \langle actionSequence\rangle$

$\langle actionSequence\rangle \rightarrow \langle action\rangle\ \langle actionSequence\rangle \mid \langle action\rangle$

$\langle condition\rangle \rightarrow$ possibleMoves $\langle op\rangle\ \langle gene\rangle$ | squaresLeft $\langle op\rangle\ \langle gene\rangle$ | inCorner $\langle gene\rangle$ | onEdge $\langle gene\rangle$

$\langle op\rangle \rightarrow\ = \mid > \mid < \mid \geq \mid \leq$

$\langle action\rangle \rightarrow$ NW | NE | EN | ES | SE | SW | WS | WN

5.2.2 Knight's code

What happens when a selector or a sequence node is activated was already described in section 2.3.1. Upon activation of a condition or an action node an actual condition is evaluated or an actual action is taken. What each specific condition tests or what each specific action does is explained in the following sections.

Conditions

`possibleMoves` - Compares the number of moves knight can make from its current position against a threshold, using an operator. The operator is one of the parameters of the condition. The threshold is computed using the second parameter *gene*[4]as so: $threshold = gene \ \% \ 8$.

`squaresLeft` - Compares the number of squares that were not yet visited by the knight against a threshold, using an operator. The operator is one of the parameters of the condition. The threshold is computed using the second parameter *gene* as so: $threshold = gene \ \% \ total_squares_on_board$.

`inCorner` - Checks, whether the knight is standing in a corner further specified by a parameter. Each of the corners is marked with a number starting from the north-western corner and continuing clockwise. To get the number of the corner the following equation is used: $corner = gene \ \% \ 4$.

`onEdge` - Checks, whether the knight is standing next to an edge of the board further specified by a parameter. Each of the edges is marked with a number starting from the northern edge and continuing clockwise. To get the number of the corner the following equation is used: $edge = gene \ \% \ 4$.

Actions

There are eight possible actions a knight can make: NW, NE, EN, ES, SE, SW, WS and WN. Each of them tries to move the knight to a new position on the board. The letters stand for a direction of the move: N - north, E - east, S - south and W - west. The knight moves two squares in the direction of the first letter and one square in the direction of the second letter. For example the directive ES moves the knight two squares east and one square south.

5.2.3 Knight's tour simulation

This section speaks about the evaluation of an individual. There is one class (`KnightsTour`) responsible for simulating knight's movement on the board. For each tour a new board is initialized and the knight is placed on a random starting position. Then the first round of moves begins. The knight's behavior tree is activated and generates a sequence of moves. The simulator tries to move the knight according to the sequence. If one of the moves was to break the rules, the round is over and the rest of the moves is thrown away. However, if the knight failed to move at least once during the round, the entire tour is over. Moving knight out of the board or to a square that has already been visited is considered breaking the rules.

[4]explained in section 4.2.1

5.2.4 Knight's fitness

Computing fitness of an individual is pretty straightforward. An individual is sent on a single knight's tour several times, each time starting from a random position on the board. The number of visited squares on the tour divided by the total number of squares of the board is considered a partial fitness of an individual. The total fitness of an individual is the arithmetic mean of all his partial fitness values. The fitness of a population is the arithmetic mean of each individual's fitness. In addition to that, the highest fitness value from all individuals is recorded for each generation.

5.2.5 Knight's evolution

The most of the evolution process has already been explained, the rest is found here. A selection strategy for parents is fitness based. The individuals are sorted by their fitness and the fittest individuals form mating pairs. As mentioned in section 3.5 only one of the selection should be fitness-based, so the replacement was to be set at random. All of the new individuals would join the population, replacing the old ones selected at random. First experiments proved that the design was not right and the population did not improve. Sometimes good solutions were replaced by poor ones and the valuable information had been lost. An improved strategy was implemented. Still all of newborns were to replace the old ones selected at random, but only if the new ones were fitter. This updated increases the selection pressure, but it is on demand. Because knight always starts its tour from a random square, the fitness of each individual is recomputed for each generation.

Once the evolution is run several times, some results are produced. The enhancement called *Adam* uses results from previous evolution runs to start a new evolution. If an Adam is provided, he is inserted to the otherwise random initial population.

5.3 Specifics of Liar's dice player

Sections 3.1.3 and 3.6 introduced a very interesting concept of coevolution, which is implemented to solve the problem of Liar's dice. The most significant change compared to the straight, single population, evolution is in computation of fitness values. Otherwise both algorithms are more or less the same.

5.3.1 Liar's grammar

The common productions from section 5.1.4 are simplified a bit for performance reasons. The game is played in turns and each turn can be represented as a single action in the behavior tree. A single activation of the behavior tree should then produce only single action.[5]So there is no need for the *actionSequence* symbol in the grammar. It is replaced by a simple *action* and the third production, with *actionSequence* on its left hand side, is dropped completely. However, there is one special compound action hidden in the grammar. It is the move when a player reveals some of his or her dice and than makes a new bid. The are two stages of this move, the first is revealing some of player's dice. The second stage consists of announcing a new bid after re-rolling the remaining hidden dice. This compound action activates the behavior tree twice. First time from the root, second time from the spot where the first activation ended.

[5]The original design was generating sequences of actions. It would be possible to pre-generate moves for more turns and activate the tree only when there were no actions left. However the environment and the

For generating behavior trees, which could play Liar's dice, the following grammar was implemented. $G_l = (N_l, T_l, P_l, S_l)$, where:

$N_l = \{\langle rootSelector \rangle$, $\langle behaviorBlock \rangle$, $\langle actionSequence \rangle$, $\langle condition \rangle$, $\langle action \rangle$, $\langle reveal \rangle$, $\langle bid \rangle$, $\langle gene \rangle\}$

$T_l = \{$ firstToPlay, firstRound, haveHiddenDice, valueMatches, probableBid, impossibleBid, wonLastRound, lostLastRound, haveWilds, goodHand, bidderLowOnDice, lowOnDice, challenge, revealBidden, revealAny, revealWilds, revealAll, simpleRaise, raiseMore, newBid, bidWilds $\}$

$S_l = \langle rootSelector \rangle$

$P_l =$

$\langle rootSelector \rangle \rightarrow \langle behaviorBlock \rangle \ \langle rootSelector \rangle \mid \langle action \rangle$

$\langle behaviorBlock \rangle \rightarrow \langle condition \rangle \ \langle behaviorBlock \rangle \mid \langle condition \rangle \ \langle action \rangle$

$\langle condition \rangle \rightarrow$ firstToPlay | firstRound | haveHiddenDice | valueMatches | probableBid | impossibleBid | wonLastRound | lostLastRound | haveWilds $\langle gene \rangle$ | goodHand $\langle gene \rangle$ | bidderLowOnDice $\langle gene \rangle$ | lowOnDiceinCorner $\langle gene \rangle$

$\langle action \rangle \rightarrow$ challenge | $\langle reveal \rangle \ \langle bid \rangle \mid \langle bid \rangle$

$\langle reveal \rangle \rightarrow$ revealBidden | revealAny | revealWilds | revealAll

$\langle bid \rangle \rightarrow$ simpleRaise | raiseMore | newBid | bidWilds

5.3.2 Liar's code

Designing conditions, which examine the environment well, and actions, which bring the player closer to victory, is one of the key parts of the solution. A good knowledge of the rules of the game as well as some experience from playing the game is required. The following conditions and actions were crafted, in order to generate somewhat clever and competitive players. Adding new conditions or actions to the model is quite simple and the code structure is ready for that. Removing already implemented actions or conditions is also very trivial.

counting probability

Some of the conditions and actions use probability to make decisions. An auxiliary function is implemented to help with the computation. It has one argument, the value of interest, and its name is `get_probable_count(value)`. It computes minimum number of dice matching the value of interest that is most likely rolled on the table. If the argument in not the wild value, also the probable count of the wild valued dice is added to the final count. First the player's hidden dice are examined and the number of dice matching the target values is added. Then all the revealed dice is examined in the same manner. The last numbers added to the count is one sixth of the total number of all hidden dice of the opponents, and another sixth if the targeted value is not wild. An example should clarify how does the function compute the result.

state of the game is changing with each turn. Therefore activating the tree and examining the state of the game at each turn is more clever solution.

Example 5

The situation on the table is like this and the wild value is one:

Martin's hidden dice: 4 5 1

Martin's revealed dice: 5

Helen's hidden dice: 2 3 3 1

Helen's revealed dice: 1 5

Andrew's hidden dice: 4 4 1 6 3

Andrew's revealed dice:

Martin calls the function `get_probable_count`(5) and the counting starts. $c = 1$ (for the one 5 Martin has hidden) +1 (for the one wild value Martin has hidden) +1 (for the one 5 Martin has revealed) +1 (for the one 5 Helen has revealed) +1 (for the one wild Helen has revealed) +1 (as the one sixth of all hidden dice) +1 (as another one sixth of all hidden dice because the target value is not wild) = 7. In fact this 'guess' is quite accurate as the actual count of fives + wilds on the table is 7.

conditions

`firstToPlay` - Checks if the player is to start the round.

`firstRound` - Checks if the current round is the first round of the game.

`haveHiddenDice` - Checks if the player has some hidden dice left.

`valueMatches` - Checks whether the value of the current bid is the most frequent dice among the player's hidden dice.

`probableBid` - Compares the result of the `get_probable_count()` function passing the current bid value as the argument with the current bid count. The following comaprison is made: $computed_count \geq current_bid_count$

`impossibleBid` - Checks if the current bid count is less or equal to the all dice on the table.

`wonLastRound` - Checks whether the player has won the last round.

`lostLastRound` - Checks whether the player has lost the last round.

`haveWilds` - Checks if the player has more or equal number of wild valued dice among his hidden dice than a threshold. The threshold is computed from the gene parameter as so: $threshold = gene$ % `INITIAL_DICE`[6]

`goodHand` - Counts the number of the most frequent dice among the player's hidden dice and comperes the proportion to all of his hidden dice with a threshold. The threshold is computed from the gene parameter as so: $threshold = gene$ % 100

[6] `INITIAL_DICE` - the number of dice each player had at the beginning of the game

`bidderLowOnDice` - Checks if the player who called the last bid has less hidden dice than a threshold. The threshold is computed from the gene parameter as so: $threshold = (\ gene\ \%\ $ `LOW_ON_DICE_MAX`[7] $) + 1$

`lowOnDice` - Checks if the player has less hidden dice than a threshold. The threshold is computed from the gene parameter as so:
$threshold = (\ gene\ \%\ $ `LOW_ON_DICE_MAX` $) + 1$

actions

`revealBidden` - Reveals all the hidden dice that match the value of the last bid.

`revealAny` - Reveals one random dice.

`revealWilds` - Reveals all the hidden dice of the wild value.

`revealAll` - Reveals all the hidden dice that match the value of the last bid as well as all the hidden dice of the wild value.

`simpleRaise` - Bids the same value as the current bid and only increases the count by one.

`raiseMore` - Bids the same value as the current bid. The count is computed by the `get_probable_count()` function passing the current bid value as the argument. If the computed count is smaller or equal to the current bid count, the current bid count increased by two is bidden. Otherwise the computed count is bidden.

`newBid` - Changes the value of the current bid. The new value is the most frequent value among the players hidden dice. However, if the most frequent value is the same as the current bid value, the second most frequent dice among the players dice is bidden. As for the count it is computed by the `get_probable_count()` function passing the new value as the argument. If the computed count is smaller or equal to the current bid count, the current bid count increased by one is bidden. Otherwise the computed count is bidden.

`bidWilds` - Bids the wild value. The count is computed by the `get_probable_count()` function passing the wild value as the argument. If the computed count is smaller or equal to the current bid count, the current bid count increased by one is bidden. Otherwise the computed count is bidden.

[7] `LOW_ON_DICE_MAX` - maximum value for the threshold of testing if a player has low number of dice

5.3.3 Liar's evolution

Algorithm 3: Gramatical Competitive Coevolution

```
1  grammar.init()
2  fitnessFunctions.define()
3  parameters.set()
4  foreach p ∈ populations do
5      p.init()
6      p.develop()
7  end foreach
8  populations.compete()
9  repeat
10     foreach p ∈ populations do
11         parents = p.getParents()
12         offsprings = parents.breed()
13         offsprings.develop()
14         p.update(offsprings)
15     end foreach
16     populations.compete()
17 until terminating condition
```

Similarly to the straight evolution algorithm (2), the new algorithm starts by creating a grammar. Then a set of fitness functions is defined and all of the evolutionary parameters are set. Each population is initialized, with the number of individuals specified by the `POPULATION_SIZE` parameter. The initialization continues with genotypes being mapped to phenotypes and fitness functions being chosen for each population. A new operation *compete()* takes all individuals of one population and let them compete against all other individuals from the other populations. After all the competitions, fitness values of all individuals are computed. The evolutionary loop did not change much either. Each population advances to a next generation and then once again the round of competitions gets underway. The same evolutionary parameters are used for each population. Parents are selected at random, and the replacement strategy is fitness based. Individuals are sorted based on their fitness and the worst are replaced by newborns. The evolution cycle terminates after some number of generations. The number is specified by the parameter `GENERATIONS`.

5.3.4 Liar's competition

This section explains how do the two individuals compete against each other, during the evolution. The competition is actually a set of games of Liar's dice, that the individuals play against each other. Playing only one game does not say a lot about the quality of an individual, because the victor might just be lucky in that sole game. Good performance over multiple games is what makes individuals more valuable. The number of games per competition is specified by parameter `COMPETITION_GAMES`. Each game is played according to the rules specified in section 2.2. Both players take turns in activating their behavior trees and making the move their tree chooses. If a player makes an invalid move, he or she immediately looses the game. Along the game both players collect some data that is used

for computing their fitness. The list of information being collected with short description follows.

- `games_won` - number of games a player has won
- `dice_saved` - sum of the number of dice a player had at the end of each game played
- `challenger` - number of challenges a player made in all games played
- `challenger_won` - number of successful challenges made in all games played
- `challenged` - number of challenges called on player's bids in all games played
- `challenged_won` - number of unsuccessful challenges against a player in all games played
- `giveaways` - total number of dice a player gave to other players in all games played
- `takeaways` - total number of dice a player took from other players in all games played

5.3.5 Liar's fitness

To compute ones fitness, the data described in section 5.3.4 is used. The fitness is calculated, after all individuals ended all of their competitions and after all data has been collected. The fitness of a population is the arithmetic mean of fitness values of all individuals of the population. Four fitness functions were designed to measure the quality of individuals. In all functions the fitness of an individual is always relative to individuals of other populations. Descriptions of each fitness function follow.

Games won

This is the simplest function implemented. It counts the games an individual has won and divides the number by the total number of games he or she played. The goal is maximizing the number of victories of course.

$$f_1 = \frac{\texttt{games_won}}{\texttt{total_games}}$$

Dice saved

This function sums the number of dice that an individual had left at the end of each game he played. This number is divided by the total number of dice a player received at the start of each game.

$$f_2 = \frac{\texttt{dice_saved}}{\texttt{total_dice_recieved}}$$

This measure should capture player's qualities a bit better, as during the game each player is loosing dice. The player who looses the least number of dice is the winner of the game. The goal is once again maximizing the number of dice saved in each game.

Successful challenges

Another approach of evaluating individuals is to investigate their success in each round of the game. A round of Liar's dice ends when one of the players calls a challenge. To compute the fitness, this function counts all challenges made by a player, the number of successful ones, the number of challenges against a player and the number of unsuccessful ones. The goals are to maximize the successful challenges made and minimize the success of challenges against.

$$f_3 = \frac{\texttt{challenger_won}}{\texttt{challenger}} + \frac{\texttt{challenged_won} + \texttt{takeaways}}{\texttt{challenged}}$$

Dice transfers

The final and most sophisticated fitness function is similar to the the one counting challenges. The specialty of this method are `giveaways` and `takeaways`. In ice hokey, these statistics count the number of pucks a player either 'gave away' to an opponent or 'took away' from an opponent. In Liar's dice, it is about counting dice. A special situation occurs at the end of a round when the bid precisely matches the state on the table. Then the challenger gives one of his or her dice to the bid caller. This act of donating a dice to an opponent is very costly and this fitness function penalizes or rewards it. With the two parameters involved, `GIVEAWAY_FACTOR` and `TAKEAWAY_FACTOR`, one is able to set how much the donation of a dice affects the value of final fitness. Similarly to f_3, the goals are to maximize the successful challenges made and minimize the success of challenges against. On top of that, lays minimizing the number of `giveaways` and maximizing `takeaways`.

$$g = \frac{\texttt{challenger_won} - \texttt{giveaways} \times \texttt{GIVEAWAY_FACTOR}}{\texttt{challenger}}$$

$$h = \frac{\texttt{challenged} - \texttt{challenged_won} - \texttt{takeaways} \times \texttt{TAKEAWAY_FACTOR}}{\texttt{challenged}}$$

$$f_4 = g - h$$

Chapter 6

Experiments

This chapter details some interesting experiments that were conducted. For the following experiments these common parameters of evolution were set:

MAX_GENE_VALUE = 99

GENOME_LENGTH = 23

MUTATION_SIZE = 2

MUTATION_INTENSITY = 3

MATES_NUMBER = POPULATION_SIZE / 2

6.1 Solving knight's tour

In the early stages of investigations on the topic, this problem seemed much easier to solve than playing the Liar's dice. However, after some behavior trees were generated and knight's tours were attempted, it became apparent that solving this problem with behavior tree would be impossible. Therefore the initial goal slightly changed. Instead of looking for the perfect solution or abandoning the idea completely, new purpose of life was found for the problem.

Evaluating a tour is pretty simple and designing a fitness function was also straightforward. With this in hand, the concepts of the algorithm were to be tested on the knight's tour problem. Therefore, the potential experiments were focused on parameters of evolution.

6.1.1 Size of population

It is one of the questions that bothers each scientist: How to set up the constants or parameters? This experiment tries to figure out what is the best size of population for the grammatical evolution. Running the algorithm with different sizes of population and checking out which one gives the best results, looks as the way to go. However, trying this approach would most likely lead to a conclusion that larger population is always better. The reason is simple. The more individuals are in population the more genotypes and phenotypes are tested and there is a higher probability of finding a good solution. So that the experiments were objective, the number of individuals tested against the environment should be constant. To satisfy this condition, the number of generations was always adjusted according to the size of the population. GENERATIONS $\times$ POPULATION_SIZE = *const*

Five different scenarios were tested. In all of them the following parameters of the knight's tour were set:

`KNIGHTS_TOUR_RUNS` = 5

`BOARD_SIZE` = 5

The respective results are summarized by the following table.

POPULATION_SIZE	GENERATIONS	execution times	generation 40 average fitness	generation 40 best fitness	last generation average fitness	last generation best fitness
4^2	4^6	3m41.186s	0.102	0.2	0.1315	0.336
4^3	4^5	4m51.406s	0.1316	0.432	0.1355	0.32
4^4	4^4	4m29.982s	0.1039	0.32	0.1507	0.48
4^5	4^3	4m52.332s	0.1006	0.384	0.1237	0.456
4^6	4^2	4m49.182s			0.0862	0.512

Table 6.1: Knight's tour population size experiment

What is the best size of population then? Looking at the results, it is not easy to answer the question. Based on the best fitness at last generation, it looks as the more individuals in population the better. However, this sole value is not the result of evolution rather a result of a random search. Because of the specifics of the fitness function evaluation, found in section 5.2.4, the average value of fitness needs to be taken into account as well. In the second row of the table, one can see that the average fitness value has not improved much since the 40th generation. It was around the 40th generation in each case that the values of both best and average fitness stabilized and did not tend to improve. The execution times prove that the number of individual evaluations was kept the same throughout all runs. There is no clear victor and any population size produced more or less the same quality of individuals at the end. Choosing smaller population size is 'more evolutionary' approach, as the solutions are found through genetic operations and not by brute force of a random search.

When investigating the phenotypes of the best individuals, an interesting discovery was made. Behavior trees of best individuals of all evolution runs were not containing any condition nodes. The sole *behaviorBlock* was the *defaultActionSequence*. The evolution chose not to use the condition nodes and rather started building a sequence of moves, like the traditional methods solving the knight's tour problem. This experiment proved that using behavior trees to navigate a knight on his tour over the board is not a good idea.

6.2 Playing the Liar's dice

In the experiments the game of Liar's dice was restricted to only two players. So all the games played during the competitions were games of two players. The obvious reason is performance, the game of more players is longer, as more dice are in the game and more rounds are played. With having games of two players only, the logical consequence is having two populations competing in the coevolution.

6.2.1 Are populations improving?

The first experiment examines the course of coevolution on populations having the same fitness functions. Both populations used f_1, only counting victories. To be able to see more changes, the evolution was run for more generations than probably needed. To mitigate the larger number of generations the number of individuals in each population was kept small. The following parameters were set:

POPULATION_SIZE $= 8$

GENERATIONS $= 10000$

COMPETITION_GAMES $= 5$

LOW_ON_DICE_MAX $= 3$

This setup has been executed five times and the results were similar. The evolution of fitness of two runs is shown on figures 6.1 and 6.2. The charts are displaying the fitness of one population only because the sum of both fitness values in one generation is 1. The value 0.5 means that both populations are equally good.

Fitness values of each 10th generation were captured and from each ten values the arithmetic mean was computed and is shown on the chart. Both figures show that a population was able to gain the upper hand for a longer period. The charts also prove that the other population was able to recover and evolve a feature to swing the pendulum to its side. This behavior, of searching for ways to overcome the opposing side, is what the coevolution was designed to do. The fact that 'the pendulum swings'[1] proves that the populations are improving.

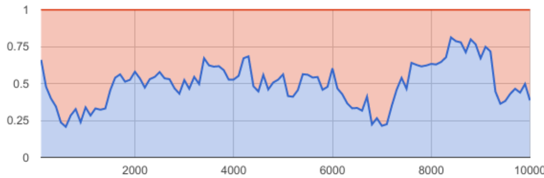

Figure 6.1: Liar's dice coevolution (f_1 1).

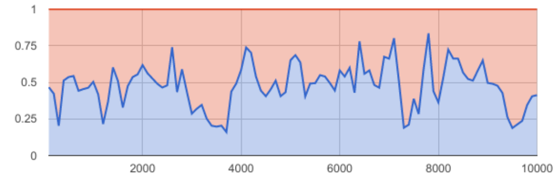

Figure 6.2: Liar's dice coevolution (f_1 2).

The same setup was used with changing both fitness functions to f_2, counting saved dice. One can see that the top line is not straight like when counting wins. The sum of fitness values is not necessarily 1, because both players could loose some dice in a single game. Some values are over 1, and that is more likely because of rounding errors. Most importantly the behavior of 'the swinging pendulum' is ever present, so the populations improved over generations.

6.2.2 Comparing fitness functions

The second experiment was about each of the populations having different fitness functions. In this scenario each population is pursuing a different goal and both populations should possibly develop different strategies. The following setup was used for the experiments. The evolution is going to be shorter but with more individuals involved. Both parameters were not chosen randomly but based on the Blondie24 experiment [6]. The setup was like this:

[1] the position of the leading population alternates between both populations

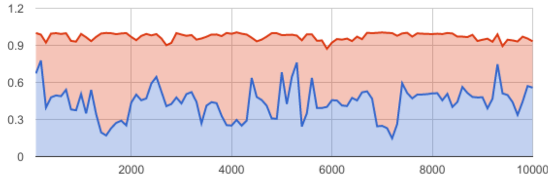

Figure 6.3: Liar's dice coevolution (f_2 1).

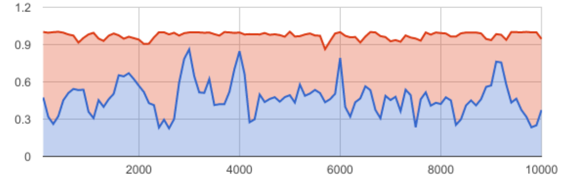

Figure 6.4: Liar's dice coevolution (f_2 2).

POPULATION_SIZE = 16

GENERATIONS = 800

COMPETITION_GAMES = 5

LOW_ON_DICE_MAX = 3

GIVEAWAY_FACTOR = 4

TAKEAWAY_FACTOR = 4

The first interesting comparison is the coevolution where one population uses f_1 and the other uses f_2. Both fitness functions look at the results of games. As figure 6.5 reveals, the evolution of fitness values is similar to the first experiment (6.2.1). At the beginning f_1 is better, but f_2 improves and between generations 200 and 300 has the upper hand. Then for some 150 generations the situation is calm before the storm. As f_2 finds some good individuals and tries to overcome f_1, the battle is on. After all f_2 seems victorious, because towards the end of the evolution it leads the competition. What conclusions can be made from this run? Once a again 'the swinging pendulum' is present, probably because both functions work on similar bases. However, it seems that over a longer period fitness function that counts saved dice tends to improve and gains an advantage.

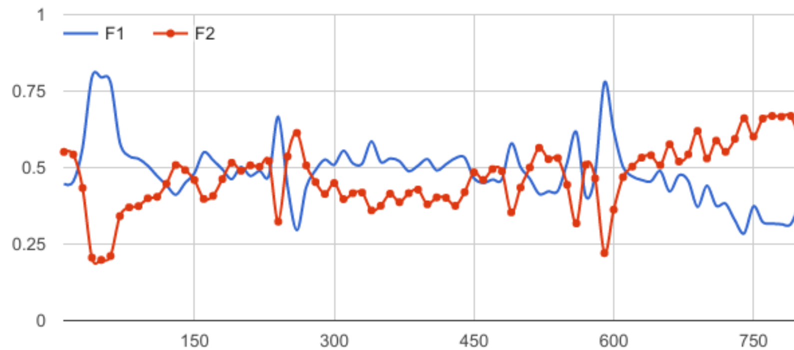

Figure 6.5: Liar's dice coevolution (f_1 f_2).

The second comparison is between the other two fitness functions f_3 and f_4. These two are also very similar, but the ranges of their values are different. On figure 6.6 both functions were shifted and rescaled to range $\langle 0, 1\rangle$. At first glance, one can see the difference from the previous figure. There is way more oscillation present. Why? Functions f_3 and f_4, both look at the results of each round of each game of an individual. Their measure of quality is based on more data, but if more is better in this case, is hard to tell. Other than

oscillation, one can see some alterations of the leading population. It look as though the f_4 is in the lead for more generations, but because of the scaling and shifting no conclusions are made.

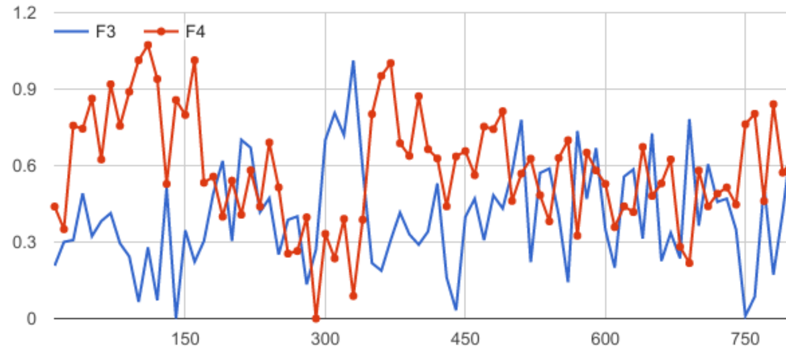

Figure 6.6: Liar's dice coevolution (f_3 f_4).

As the third comparison, functions measuring performance on a different level were chosen. There is a coevolution of populations graded by functions f_1 and f_3 on figure 6.7 and a coevolution of populations graded by functions f_2 and f_3 on figure 6.8. This time no scaling and shifting was applied to the fitness values. Therefore, each fitness will be examined separately. For the first time, it is easy to determine which population evolved into being supreme to the other, or is it? Looking at the fitness values of the f_1 and f_2 function respectively, most of the time, they are over 0.5. That is from the previous experiences considered as having the upper hand. However, there are two weird artifacts that could not be easily explained, like the sudden peak decrease of performance close to 600th generation in the first chart. Comparing figures 6.5 and 6.8 one can see that in both cases the population with function f_2 had very low score at the beginning, but towards the end, the tendency was to improve. Values of fitness of population with function f_3 are still oscillating a lot, but not decreasing globally. This can not be said about f_1 where the slight decline is obvious.

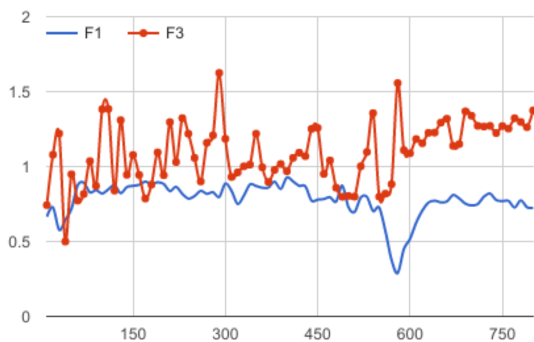

Figure 6.7: Liar's dice coevolution (f_1 f_3).

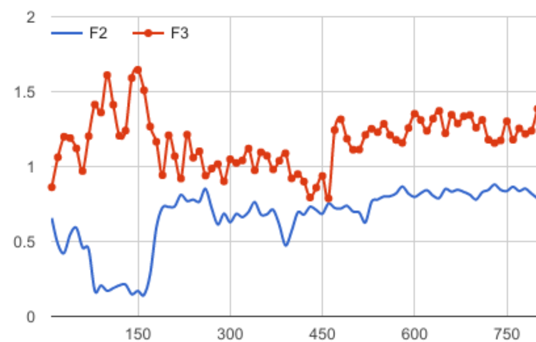

Figure 6.8: Liar's dice coevolution (f_2 f_3).

And what are the outcomes of the experiments? If one should choose from fitness functions that were presented, according to the experiments, the most promising is f_2. If one was to employ coevolution, using both f_2 and f_3 together is the recommended choice. The results of function f_4 were not bad at all, but tweaking the parameters a bit would be

necessary. Going for the simplest fitness function f_1 is not a bad choice because winning games is the thing that counts in the end.

Having different fitness functions for evaluating individuals is definitely beneficial in the search for better solutions. A good way to compare the functions is using the coevolutionary approach. Based on the gathered knowledge and observed properties, some new and more sophisticated functions can be designed.

Chapter 7

Conclusion

There are almost no boundaries in evolution and it is possible that the best methods of evolutionary design are yet to be discovered. This thesis aimed to show the use of already established methods on different domains of problems. An algorithm using grammatical evolution to produce behavior trees was designed, developed and implemented. The generated behavior trees were used to control individuals, solving problems. The algorithm was designed in a way that a grammar representing the problem is plugged in and the solutions are produced. The only other thing needed is a fitness function which quantifies the performance of an individual in the environment.

The first problem of knight's tour searched for a tree that would navigate the knight to complete his tour from any initial position. The experiments proved that the designed model was not able to solve the knight's tour completely. However, the population was improving and making longer and longer tours, with more generations passed. The evolution chose to build a single sequence of moves rather than a decision making structures of behavior trees. Perhaps the designed condition operations were not providing the right information from the environment. The experiment on choosing a proper population size showed that the results of evolution have to be examined closely and the interpretation is not easy.

The other problem of interest, playing Liar's dice, is considered the more interesting one. For the likes of Mario [7] or Checkers [8] the common strategies for playing the games are well known and many algorithms playing the games were already created. With the game of Liar's dice, the building of the algorithm started from the scratch. At first, the same approach of straight, single population evolution was implemented. However, it was hard to interpret the results and tell, whether the evolution was actually working. Simplifying the structure of behavior trees helped with the performance of the algorithm as well as with the debugging. After some discussions with the thesis supervisor, a coevolutionary approach was adopted. With two populations of individuals, one is able to determine which individuals are better. More importantly the experiments demonstrated that the evolution was working, as the populations were competing against each other to evolve fitter individuals. The coevolution also solves the problem of individuals learning to play against a certain type of players. Within a single population the individuals tend to follow a similar path of solutions, but there is no genetic dependency between two separate populations.

The main motivation for generating a player of Liar's dice, is a private programmers contest between friends.[1]The evolutionary developed player from this thesis is going to hit a real competition of different algorithms playing the Liar's dice. It is the best fitness measure that the presented algorithm could get. Before the contests begins, there is still some time to improve the current solution. Adding new conditions and possibly also actions is a way

to improve, other than fine tuning the parameters of evolution. The algorithm has to be generalized for playing games of multiple players. Using coevolution of more populations is an option here.

The evolutionary computing has already established itself as one of the big fields of AI. All the algorithms and results prove that there is a big potential. Similarly to the beginning, the thesis is ended with a quote by Turing.

„We can only see a short distance ahead, but we can see plenty there that needs to be done.“

Alan Turing, *Computing Machinery and Intelligence*, 1950 [14]

[1] Announcement about the contest: https://marekkukan.github.io/liars-dice/

Bibliography

[1] Bell, J. T.; Pai, A. A.; Pickrell, J. K.; et al.: DNA methylation patterns associate with genetic and gene expression variation in HapMap cell lines. *Genome Biology*. vol. 12, no. 1. 2011: page R10. ISSN 1474-760X.
Retrieved from: http://dx.doi.org/10.1186/gb-2011-12-1-r10

[2] Brabazon, A.; O'Neill, M.; McGarraghy, S.: *Natural Computing Algorithms*. Springer Publishing Company, Incorporated. first edition. 2015. ISBN 3662436302, 9783662436301.

[3] Chomsky, N.: *Syntactic Structures*. The Hague: Mouton and Co.. 1957.

[4] Darwin, C.: *On the origin of species*. New York :D. Appleton and Co.,. 1859. 470 pp.
Retrieved from: http://www.biodiversitylibrary.org/item/71804

[5] Depot, D. G.: Dice game rules: Liar's Dice.
Retrieved from: http://www.dicegamedepot.com/dice-n-games-blog/dice-game-rules-liars-dice

[6] Fogel, D. B.: *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann Publishers Inc.. 2002. ISBN 1-55860-783-8.

[7] Liebana, D. P.; Nicolau, M.; O'Neill, M.; et al.: Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution. In *EvoApplications*. 2011.
Retrieved from: https://pdfs.semanticscholar.org/af31/03a750ab88d4903bcfe264aaf7d74090f8ed.pdf?_ga=1.103289280.1794265660.1484083956

[8] Michalewicz, Z.; Fogel, D. B.: *How to Solve It: Modern Heuristics*. Springer. second edition. 2010. ISBN 9783642061349.
Retrieved from: http://www.springer.com/us/book/9783540224945

[9] Pereira, R.: An Introduction to Behavior Trees.
Retrieved from: http://blog.renatopp.com/2014/08/15/an-introduction-to-behavior-trees-part-3

[10] Repík, T.: Systems of Sequential Grammars Applied to Parsing. june 2014.

[11] Sanchez, E.; Mange, D.; Sipper, M.; et al.: *Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware*. Berlin, Heidelberg: Springer Berlin Heidelberg. 1997. ISBN 978-3-540-69204-1. pp. 33–54.
Retrieved from: http://dx.doi.org/10.1007/3-540-63173-9_37

[12] Tomassini, M.: *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time (Natural Computing Series)*. Springer. 2005 edition. 2005. ISBN 9783540241935.
Retrieved from: https://link.springer.com/book/10.1007/3-540-29938-6

[13] Turing, A. M.: On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*. vol. 2, no. 42. 1936: pp. 230–265.
Retrieved from: https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

[14] Turing, A. M.: Computing Machinery and Intelligence. 1950.
Retrieved from: http://cogprints.org/499/

Appendix A

List of attachements

Here a list of files that are available on the Compact Disc supplement to this thesis.

`GE-KT.py` - script dealing with the problem of knight's tour

`GE-LD.py` - script dealing with the problem of Liar's dice

`kt/` - folder containing some results of experiments presented in section 6.1

`e1-8-2-5t-5b.csv` - a file from the folder `kt` carrying results of one evolution run

`ld/` - folder containing some results of experiments presented in section 6.2

`e2-10k-8i-5g-f2-f1-1.csv` - a file from the folder `ld` carrying results of one evolution run

`thesis/` - folder containing the source files for generating the text of the thesis

`names-0.3.0.tar.gz` - a python module used for generating random names

The naming conventions of the `.csv` files is explained:

e1 - experimnet number

8-2 - using 4^8 evaluations of individuals and 4^2 as the population size

5t - 5 knight tours

5b - board size is 5x5

10k - 10000 generations

8i - 8 individuals per population

5g - 5 games per competition

f2 - first population uses f_2 to evaluate its individuals

f1 - second population uses f_1 to evaluate its individuals

1 - first run of the setup