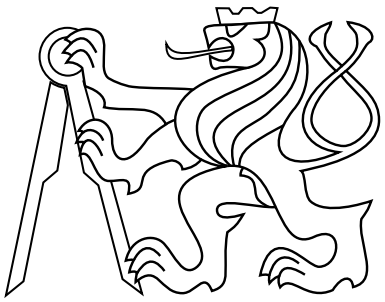
Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Control Engineering



Bachelor thesis

Evolutionary methods for mobile robot collision avoidance

Author: Dina Sushkova

Advisor: Ing. Karel Košnar, Ph.D.

May 2014

Acknowledgement

I would like to thank my supervisor Ing. Karel Košnar, Ph.D. for his help and valuable advises during the work on this thesis. I would also like to thank my family for the great support during my studies on CTU FEE and through my life.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

Prague 23.05.2014 .....

Abstract

Evoluční algoritmy se osvědčily při řešení mnoha problémů z různých oblastí. Jejich použití v úlohách mobilní robotiky může být přínosné jak pro nalezení nových způsobů řešení těchto úloh, tak i pro rozšíření míry autonomie mobilních robotů. V této práci je rozebráno použití genetického programování pro řešení takových úloh, jako je dosažení cíle a vyhýbání překážkám pro mobilního robota vybaveného laserovým senzorem vzdálenosti a informací o poloze cíle. Pro řešení zmíněných úloh byla definována potřebná gramatika kontrolérů. Genetické programování bylo použito k vyvíjeni kontrolérů dvou typů: structure-free a structure-restricted. Testování a zhodnocení výsledných kontrolérů bylo provedeno v simulačním prostředí. Práce popisuje použité fitness funkce a kontroléry pomocí nich nalezené. Ukázalo se, že pomocí genetického programování je možné nalézt kontrolér schopný dosažení cíle a objíždění překážek.

Klíčová slova

Evoluční algoritmy; Genetické programování; mobinlí robotika;

Abstract

Evolution algorithms have proven to be useful in finding solutions to many tasks from various fields. Their usage in mobile robotics can be beneficial for finding novel solutions to some tasks, and could also increase the level of autonomy of mobile robots. This work aims to examine the usage of genetic programming for goal reaching and collision avoidance tasks of a mobile robot equipped with the laser sensor and information about the goal position. For these tasks, a grammar for controllers was defined. Genetic programming was used for developing controllers of two types: structure-free and structure-restricted. Obtained controllers were tested in a simulation environment. This work describes used fitness functions and controllers developed using them. It was shown, that it is possible to find a controller capable of goal reaching and collision avoidance task using genetic programming.

Keywords

Evolutionary methods; Genetic programming; collision avoidance; obstacle avoidance; mobile robotics; wheeled robots;

Contents

Abbreviations

In this thesis following abbreviations are used:

CFG	Context free grammar
DS	Derivation step
EA	Evolutionary algorithms
GP	Genetic programming

List of Figures

List of Tables

1 Introduction

For few decades autonomous mobile robots were used in many areas including security, military and industrial applications. In these areas robots performed tasks, that could be dangerous or difficult for a human [1]. Nowadays mobile robots are also spreading into the new areas, such as household and health care [2]. Here they are used to simplify daily activities and reduce the amount of work performed by human, e.g. for lawn mowing or cleaning [3]. Such large area of usage demands a collision avoidance behaviour, so that robot doesn't damage itself or objects around. In industrial applications such task may be solved by the path planning, but in other areas where the environment may be unknown and changing the other approach is needed. This task requires such controller which would use data from sensors to move the robot around the environment safely without collisions.

The autonomy of the mobile robot is desirable in the civil mobile robotics. In military or exploration tasks it may be the necessity. Since it's hard to program the behaviour of the robot in all possible situations, the idea of the robot, that is programming itself, is very attractive. The evolutionary methods are a promising way to ensure the autonomy of the robot, because they are offering a possibility to create programs without human interaction. If necessary, the robot could only specify the required result and the solution to a problem could be found by a suitable evolutionary algorithm.

This thesis is focused on the analysis of controller types and fitness functions used in evolutions of the robot motion controllers.

1.1 Evolutionary algorithms

Evolutionary algorithms (EA) are one of the approaches in the artificial intelligence used for solving optimization problems. The EA uses mechanisms similar to the ones found in the process of the biological evolution, such as selection, reproduction and mutation.

Similarly to the biological evolution, the most adapted to the environment individuals are rewarded by the opportunity to participate in the reproduction. Their genes are propagated to the next generations to create even more adapted offspring. In the biological evolution, the level of adaptation is determined by the natural selection. In the EA the fitness function is used to evaluate the capabilities of each individual. The new generation is created from the genes of the most adapted individuals proportionally to their fitness score [4].

The robot motion controllers in this thesis are created using the genetic programming (GP). The GP is a methodology based on the evolutionary algorithms, where the evolving individuals are programs. The fitness of these programs is determined by their ability to solve the predefined task.

2 Grammatically-based Genetic Programming

2.1 Genetic programming

Genetic programming is one of the methodologies based on the biological evolution, created to find programs that perform predefined tasks. A process of finding the optimal result is called evolution or learning. An initial population is formed as few randomly generated programs in the beginning of the evolution. A process of learning is then started. The evolution ends after a predefined number of generations passed or when the optimal result is discovered. Generated programs are compared by the score of the fitness function, which evaluate the result of every generated program. The value returned by the fitness function shows how adapted the tested controller is. Evolution select the fittest individuals in every generation to participate in breeding and create the next generation.

In this thesis genetic programming is used to create programs that control robot movement to avoid obstacles in simulated and real environment. Each program has a form of a mathematical expression where inputs from robot sensors, goal's location and some constants can be used to calculate desired velocity of robot wheels. The context-free grammar is used for the creation and the representation of these expressions. This grammar enables a simple creation of syntactically correct programs and ensures that they remain correct after two of them swap components.

2.2 Context free grammar

A Context-Free Grammar (CFG) G is a set of grammar rules to describe a language. CFG is defined as a quadruple $G = (N, T, P, S)$, where

- N is a set of non-terminal symbols (non-terminals),
- T is a set of terminal symbols (terminals),
- P is a set of productions (rules, replacing rules),
- S is a start symbol, symbol from the set N. [5]

Two sets of symbols are used to define a CFG. First set T includes terminal symbols, which makes the content of a legal program (sentence). The second set N includes non-terminal symbols, that are useful for creating a structure of a program. Non-terminals define sub-languages of a language described by G. In programs they can represent various data types.

Start symbol $S \in N$ is a non-terminal representing the whole language defined by G.

Productions from the set P are used to define rules by which symbols can be replaced by other symbols. They define how each type of non-terminals can be expanded by strings of non-terminals or terminals.

Productions can be written in a form

$$\alpha \to \beta, \quad \text{where } \alpha \in N,\ \beta \in \{T \cup N\}^*.$$

In the notation above α is a non-terminal and β is representing a string of zero or more terminals and/or non-terminals.

Several productions defined for a non-terminal α which have a form $\alpha \to \beta_1$, $\alpha \to \beta_2$ may be also written as $\alpha \to \beta_1 \mid \beta_2$.

2.2.1 Derivation step

Programs created by genetic programming for this thesis have a form of mathematical expression, which is a string of terminals. These strings are derived by applying productions on every non-terminal starting with the start symbol (first string). Derivation step rewrites the actual string containing non-terminals and zero or more terminals to another string of non-terminals and/or terminals. Syntactically correct programs are created by performing several derivation steps until all the symbols in string are terminals.

Derivation step (DS) is defined as an application of a production from P to a non-terminal $\alpha \in N$ [5].

Derivation step can be written as:

$$A\ \alpha\ C \overset{\alpha\to\beta}{\Longrightarrow} A\ \beta\ C$$

where $\alpha \in N$ is a non-terminal and $A, C, \beta \in \{T \cup N\}^*$ are strings of zero or more terminals and/or non-terminals. In the relation above string $x = A\ \alpha\ C$ has been changed to string $y = A\ \beta\ C$ by applying the production $\alpha \to \beta$.

Repeated derivation rooted in a non-terminal A has a form

$$A \overset{*}{\Longrightarrow} \alpha$$

where $A \in N$ and $\alpha \in \{T \cup N\}^*$. Symbol $\overset{*}{\Longrightarrow}$ represents zero or more derivation steps. Such series of derivation steps may be represented as a tree. Start symbol is placed in the root of tree, non-terminals are placed at branch nodes, terminals are placed at leaf nodes. Each derivation step is pictured as an expansion of a non-terminal node to various combinations of terminals and non-terminals.

2.2.2 Example of CFG program

For creating an expression

$$((a_1 - a_2) + (a_3 \times a_4))$$

where a_1, a_2, a_3, a_4 are real numbers, many grammars may be used. One of the possible grammars is described below and contains only the minimal number of symbols and productions.

Type of symbols	Symbols
Terminals	$a_1\ a_2\ a_3\ a_4\ +\ -\ \times$
Non-terminals	$S\ R$

Table 1 Possible set of symbols to create a program with CFG

Terminals $+$, $-$ and $\times$ are mathematical operators of addition, subtraction and multiplication. These terminals only occur on the right side of productions together with

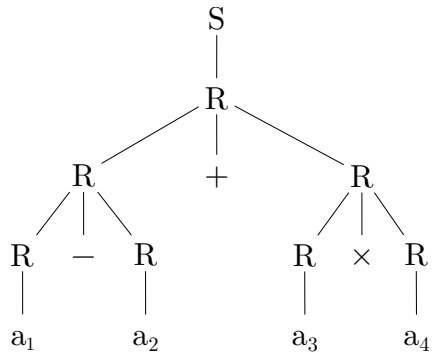


Figure 1 Example of a program made with CFG

other terminals or non-terminals (in this case only non-terminals). Terminals a_1, a_2, a_3, a_4 are terminals representing inputs of the program. In contrast to in-function terminals these may occur alone on the right side of the production.

The non-terminal R represents real numbers, the non-terminal S is a start symbol (R may also be used as a start symbol, then only one type of non-terminal will exist).

Productions defined for this grammar may have the following form

$$S \to R$$
$$R \to R + R \mid R - R \mid R \times R \mid a_1 \mid a_2 \mid a_3 \mid a_4$$

For representing the series of derivation steps

$$S \overset{*}{\Longrightarrow} ((a_1 - a_2) + (a_3 \times a_4))$$

a derivation tree which is shown on Fig. 1 may be used.

2.3 Creating of the initial population

Initial population is filled with randomly created programs. It is recommended for initial population to have some level of diversity [5, 6], so that different functions and inputs were presented in gene pool. Diversity is important, because the main changes between populations in genetic programming are achieved by crossovers, but when the population is homogeneous the new unique individuals are not likely to be found.

For obtaining the diversity in an initial population few techniques may be used. The two basic, which were described by John Koza [6], are called "full" method and "grow" method. These techniques are using only one parameter, maximum depth D, to control enlargement of the trees.

- "Full" method creates an initial population, where all the trees are of the full depth. For the trees created with CFG that means, that leaf nodes made of input terminals may only occur at the maximum depth D.
- "Grow" method creates trees with depth not exceeding the maximum depth D. This method allows algorithm to create derivation trees of various sizes and forms.

John Koza [6] also presented more complex technique, which he called "ramped half-and-half" method. This technique combines both "full" and "grow" methods to ensure the diversity of the initial population.

This technique creates equally large groups of derivation trees distinguished between each other by different depth parameter. This parameter may have a value in a range

between the smallest possible tree depth defined for used grammar and any bigger number specified by user. Half of the derivation trees from each group is created using the "grow" method and other half using the "full" method.

"Ramped half-and-half" method ensures, that initial population includes derivation trees with several different depths and shapes, however it doesn't guarantee, that duplicates don't occur. Identical trees in the initial population are not desirable, because same computing resources and time are spent on both duplicates, when they could be used more effectively by processing other individuals. It is recommended to avoid adding duplicates to the initial population.

In this thesis "ramped half-and-half" technique is used. If size of the population is not divisible by number of used depths, the method and the depth parameter of excessed trees are chosen randomly.

Individuals of the initial population are formed using algorithm 1 and CFG defined by a quadruple (N, T, P, S). Productions have a form $\alpha \rightarrow \beta$, where $\alpha \in N$ and $\beta \in \{N \cup T\}$. For every production $minDS$ is defined as a minimal number of derivation steps that are necessary to create only terminals, i.e. minimal number of derivation steps to create $\alpha \overset{\alpha \rightarrow \beta}{\Longrightarrow} \beta \overset{*}{\Longrightarrow} \gamma$ where $\gamma \in T^*$. Individuals are formed as derivation trees with the limited maximum depth D.

Creation of every new derivation tree starts with placing a start symbol S to its root and to a queue of non-terminals that don't have defined offspring. Afterwards following steps are repeated until the queue is not empty:

1. Select the first item in the queue and label it as the current non-terminal C
2. Define variable cD as a depth of C
3. Randomly select a production $P_C \in P$ of the form $C \rightarrow \beta$ with $minDS_C \leq (D - cD)$
4. Add every non-terminal $F \in \beta$ to the queue
5. Delete C from the queue and return to step 1

The algorithm above describes the way to create a derivation tree using the "grow" method. To create the "full" derivation tree another restriction should be added to a step 3.

Selected productions should additionally satisfy the condition $maxDS_C \geq (D - cD)$, where $maxDS_C$ is the maximum number of derivation steps to create only terminals. For some grammars it's possible, that "full" trees defined as above don't exist, for example because some terminals can only have the depth less than D. In such cases the definition of "full" trees may be adjusted to the requirements of the certain grammar. Algorithms appropriate for new definition are then used to create trees.

2.4 Reproduction

In this thesis GAlib library for C++ language is used to control the process of learning. The "simple" genetic algorithm was selected for all evolutions. This genetic algorithm selects the best individuals of every population based on their fitness scores for a reproduction to create the entirely new generation [7]. The best individual is copied to the text generation. New individuals are made as a result of a process called crossover, when two programs swap their parts. Every individual produced this way may also be changed by a random mutation, which replaces randomly selected part of a program by a newly created one.

Data: An empty program *tree*, maximum tree depth D
Result: A new program *tree*
$listToProcess \leftarrow$ new list of non-terminal nodes without defined offspring;
insert S into *tree* as root;
insert S into $listToProcess$;
repeat
  $currentNode \leftarrow$ first element of $listToProcess$;
  $cD \leftarrow$ depth of $currentNode$;
  $P_C \in P \leftarrow$ randomly chosen production with
    $\alpha_C = currentNode$ & $(minDS_C \leq (D - cD))$;
  remove $currentNode$ from $listToProcess$;
  foreach $newNode \in \beta_C$ **do**
    insert $newtNode$ into *tree* below $currentNode$;
    if $newNode \in N$ **then**
      insert $newNode$ into $listToProcess$;
    end
  end
until $listToProcess = \varnothing$;

Algorithm 1: Creating of the initial program using the CFG (N, T, P, S) with productions P_i of form $\alpha_i \rightarrow \beta_i$ and minimal number of DS to create only terminals $minDS_i$

2.4.1 Crossover

Crossover is an operation of the part exchange between two programs.

In classical GP this operation was a difficult task, because of the problem with choosing crossover points so that new individuals were grammatically correct, but CFG offers a very elegant way to perform a crossover.

Context Free grammar defines two types of symbols: terminals and non-terminals. Since every type of non-terminal has a defined set of productions, grammatically correct structures are produced when sub-trees below non-terminals of the same type are swapped. The structure of derivation trees made with CFG ensures, that there is at least one non-terminal above every terminal in every individual. Due to this facts non-terminals are the perfect candidates for crossover points.

Pseudo-code for a crossover of two derivation trees is shown at algorithm below. During every operation children are created as copies of parent trees. After that non-terminals of the same type are randomly selected in both children and sub-trees below them are swapped.

Data: Parent program p_1, parent program p_2
Result: Child program c_1, child program c_2
$c_1 \leftarrow p_1$;
$c_2 \leftarrow p_2$;
repeat
  $n_a \in \{N \cap c_1\} \leftarrow$ randomly chosen non-terminal from c_1;
  $n_b \in \{N \cap c_2\} \leftarrow$ randomly chosen non-terminal from c_2;
until $n_a = n_b$;
swap the subtrees below n_a and n_b;

Algorithm 2: Crossover algorithm for the CFG (N, T, P, S)

An example of crossover is shown at Fig. 2. In two parent trees p_1 and p_2, which are

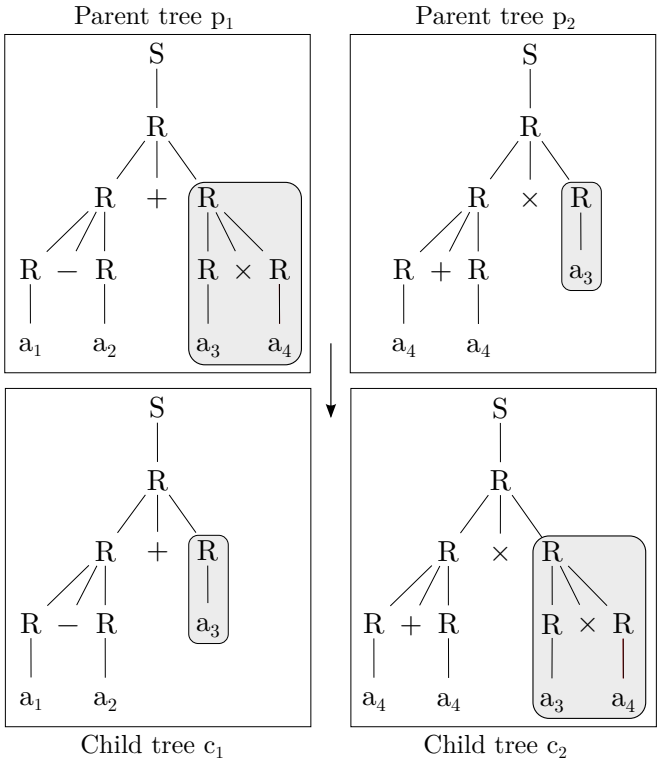


Figure 2 Example of a crossover of programs made with CFG

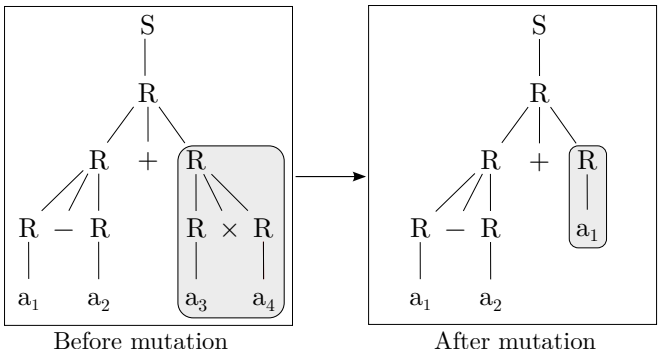


Figure 3 Example of a mutation of program made with CFG

pictured on top, non-terminals of type R were selected as crossover points. Children c_1 and c_2, pictured on the bottom, are produced by algorithm 2.

This figure also demonstrates, that as non-terminals make all the branch nodes of a derivation tree, crossover may dramatically change a structure and a content of parent trees if the right crossover points are chosen.

2.4.2 Mutation

Similarly to a natural evolution, in GP individuals get their properties directly from their parents. Once in a while random mutation happens and additionally changes some of children qualities. This change has a very little probability to occur both in natural evolution and GP.

After every crossover newly created individual may randomly mutate, with the probability defined by user. Mutation applies to a single program. A pseudo-code for this

operation is shown in algorithm below. When the program is chosen for a mutation a non-terminal is randomly selected in its derivation tree, sub-tree below this non-terminal is then replaced by new randomly created sub-tree. An example of a mutation is shown on Fig. 3.

Data: Parent program p
Result: Child program c
$c \leftarrow p$;
$n \in \{N \cap c\} \leftarrow$ randomly selected non-terminal from c;
$sub \leftarrow$ new tree created using algorithm 1 with the CFG (N, T, P, n);
replace subtree below n with sub;

Algorithm 3: Mutation algorithm for the CFG (N, T, P, S)

3 General structure of robot motion controllers

In this chapter, the usage of CFG for the representation of the robot motion controllers is discussed. The used grammar is described in section 3.1. The implemented way of execution of the created program is described in section 3.2. Types of the used controllers are described in 3.3.

3.1 CFG for motion controllers

Context free grammar which was used to form programs in this thesis may be described as $G = (N, T, P, S)$, where set of non-terminals is $N = \{S, R\}$, set of terminals is $T = \{in_1, \ldots, in_{14}, min, \ldots, ifGrDelay\}$ and set of productions P is shown in Tab. 2.

As it's seen above, two types of non-terminals are used. Non-terminals R represent parts of the program, while S stands for the whole program. There is no fundamental difference between these types of non-terminals. The only reason S is used is to easily distinguish the start of program tree, i.e. output. In our implementation crossover and mutation points are only located in non-terminals of R type.

Depending on their functions, terminals may be divided into two groups. In-function terminals $\{min, \ldots, ifGrDelay\}$ are identified by strings, and occur only as first elements on the right side of productions and represent operations, which are made on vectors placed in the corresponding non-terminals. Terminals of this type are shown in Tab. 2 as parts of productions.

Input terminals $T = \{in_1, \ldots, in_{14}\}$, listed in Tab. 3, are the actual data available for programs, they represent constants or information about environment received by

Subset	Production	Subset	Production
1	S → R	1	R → productNumber R R
1	R → min R	1	R → ratioNumber R R
1	R → max R	1	R → in_1 | ... | in_6
1	R → mean R	2	R → cutLeft R
1	R → plusLong R R	2	R → cutRight R
1	R → plusShort R R	2	R → double R
1	R → minusLong R R	2	R → leftThird R
1	R → minusShort R R	2	R → centralThird R
1	R → productLong R R	2	R → rightThird R
1	R → productShort R R	2	R → in_7 | ... | in_9
1	R → ratioLong R R	3	R → mergeFirst R R
1	R → ratioShort R R	3	R → plusMinus R R
1	R → mergeVectors R R	3	R → ifGreater R R R R
1	R → plusNumber R R	3	R → in_{10} | ... | in_{14}
1	R → minusNumber R R	3	S → ifGrDelay R R R R R

Table 2 Set of productions available for GP during the evolution process.

Type of input	Symbol	Terminal
Robot wheel velocity	in_1	[left wheel, right wheel]
Goal position	in_2	Cartesian coordinates: [x, y, z]
Goal position	in_3	$[-\tan\alpha]$, where α is the polar angle
Goal position	in_4	[distance to the goal]
Constant	in_5	[1.0, 1.0]
Constant	in_6	[0.0, 0.0]
Constant	in_7	[5.6]
Constant	in_8	[0.2]
Input from laser sensor	in_9	All data from laser sensor: $[i_0, \ldots, i_{679}]$
Input from laser sensor	in_{10}	The 1 fifth of in_9: $[i_0, \ldots, i_{135}]$
Input from laser sensor	in_{11}	The 2 fifth of in_9: $[i_{136}, \ldots, i_{271}]$
Input from laser sensor	in_{12}	The 3 fifth of in_9: $[i_{272}, \ldots, i_{407}]$
Input from laser sensor	in_{13}	The 4 fifth of in_9: $[i_{408}, \ldots, i_{543}]$
Input from laser sensor	in_{14}	The 5 fifth of in_9: $[i_{544}, \ldots, i_{679}]$

Table 3 List of input terminals available for GP during the evolution process

robot during simulation.

Four out of fourteen terminals are constants, they are defined once at the beginning of evolution and are never changed again. Such terminals may be useful as references. Constants has indexes five to eight. Terminal in_5 is a vector of two ones, in_6 is a vector of two zeros, in_7 is a vector with one element, which is a range of a laser sensor (5.6 m), in_8 is a vector with one element—distance from laser sensor to obstacle, on which penalizations are started.

Remaining terminals are changing their values during simulation. These terminals provide robot with knowledge about itself and surrounding environment. Terminal in_1 is a vector with two elements: left and right wheel velocities of a robot in $\mathrm{rad\,s^{-1}}$. Terminals in_2–in_4 specify the goal position relative to a robot: in_2 gives Cartesian coordinates of the goal in meters, in_3 gives $-\tan\alpha$, where α is a polar angle of the target, and in_4 gives the distance to the target in meters. Inputs in_9–in_{14} represent data from a laser sensor. Terminal in_9 is a vector that contains data from the whole angular range of laser sensor. Remaining terminals in_{10}–in_{14} contain a fifth part of in_9 each, in given order starting on the left part of the robot.

Since for different evolutions different grammars were used, to distinguish between them P is divided into three subsets P_1–P_3. Each subset is labelled with a number to highlight its usage in evolutions.

In this work, productions are used to construct mathematical operations which will be performed on input vectors. Description of production structures and its execution is in section 3.2.

3.2 Execution of CFG program

As it was specified above, each program is constructed as a tree, therefore each one has multiple inputs placed in leaf nodes and one output placed in a root. Each node except for in-function terminals has few attributes including a vector of real numbers. During execution vectors of non-terminals are successively specified until start symbol is reached and the output is generated.

Each production $P : \alpha \to \beta$ is viewed as an assignment, where vector $\mathbf{v}$ in α is

calculated by processing an operation coded in β. In this thesis $\beta \in \{T \cup N\}^*$ may consist of one to six symbols. If β contains just one symbol, which may be both terminal or non-terminal with specified vector $\mathbf{a}$, then $\mathbf{v} = \mathbf{a}$. Otherwise, if β contains more than one symbol, $\mathbf{v}$ is obtained by changing vectors in β. In such case, β is composed of one to five non-terminals with vectors, which are marked with symbols **a**–**e**, and one in-function terminal *operation*, which specifies rules by which $\mathbf{v}$ is calculated from vectors **a**–**e**. Vector $\mathbf{v}$ is obtained by applying *operation* on vectors **a**–**e**, written as $\mathbf{v} = operation(\mathbf{a}, \ldots, \mathbf{e})$. List of used operations with associated productions is shown in Tab. 4.

3.3 Structure-free and structure-restricted controllers

During work on this thesis two types of controllers were developed by GP depending on level of its autonomy. Controllers of the first type were created completely by the evolution process, without restrictions applied on their structure. The first production in such controllers is $S \rightarrow R$, which doesn't change the output and is used so that reproduction could change the significant part of program. Such controllers need minimum of human interaction to be created. The result of evolution is a universal algorithm, which is capable of reaching the goal without collisions with obstacles. The problem of this approach is that evolution tries to optimise several often contradicting values. For example, avoidance of a collision may lead to increasing of distance from the goal. Such conflicts may cause, that desired controller might not evolve.

This is why structure-restricted controllers were also designed. In contrast to structure-free programs, these algorithms include parts defined by user, which can't be changed by evolution. This approach allows user to bias the evolution towards more promising results or to assemble controllers produced by different evolutions.

Production	Output vector
S → R	$\mathbf{v} = \mathbf{a}$
R → min R	$\mathbf{v} = [\min(\mathbf{a}_1, \ldots, \mathbf{a}_n)]$, where $n = \lvert\mathbf{a}\rvert$
R → max R	$\mathbf{v} = [\max(\mathbf{a}_1, \ldots, \mathbf{a}_n)]$, where $n = \lvert\mathbf{a}\rvert$
R → mean R	$\mathbf{v} = [\text{mean}(\mathbf{a}_1, \ldots, \mathbf{a}_n)]$, where $n = \lvert\mathbf{a}\rvert$
R → plusLong R R	$\mathbf{v} = [\mathbf{a}_1 + \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{b}\rvert} + \mathbf{b}_{\lvert\mathbf{b}\rvert}, \mathbf{a}_{\lvert\mathbf{b}\rvert+1}, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert}]$, if $\lvert\mathbf{a}\rvert > \lvert\mathbf{b}\rvert$, otherwise $\mathbf{v} = [\mathbf{a}_1 + \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} + \mathbf{b}_{\lvert\mathbf{a}\rvert}, \mathbf{b}_{\lvert\mathbf{a}\rvert+1}, \ldots, \mathbf{b}_{\lvert\mathbf{b}\rvert}]$,
R → plusShort R R	$\mathbf{v} = [\mathbf{a}_1 + \mathbf{b}_1, \ldots, \mathbf{a}_n + \mathbf{b}_n]$, where $n = \min(\lvert\mathbf{a}\rvert, \lvert\mathbf{b}\rvert)$
R → minusLong R R	$\mathbf{v} = [\mathbf{a}_1 - \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{b}\rvert} - \mathbf{b}_{\lvert\mathbf{b}\rvert}, \mathbf{a}_{\lvert\mathbf{b}\rvert+1}, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert}]$, if $\lvert\mathbf{a}\rvert > \lvert\mathbf{b}\rvert$, otherwise $\mathbf{v} = [\mathbf{a}_1 - \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} - \mathbf{b}_{\lvert\mathbf{a}\rvert}, -\mathbf{b}_{\lvert\mathbf{a}\rvert+1}, \ldots, -\mathbf{b}_{\lvert\mathbf{b}\rvert}]$,
R → minusShort R R	$\mathbf{v} = [\mathbf{a}_1 - \mathbf{b}_1, \ldots, \mathbf{a}_n - \mathbf{b}_n]$, where $n = \min(\lvert\mathbf{a}\rvert, \lvert\mathbf{b}\rvert)$
R → productLong R R	$\mathbf{v} = [\mathbf{a}_1 \times \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{b}\rvert} \times \mathbf{b}_{\lvert\mathbf{b}\rvert}, \mathbf{a}_{\lvert\mathbf{b}\rvert+1}, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert}]$, if $\lvert\mathbf{a}\rvert > \lvert\mathbf{b}\rvert$, otherwise $\mathbf{v} = [\mathbf{a}_1 \times \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} \times \mathbf{b}_{\lvert\mathbf{a}\rvert}, \mathbf{b}_{\lvert\mathbf{a}\rvert+1}, \ldots, \mathbf{b}_{\lvert\mathbf{b}\rvert}]$,
R → productShort R R	$\mathbf{v} = [\mathbf{a}_1 \times \mathbf{b}_1, \ldots, \mathbf{a}_n \times \mathbf{b}_n]$, where $n = \min(\lvert\mathbf{a}\rvert, \lvert\mathbf{b}\rvert)$
R → ratioLong R R	$\mathbf{v} = [\mathbf{a}_1 \div \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{b}\rvert} \div \mathbf{b}_{\lvert\mathbf{b}\rvert}, \mathbf{a}_{\lvert\mathbf{b}\rvert+1}, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert}]$, if $\lvert\mathbf{a}\rvert > \lvert\mathbf{b}\rvert$, otherwise $\mathbf{v} = [\mathbf{a}_1 \div \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} \div \mathbf{b}_{\lvert\mathbf{a}\rvert}, \mathbf{b}_{\lvert\mathbf{a}\rvert+1}, \ldots, \mathbf{b}_{\lvert\mathbf{b}\rvert}]$,
R → ratioShort R R	$\mathbf{v} = [\mathbf{a}_1 \div \mathbf{b}_1, \ldots, \mathbf{a}_n \div \mathbf{b}_n]$, where $n = \min(\lvert\mathbf{a}\rvert, \lvert\mathbf{b}\rvert)$
R → mergeVectors R R	$\mathbf{v} = [\mathbf{a}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert}, \mathbf{b}_1, \ldots, \mathbf{b}_{\lvert\mathbf{b}\rvert}]$
R → in_1 | … | in_6	$\mathbf{v} = \mathbf{a}$
R → cutLeft R	$\mathbf{v} = [\mathbf{a}_n, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert}]$, where $n = \lceil\lvert\mathbf{a}\rvert \div 2\rceil$
R → cutRight R	$\mathbf{v} = [\mathbf{a}_1, \ldots, \mathbf{a}_n]$, where $n = \lfloor\lvert\mathbf{a}\rvert \div 2\rfloor$
R → double R	$\mathbf{v} = [2 \times \mathbf{a}_1, \ldots, 2 \times \mathbf{a}_{\lvert\mathbf{a}\rvert}]$
R → leftThird R	$\mathbf{v} = [\mathbf{a}_1, \ldots, \mathbf{a}_n]$, where $n = \text{round}(\lvert\mathbf{a}\rvert \div 3)$
R → centralThird R	$\mathbf{v} = [\mathbf{a}_n, \ldots, \mathbf{a}_m]$, where $n = \text{round}(\lvert\mathbf{a}\rvert \div 3)$, $m = \lvert\mathbf{a}\rvert - n$
R → rightThird R	$\mathbf{v} = [\mathbf{a}_n, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert}]$, where $n = \lvert\mathbf{a}\rvert - \text{round}(\lvert\mathbf{a}\rvert \div 3)$
R → plusNumber R R	$\mathbf{v} = [\mathbf{a}_1 + \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} + \mathbf{b}_1]$
R → minusNumber R R	$\mathbf{v} = [\mathbf{a}_1 - \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} - \mathbf{b}_1]$
R → productNumber R R	$\mathbf{v} = [\mathbf{a}_1 \times \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} \times \mathbf{b}_1]$
R → ratioNumber R R	$\mathbf{v} = [\mathbf{a}_1 \div \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} \div \mathbf{b}_1]$
R → in_7 | … | in_9	$\mathbf{v} = \mathbf{a}$
R → mergeFirst R R	$\mathbf{v} = [\mathbf{a}_1, \mathbf{b}_1]$
R → plusMinus R R	$\mathbf{v} = [\mathbf{a}_1 + \mathbf{b}_1, \mathbf{a}_2 - \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} + \mathbf{b}_1]$ if $\lvert\mathbf{a}\rvert$ is odd, otherwise $\mathbf{v} = [\mathbf{a}_1 + \mathbf{b}_1, \mathbf{a}_2 - \mathbf{b}_1, \ldots, \mathbf{a}_{\lvert\mathbf{a}\rvert} - \mathbf{b}_1]$
R → ifGreater R R R R	$\mathbf{v} = \mathbf{c}$ if $\mathbf{a}_1 > \mathbf{b}_1$, otherwise $\mathbf{v} = \mathbf{d}$
R → in_{10} | … | in_{14}	$\mathbf{v} = \mathbf{a}$
S → ifGrDelay R R R R R	$\mathbf{v} = \mathbf{c}$ if $\mathbf{a}_1 > \mathbf{b}_1$, otherwise $\mathbf{v} = \mathbf{e}$ for $\mathbf{d}$ seconds

Table 4 Set of productions available for GP during the evolution process. Every production has one output vector $\mathbf{v}$ on the left side, and on the right side one to five input vectors $\mathbf{a}$–$\mathbf{e}$ placed in non-terminals starting from the left. Symbol $\lvert\mathbf{a}\rvert$ stands for number of elements in vector $\mathbf{a}$.

4 Experiments

In this chapter the simulation environment and fitness functions used in the evolutionary process will be discussed.

4.1 Simulation environment

In genetic programming it's required that every generated individual is tested under the same conditions. In the real environment it is impossible to ensure such thing during evolution, where hundreds and thousand evaluations are performed. Furthermore, maps on which generated algorithms are tested contain obstacles and since robots are mobile, programs from first generations would most likely lead robot to collision and may damage real robots. In order to avoid this disadvantages, tests were performed in a simulator called V-REP [8, 9] (The Virtual Robot Experimentation Platform).

V-REP allows user to create scenes filled with the diversity of scene objects: robots, obstacles and sensors. It offers several existing models available for usage, as well as tools to edit them and their functionality or to create new ones. The program uses Bullet physics library [10] for simulation of the movement of objects, collision detection and response. Further advantage of V-REP is that it "implements a ROS [11, 12] node with a plug-in which allows ROS to call V-REP commands via ROS services, or stream data via ROS publishers/subscribers", as stated in [9]. This functions make possible to switch between created scenes, start and stop simulation from program in which evolution is running (main program). Data exchange between simulator and main program is performed via ROS publishers and subscribers on the 10 Hz update rate. Simulator provides program with data from laser sensor, information about goal position and information about simulator state, including simulation time. Main program processes these data and publishes the required velocity of robot wheels. Since the simulation doesn't take place in real time and V-REP controlling via ROS services may have variable delay, main program also publishes the duration of the simulation. The simulator may then stop the robot when needed and publish information about the end of the simulation. This ensures that all controllers are tested for the same amount of time.

Robot dr12 (courtesy of Cubictek co. ltd.) with attached Hokuyo URG 04LX UG01 laser sensor, both from V-REP model collection, was used in simulations of tested controllers. Sensor provides information about obstacle distances for 680 points in range 5.6 m×240°. The script for this laser sensor originally published data in the PointCloud2 message type, but was adjusted to publish in the LaserScan message type.

4.2 Maps

Three pairs of maps, that were used for evolution of collision avoidance algorithms, are shown in figures below. In these figures white color shows the area on which robot may move, light-blue color shows areas where obstacles are placed and yellow square shows the goal.

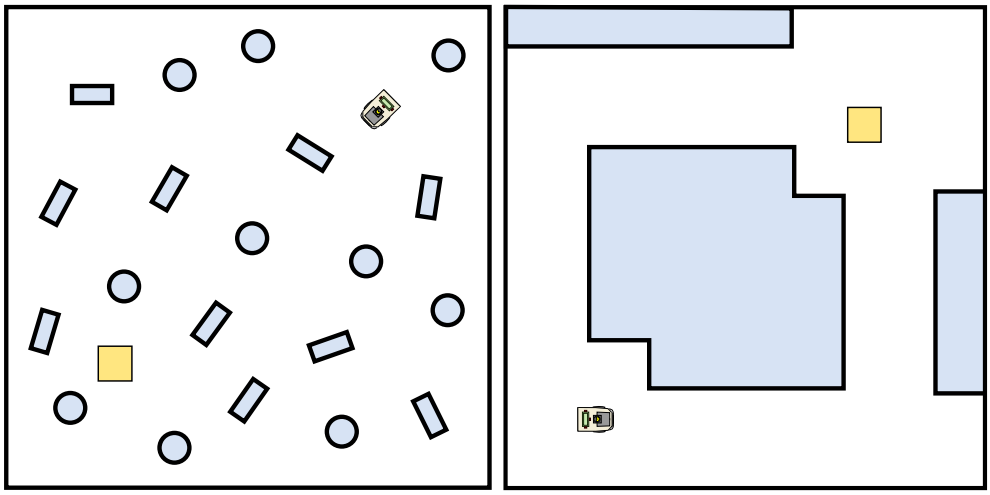


Figure 4 Maps used for the evolution of collision avoidance controllers

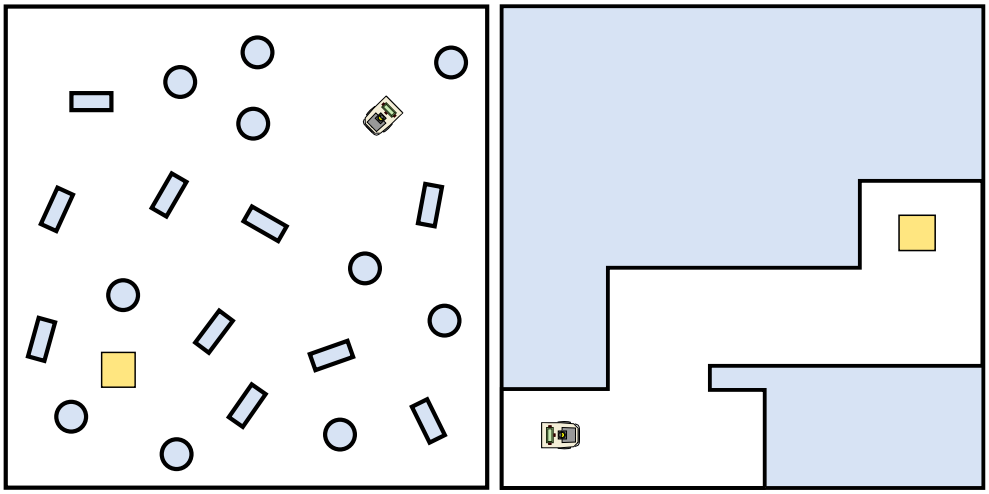


Figure 5 Maps used for the evolution of collision avoidance controllers

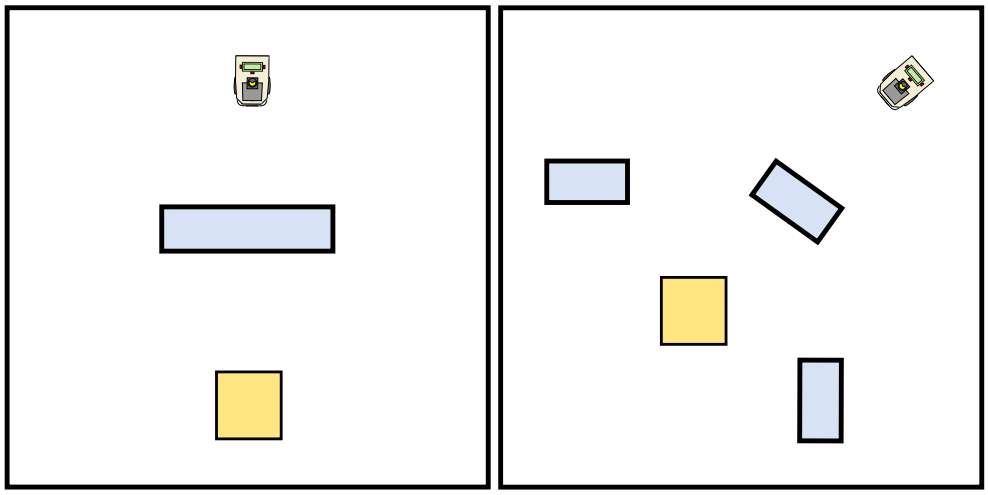


Figure 6 Maps used for the evolution of collision avoidance controllers

Maps shown in Fig. 4 were used in first evolutions of structure-free collision avoidance controllers. The map depicted on the left was filled with randomly placed obstacles, the robot and the goal were placed in the opposite corners. The map depicted on the right was created with corridors connecting two rooms. Rooms were placed in opposite corners, where the robot and the goal were located.

This maps were replaced after several evolutions, due to the following problem. Robots that went straight to goal had a disadvantage already on early stages of evolution, because an obstacle was placed ahead of them. Every algorithm that lead the robot straight to the target caused immediate collision with an obstacle and had worse fitness scores, than algorithms which made robot go the other way. To eliminate such disadvantage the new set of maps shown in Fig. 5 was created.

Scenes shown in Fig. 6 were created for evolution of structure-restricted controllers. These controllers were composed of two parts: a goal reaching algorithm defined before the start of the evolution and an algorithm of collision avoidance, which was generated by the evolution. These maps were created smaller with only few obstacles to make the task simpler.

4.3 Structure-free controllers

Structure-free controllers were developed in several evolutions in different environments with different fitness functions F, population sizes S, number of generations N_{gen}, probabilities of mutation P_{mut} and simulation durations T_{sim}. Every created controller was tested on $n = 2$ maps to eliminate the influence of surroundings on fitness score. Probability of crossover $P_{cross} = 0.9$ was same for all evolutions. Initial populations were created using "ramped half-and-half" method with minimum tree depth $d_{min} = 3$ and maximum tree depth $d_{max} = 6$.

First implemented fitness function was

$$F = \sum_{i=1}^{n} (1 + D_i) \times (1 + C_i), \tag{1}$$

where D_i is the final distance from the goal and C_i is a number of algorithm evaluations spent near obstacles, i.e. time spent near obstacles in $\mathrm{s} \cdot 10^{-1}$. The goal was to minimize the fitness score.

Two evolutions with $S = 25$, $P_{mut} = 0.01$, $T_{sim} = 30\,\mathrm{s}$ took place on maps shown in Fig. 4. Numbers of generations were $N_{gen\,1} = 46$ and $N_{gen\,2} = 58$. In addition to that one evolution with $S = 20$ and $N_{gen\,3} = 25$ took place on the same maps. All evolutions were using the set of productions $P = P_1 \cup P_2$ and inputs in_1–in_9. All had the similar result. Produced controllers made the robot to stay near the start position. In the first case, the robot was driving around the start, in the second case, it was rotating around the start position and the last one was slowly moving forward.

Such results were more probably caused by variable C_i, which was increasing rapidly every time when the robot approached the obstacle. This way the robot had to travel 1 m towards the goal only to compensate 0.1 seconds near the obstacle. Moreover maps, on which evolution took place, had first obstacles close to a robot. Taking in consideration this conditions, most controllers, that would have tried to move would most probably lose to those which would stay on the start.

These results have shown, that both maps and fitness functions had to change. New scenes depicted in Fig. 5 were created. Obstacles were placed farther from the start

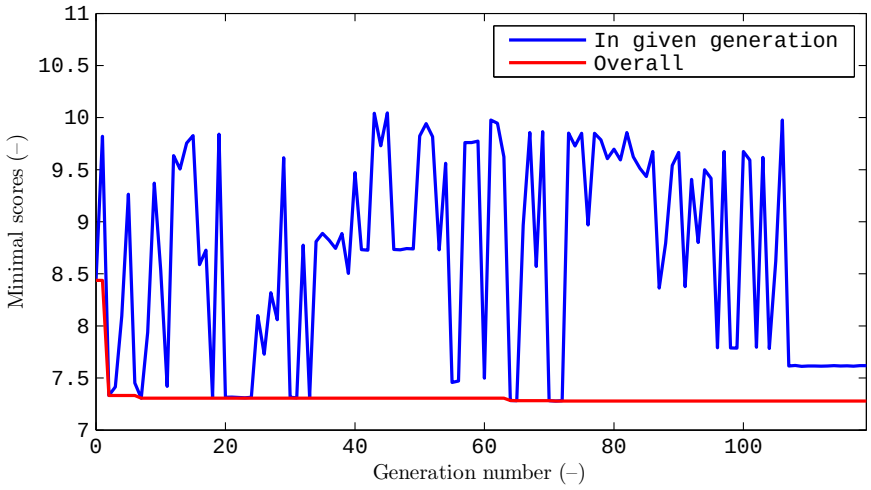


Figure 7 The development of the best score for the first evolution with fitness function 2.

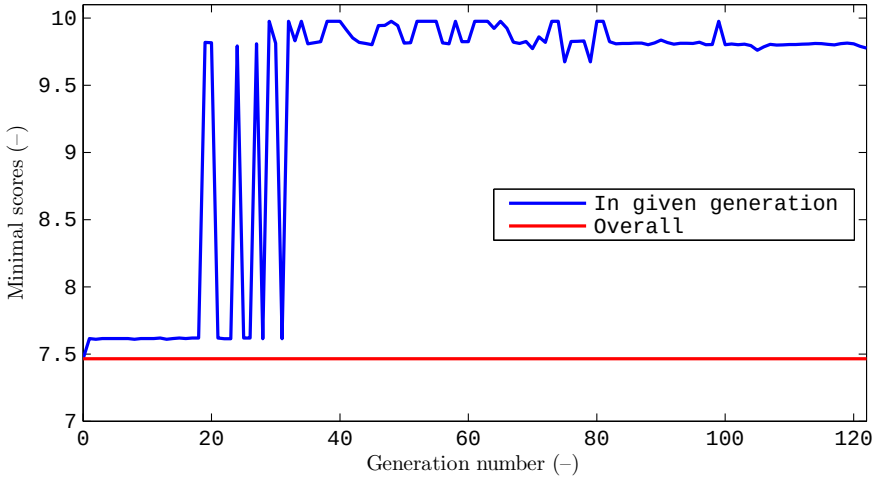


Figure 8 The development of the best score for the second evolution with fitness function 2.

position of the robot, so it could start moving towards the goal without the immediate collision.

The fitness function was changed to the following:

$$F = \sum_{i=1}^{n} (1 + D_i) \times (1 + \gamma_i \times C_i), \tag{2}$$

where variables D_i and C_i are the same as in fitness function 1. Coefficient $\gamma_i \in \mathbb{R}$, $0.01 \leq \gamma_i \leq 0.45$ was increased by 0.01 after every generation until gained its maximum value. Similarly to the function above, the goal was to minimize the fitness score.

Two evolutions with parameters $S = 30$, $P_{mut} = 0.01$, $T_{sim} = 30\,\text{s}$, $N_{gen\,1} = 117$ and $N_{gen\,2} = 122$ took place. For both evolutions set of productions $P = P_1 \cup P_2$ and inputs in_1–in_9 were used. The development of the best scores is shown in Fig. 7 for the first evolution and in Fig. 8 for the second one.

As it seen in these graphs, the best generated controllers were created on early stages of evolution. In the first evolution, the minimal fitness scores have decreased from 8.44 in the initial population to 7.33 in the second generation. It practically didn't changed for the next 119 generations. The best controller had fitness score 7.28 developed in

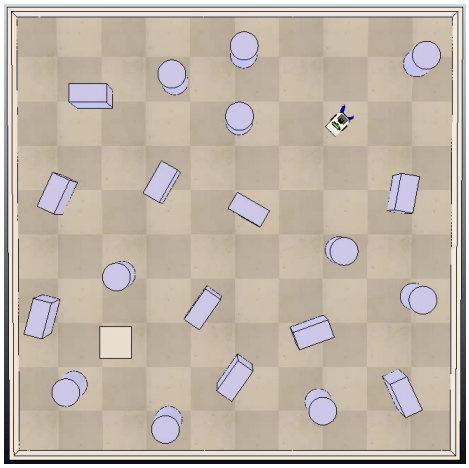


Figure 9 The best controller generated by the first evolution with fitness function 2 on map 1, $T_{sim} = 30\,\text{s}$

Figure 10 The best controller generated by the first evolution with fitness function 2 on map 2, $T_{sim} = 30\,\text{s}$

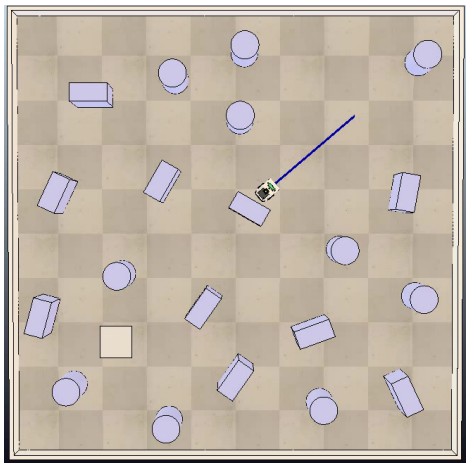


Figure 11 The best controller generated by the first evolution with fitness function 2 on map 1, $T_{sim} = 30\,\text{s}$

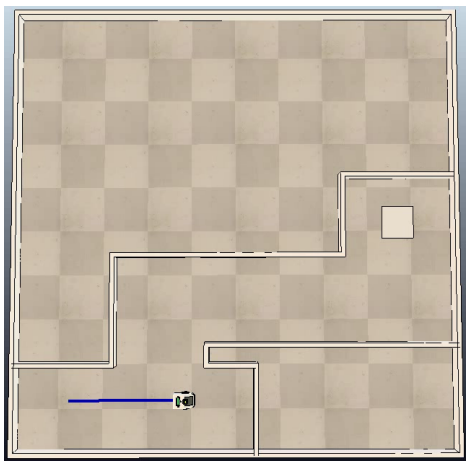


Figure 12 The best controller generated by the first evolution with fitness function 2 on map 2, $T_{sim} = 30\,\text{s}$

71 generation. The robot controlled by this algorithm slowly went forward and didn't turn. The evolution found the optimal velocity, with which the robot haven't collided with an obstacle by the end of the simulation time, as it's seen in Fig. 9 and Fig. 10.

The best score in the second evolution was 7.46. The individual which gained this value was created in the initial population. Later generations didn't manage to improve the scores. The best algorithm slowly moved the robot backwards in the direction of the goal. By the end of the simulation it was near the start position, as it's shown in Fig. 11 and Fig. 11.

As it seen in graphs, the first evolution had shown better adaptation to the changing environment, than the second one. The first evolution managed to repeatedly create individuals with scores close to the best result, when the second one haven't. This could have been due to the difference of the initial populations, but the more probable cause is that the first evolution was actually divided into several sub-evolutions. The fitness function maintained the same. The initial population of every sub-evolution was the last population of the previous one. The γ_i coefficient had the same value in these two

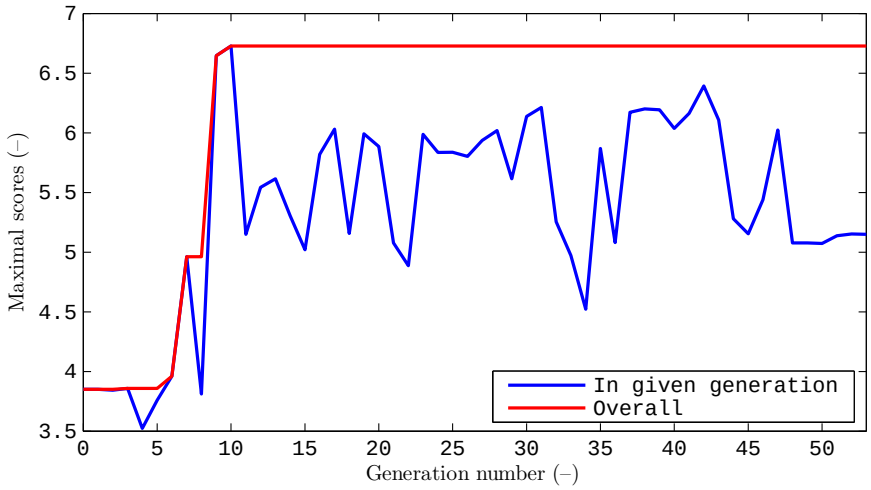


Figure 13 The development of the best score for the first evolution with fitness function 3.

populations and was changed by the same rate as in the second evolution. The first evolution was interrupted after 20 generations, then three times after 21 generations, one time after 20 and continued for 14 more generations before it ended. The second evolution was first interrupted after the 80 generation, when γ_i wasn't changing for several generations. The worse results could have been obtained because the best algorithm from each generation is copied to the next generation. This individual is tested only once on the early stage of the evolution, and it may not be adapted to the changing environment. However, it's involved in breeding and influence all generations, producing not adapted offspring. The first evolution didn't have this disadvantage, because after every interruption the best individual was tested again under new conditions. If it was no longer the best, it influence on the evolution was limited. The best individual is omitted in the statistics.

The results could have been improved by dividing the evolution into sub-evolutions and also by optimizing the initial and the final value of γ_i and the rate of its increase.

The next fitness function was tested under the assumption, that good goal reaching algorithm would avoid obstacles, because collisions would slow the robot down. The fitness score was calculated as

$$F = \sum_{i=1}^{n} G_i + 5 \times \left(1 - \frac{D_i}{D_{i\,max}}\right) + 5 \times \left(1 - \frac{T_i}{T_{max}}\right), \tag{3}$$

where $G_i = 1$ if the robot reached the goal and $G_i = 0$ otherwise. D_i is the final distance from the goal, $D_{i\,max}$ is the maximal possible distance from the goal on current map, T_i is the time needed to reach the goal and T_{max} is the duration of the simulation, both in seconds. If the robot didn't reach the goal by the end of the simulation, $T_i = T_{max}$. The goal was to maximize the result.

Two evolutions with parameters $S = 30$, $P_{mut} = 0.01$, $T_{sim} = 30\,\mathrm{s}$, $N_{gen\,1} = 53$ and $N_{gen\,2} = 80$ took place. For both evolutions set of productions $P = \{P_1 \cup P_2\}$ and inputs in_1–in_9 were used. The development of the best scores is shown in Fig. 13 for the first evolution and in Fig. 14 for the second one.

The best generated controller in the first evolution succeeded in the goal reaching task. However, it was moving backwards without the knowledge about obstacles in its way and was colliding with them. On the first map shown in Fig. 15 robot passed the first obstacle but hit it several times. However, it got stuck on the second map shown

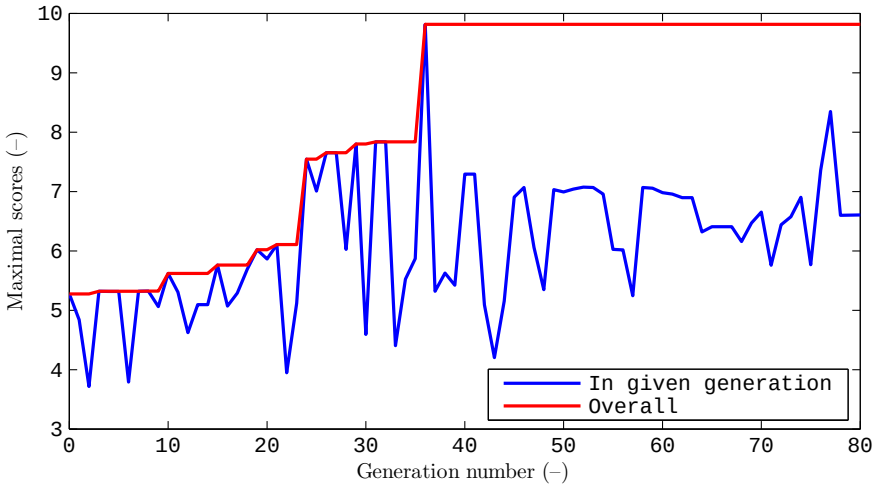


Figure 14 The development of the best score for the second evolution with fitness function 3.

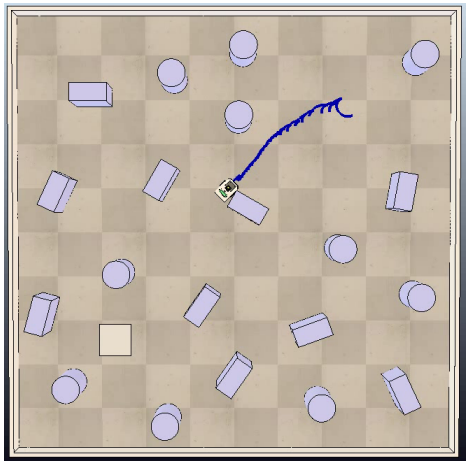


Figure 15 The best controller generated by the first evolution with fitness function 3 on map 1, $T_{sim} = 30\,\mathrm{s}$

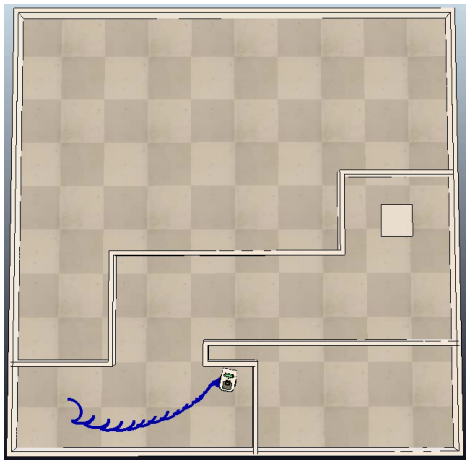


Figure 16 The best controller generated by the first evolution with fitness function 3 on map 2, $T_{sim} = 30\,\mathrm{s}$

in Fig. 16.

The second evolution produced an algorithm, which had an obstacle avoiding behaviour. Its results are shown in Fig. 18 and Fig. 19. The robot haven't collided with an obstacle once in both learning maps and have shown similar results in other scenes, for example in Fig. 17. The robot moved fast in the environment with the corridor and managed to approach the goal in less than 30 s, while on the map with random obstacles it was slow and came close to the goal in more than 90 s. Despite the fact, that the robot was near the goal, it couldn't reach the target even after two minutes.

The next tested fitness function was following:

$$F = \sum_{i=1}^{n} G_i + 5 \times \left(1 - \frac{D_i}{D_{max}}\right) + 100 \times \frac{\mathrm{d}S_i}{N_i} - O_i, \tag{4}$$

where variables G_i, D_i and $D_{i\,max}$ are the same as in fitness function 3. $\mathrm{d}S_i$ is the change of the distance between the robot and the goal between two updates (approximately 0.1 s) in meters, N_i is a number of updates and O_i is a penalization for approaching the

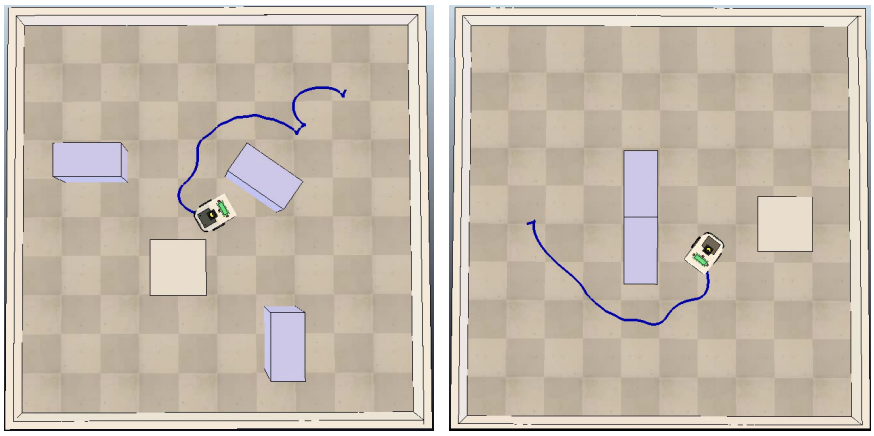


Figure 17 The result of the best controller generated by the second evolution with fitness function 3 on testing map, $T_{sim} = 30\,\mathrm{s}$

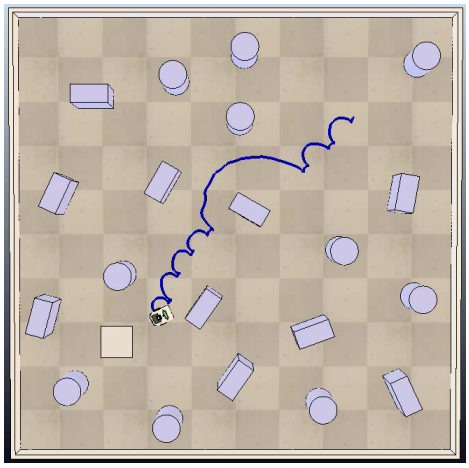


Figure 18 The best controller generated by the second evolution with fitness function 3 on map 1, $T_{sim} = 100\,\mathrm{s}$

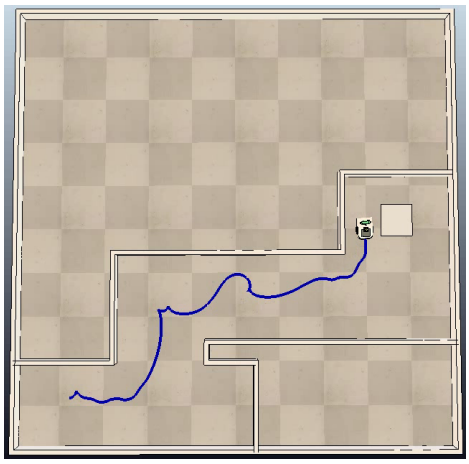


Figure 19 The best controller generated by the second evolution with fitness function 3 on map 2, $T_{sim} = 29\,\mathrm{s}$

obstacle. The penalization was calculated as

$$O_i = \sum_{j=1}^{N_i} o_j,$$

where $o_j = 0.001$ if there is an obstacle in less than 0.45 m from robot, $o_j = 0.002$ if the distance is less than 0.3 m and $o_j = 0.004$ if it's less than 0.2 m. The goal was to maximize the result.

The assumption was that the component of fitness function $\mathrm{d}S_i$, which is dependent on the velocity of the robot, would make the evolution to reward faster individuals. The penalization O_i would prevent the individuals which collide with obstacles from gaining good scores. As a result fast algorithm which avoid obstacles would evolve.

Two evolutions with parameters $S = 20$, $P_{mut} = 0.01$, $T_{sim} = 30\,\mathrm{s}$, $N_{gen\,1} = 50$ and $N_{gen\,2} = 74$ were tested. For both evolutions set of productions $P = \{P_1 \cup P_2\}$ and inputs in_1–in_9 were used. The development of the best scores is shown in Fig. 20 for the first evolution and in Fig. 21 for the second one.

The best individuals weren't able to avoid collisions, neither they moved to the goal, as it is seen in figures 22–25. The first evolution (figures 22 and 23) generated a

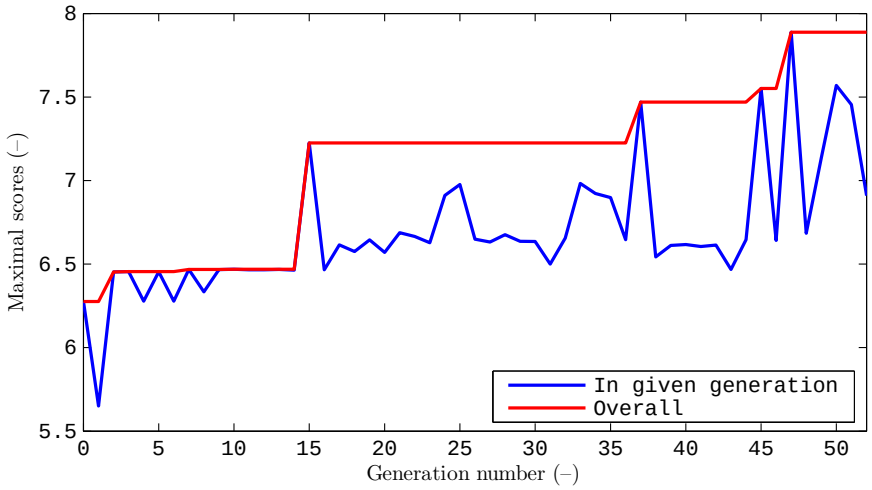


Figure 20 The development of the best score for the first evolution with fitness function 4.

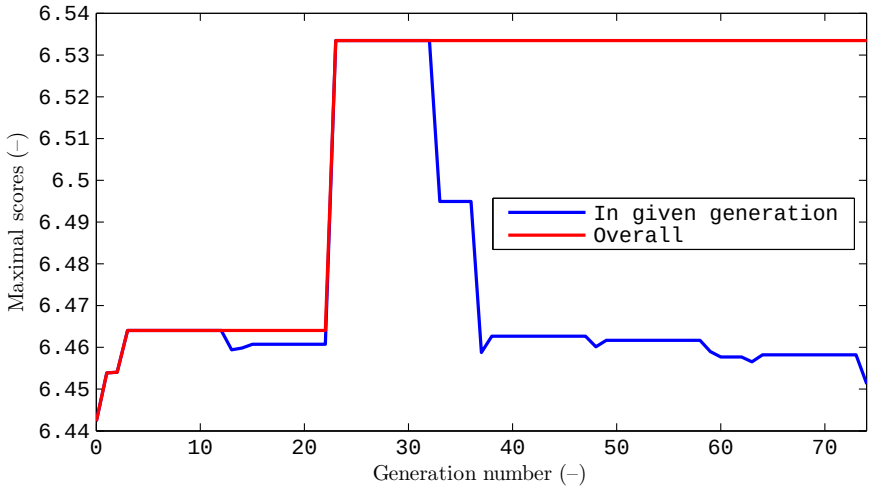


Figure 21 The development of the best score for the second evolution with fitness function 4.

controller, which moved fast and after 2–3 seconds of the random changing of the direction flipped over. The result of the second evolution (figures 24 and 25) was similar. The robot went straight and after 5–6 seconds of the movement hit against the obstacle.

In both cases evolution was generating individuals, which velocity and behaviour lead to a collision. Although the first evolution was making a progress, as it's seen in Fig. 20, it was interrupted. The reason is that the only difference between best individuals was their velocities and as a result time of being near obstacles.

The results of these experiments showed, that evolutionary algorithms have complications with optimising of several contradicting functions. Fitness functions 1, 2 and 4 had components rewarding the robot for moving towards the goal and parts punishing it for coming close to obstacles. Fitness function 3 only rewarded the robot for approaching the target. The best controllers of the evolutions with fitness function 1 stayed near the start position. Individuals tested with functions 2 and 4 haven't shown any signs of obstacle avoidance behaviour, neither they have the tendency to approach the goal. In contrast to that, evolutions using fitness function 3 have found algorithms leading the robot towards the goal. One of them found the solution to avoid the collisions with

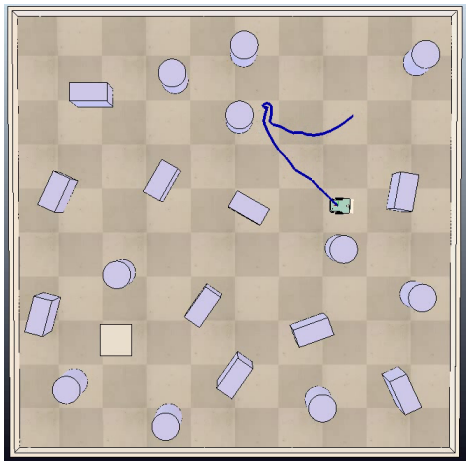


Figure 22 The best controller generated by the first evolution with fitness function 4 on map 1, $T_{sim} = 2\,\mathrm{s}$

Figure 23 The best controller generated by the first evolution with fitness function 4 on map 2, $T_{sim} = 3\,\mathrm{s}$

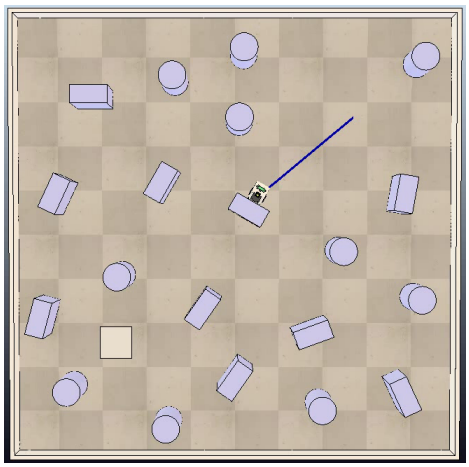


Figure 24 The best controller generated by the first evolution with fitness function 4 on map 1, $T_{sim} = 5\,\mathrm{s}$

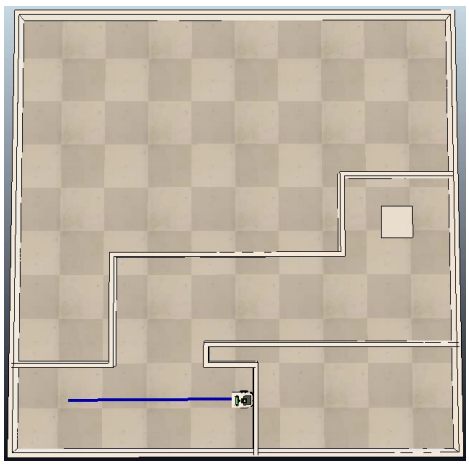


Figure 25 The best controller generated by the first evolution with fitness function 4 on map 2, $T_{sim} = 6\,\mathrm{s}$

the obstacles.

The best results were obtained by evolutions with more specialized fitness function. Therefore the next controller was assembled from the results of several evolutions with different less general goals.

4.4 Structure-restricted controller

The structure-restricted controller was created to examine how the separation of the goals would affect the outcome of the evolution. For this purpose the production S → ifGrDelay R R R R R was created. The operation coded in the terminal ifGrDelay is an if-then-else test with five inputs. If first input value is greater, than the second, the robot movement is controlled from the third input. Otherwise, the robot is controlled from the fifth input for the period specified in the fourth input. The sub-trees below the first two non-terminals are set by hand. The first input is the distance to the nearest obstacle in front of the robot, i.e. the minimal value in the central third of vector in

Figure 26 Maps used to the evolutions with the goal reaching task.

in_{12}. The second input is a constant 0.5 m. Remaining inputs are found by GP. The sub-tree under the third non-terminal was generated by evolution with the goal reaching task. The sub-trees below the fourth and the fifth non-terminals were evolving in an evolution with obstacle avoidance goal.

4.4.1 The goal reaching task

Evolutions for the goal reaching task took place on the specific maps with the specific fitness functions. The set of productions used for these evolutions was $P = \{P_1\}$, the set of terminals was $T = \{in_1, \ldots, in_6\}$. Generated algorithms were tested on four maps shown in Fig. 26 with the duration of the simulation $T_{sim} = 20\,\text{s}$. The goal was placed in different directions from the robot, so it could learn to turn towards the goal from any position.

Several evolutions with different parameters took place. Statistics are not available for these evolutions, as they were made in the beginning of the work on this thesis, when this option wasn't yet implemented.

The fitness function was calculated as

$$F = \frac{1}{1 + \sum_{i=1}^{n} (D_i)}, \tag{5}$$

where $n = 4$ is the number of maps on which the algorithm was tested and D_i is the final distance from the goal.

The evolution, that found the best controller, had the parameters $S = 15$, $N_{gen} = 26$, $P_{mut} = 0.03$, $P_{cross} = 0.9$ and $T_{sim} = 20\,\text{s}$. The initial population was created using the "grow" method with the maximum tree depth $d_{max} = 5$. The results of the generated controller are shown in Fig. 27.

4.4.2 The obstacle avoiding task

The evolution of structure-restricted controllers for obstacle avoidance had the following parameters: $S = 24$, $N_{gen} = 200$, $P_{mut} = 0.03$, $P_{cross} = 0.9$ and $T_{sim} = 20\,\text{s}$. The

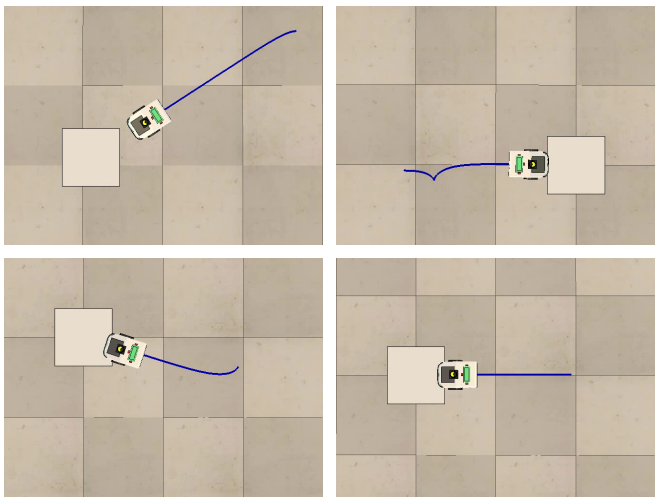


Figure 27 The best result of the goal reaching controller evolution. $T_{sim} = 30\,\mathrm{s}$

evolution could use the whole set of productions P and the whole set of the terminals T. Every created algorithm was tested on $n = 2$ maps shown in Fig. 6.

Controllers were created with a predefined part. As it was stated above, production S → ifGrDelay R_1 R_2 R_3 R_4 R_5 was applied to the start symbol. The sub-tree returning the minimal distance in front of the robot was below the non-terminal R_1. The constant 0.5 m was below the non-terminal R_2. The production R_3 → double R_6 was applied to the non-terminal R_3. The algorithm generated by evolution described in the previous subsection was placed below the non-terminal R_6. The initial population was created by the rules of the "ramped half-and-half" technique ignoring everything below non-terminals R_1–R_3. The minimum and the maximum tree depths were set as $d_{min} = 3$ and $d_{max} = 6$. The crossover and the mutation operations were set to ignore the part of the controller below non-terminals R_1–R_3. Described steps were made to ensure, that the predefined part would not change in the process of evolution. As a consequence the evolution may focus on the sub-trees below non-terminals R_4 and R_5, which is the collision avoidance part.

The following fitness function was used in the evolution of the structure-restricted obstacle-avoiding algorithm:

$$F = \sum_{i=1}^{n} G_i + 5 \times \left(1 - \frac{D_i}{D_{max}}\right) - O_i, \tag{6}$$

where $G_i = 1$ if the robot reached the goal and $G_i = 0$ otherwise. D_i is the final distance from the goal, $D_{i\,max}$ is the maximal possible distance from the goal on the map i and O_i is the penalization for approaching an obstacle. The penalization is calculated as

$$O_i = \sum_{j=1}^{N_i} o_j,$$

where N_i is the number of measurements, $o_j = 0.006$ if the distance between the robot and the nearest obstacle is less than 0.2 m and $o_j = 0$ otherwise. The goal was to maximize the fitness scores.

The development of the best score is shown in Fig. 28. The controller with the best fitness score $F = 10.18$ was created in the 157 generation. Its result is shown in Fig. 29. As it's seen, the robot collided with the obstacle after 17 s of the simulation.

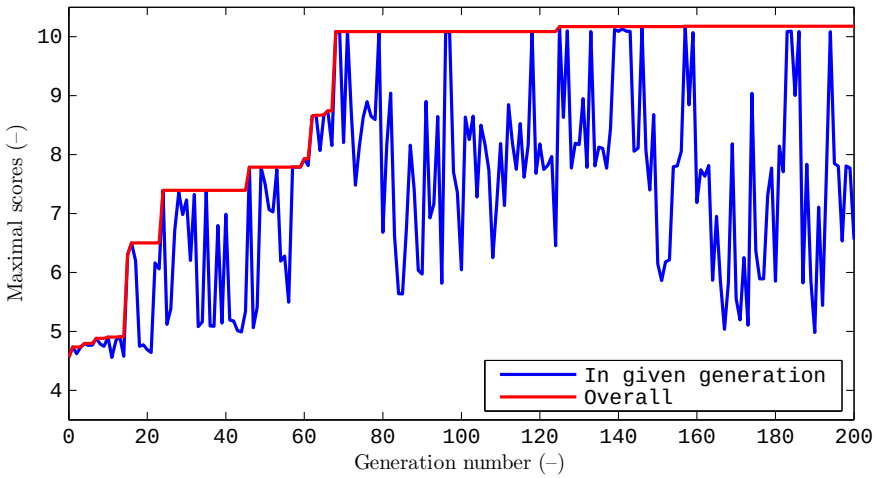


Figure 28 The development of the best score for the evolution of the structure-restricted controller.

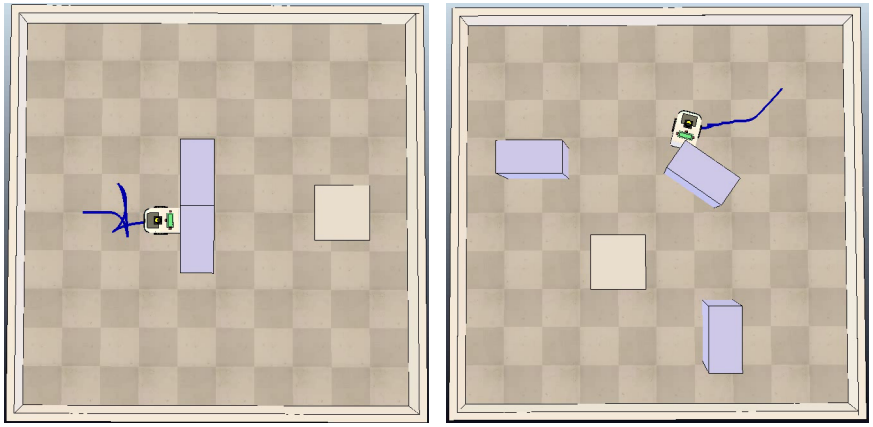


Figure 29 The result of the structure-restricted controller with the best fitness score. $T_{sim} = 17\,\mathrm{s}$

This happened due to the goal reaching part of the algorithm. The controller may lead the robot towards the goal both normally and backwards. In this case, it led the robot backwards. The controller achieved the best scores, because the penalization was working only if an obstacle was in the range of laser sensor. In this case, the collision wasn't even detected.

The other controller with good fitness scores created by this evolution developed in the 68 generation. It had the fitness score $F = 10.09$, which isn't much smaller, than the score of the best individual. However, the result of this controller shown in Fig. 30 was better. This controller managed to approach the goal before the end of the simulation and not to collide with any obstacle. The smaller fitness score may be caused by a penalization, which the robot could get while passing by the obstacle. Even though the controller shown good results on the learning maps, it failed to repeat the success on other maps it was tested on, as is shown in Fig. 31. The behaviour was similar to the behaviour of its offspring on learning maps.

The results show, that the evolution with the structure-restricted programs may develop a controller capable of avoiding obstacles and reaching the goal. The disadvantage of this approach is that the outcome of the whole controller depends strongly on the

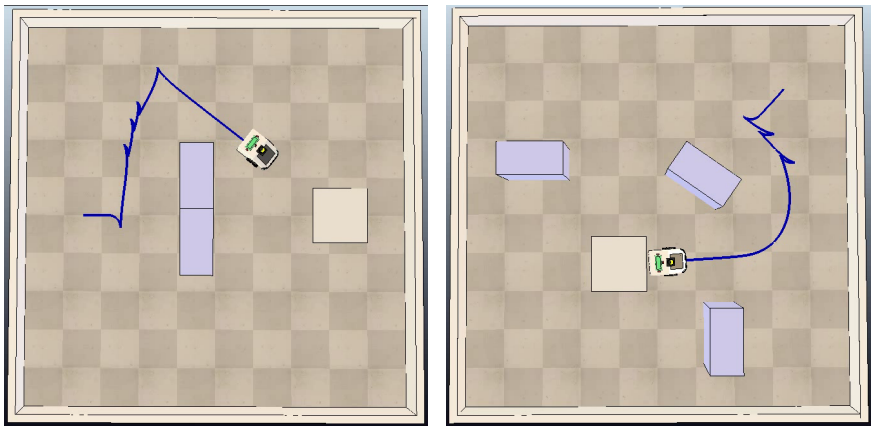


Figure 30 The result of the best structure-restricted controller. $T_{sim} = 30\,\mathrm{s}$

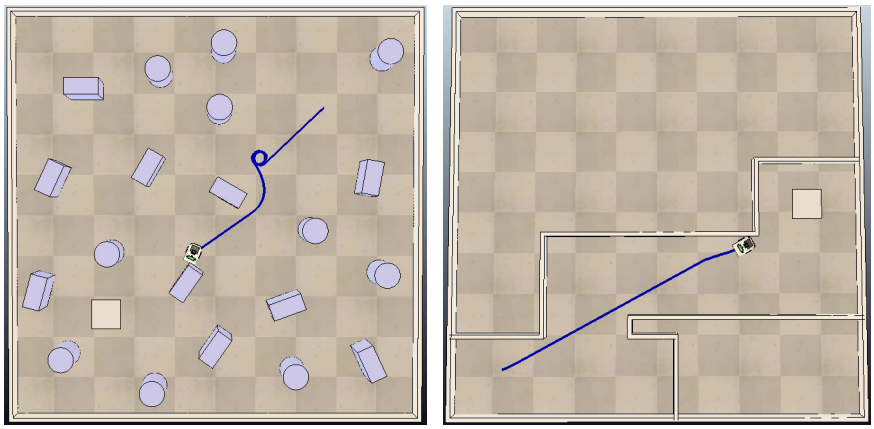


Figure 31 The result of the best structure-restricted controller on testing map. $T_{sim} = 17\,\mathrm{s}$

functionality of all its parts, particularly the ones created separately.

5 Conclusion

In this thesis the usage of evolutionary methods for creation of mobile robot controllers with collision avoidance behaviour was presented. The work was focused on controllers solving both goal reaching and collision avoidance task.

Firstly the context free grammar often used in the genetic programming for the representation of programs was described. Elements and construction rules used to form programs for the particular robot and the particular task were offered afterwards.

Several evolutions were performed to examine how different fitness functions affect the outcome of evolution of collision avoiding structure-free controllers. The results showed, that the best controllers are obtained by evolutions with the least complicated fitness function, which is focused on the satisfaction of only one goal.

Further, the evolution of the structure-restricted controller took place to examine how the controller composed of several independently evolved parts would perform. The best generated controller showed good results on the learning map, but failed to repeat them in other environments.

In my experiments, structure-free controllers have shown better results, than the structure-restricted controller. But that may not be the general rule, since many factors could affect the result.

In this thesis few of many possible fitness functions were represented, leaving many other to explore and to analyse. The future work may be focused on the improvement of the quality of evolving controllers by performing longer evolutions with larger population sizes, or using more complicated types of genetic programming.

Bibliography

[1] M.A. Jaradat, M.N. BaniSalim, and F.H. Awad. "Autonomous navigation robot for landmine detection applications". In: *Mechatronics and its Applications (ISMA), 2012 8th International Symposium on*. Apr. 2012, pp. 1–5. DOI: 10.1109/ISMA.2012.6215189.

[2] M. Takahashi et al. "A mobile robot for transport applications in hospital domain with safe human detection algorithm". In: *Robotics and Biomimetics (ROBIO), 2009 IEEE International Conference on*. Dec. 2009, pp. 1543–1548. DOI: 10.1109/ROBIO.2009.5420402.

[3] H. Sahin and L. Guvenc. "Household robotics: autonomous devices for vacuuming and lawn mowing [Applications of control]". In: *Control Systems, IEEE* 27.2 (Apr. 2007), pp. 20–96. ISSN: 1066-033X. DOI: 10.1109/MCS.2007.338262.

[4] Amir Hosseinzadeh and Habib Izadkhah. "Evolutionary Approach for Mobile Robot Path Planning". In: *in Complex environment" in IJCSI Vol. 7, Issue 4, No 8*. 2010.

[5] Peter A. Whigham and Department Of Computer Science. *Grammatically-based Genetic Programming*. 1995.

[6] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0-262-11170-5.

[7] Matthew Wall. *GAlib Documentation*. May 1996. URL: http://lancet.mit.edu/galib-2.4/API.html (visited on 05/2014).

[8] *V-REP simulator*. URL: http://www.coppeliarobotics.com (visited on 05/2014).

[9] M. Freese E. Rohmer S. P. N. Singh. "V-REP: a Versatile and Scalable Robot Simulation Framework". In: *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*. 2013.

[10] *Bullet physics library*. URL: http://bulletphysics.org (visited on 05/2014).

[11] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009. URL: http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf (visited on 05/2014).

[12] *Robot Operating System (ROS)*. URL: http://www.ros.org/ (visited on 05/2014).