elektrorevue
ISSN 1213 - 1539

Rok / Year:	Svazek / Volume:	Číslo / Number:	Jazyk / Language
2019	21	1	EN

Genetic Programming for Source Code Generation to Solve NP-hard Problems

-

Yehor Safonov, Martin Rajnoha

yehor.safonov@gmail.com

Faculty of Electrical Engineering and Communication Brno University of Technology

DOI: -

Abstract: This paper describes the usage of genetic programming method for source code generation with motivation to solve NP-hard problems. Described approach may be used in a wide range of modern applications, whose working principle allows to apply optimization techniques. Proposed method was used to find a potential solution for achieving maximal score while playing a computer game called "Robocode tanks". The main principle of the experiment is based on applying classical evolution approaches on the selected problem in order to implement adaptive machine learning technique. During the training process of presented approach convergence starts and after several cycles of evolution, created tank achieved significantly better final score compared to using a classic programming approach.

elektrorevue
ISSN 1213 - 1539

Genetic Programming for Source Code Generation to Solve NP-hard Problems

Yehor Safonov, Martin Rajnoha

The Faculty of Electrical Engineering at Brno University of Technology, Czech Republic
Email: yehor.safonov@gmail.com

Abstract *– This paper describes the usage of genetic programming method for source code generation with motivation to solve NP-hard problems. Described approach may be used in a wide range of modern applications, whose working principle allows to apply optimization techniques. Proposed method was used to find a potential solution for achieving maximal score while playing a computer game called "Robocode tanks". The main principle of the experiment is based on applying classical evolution approaches on the selected problem in order to implement adaptive machine learning technique. During the training process of presented approach convergence starts and after several cycles of evolution, created tank achieved significantly better final score compared to using a classic programming approach.*

1 Introduction

The modern world changes extremely fast and its progress is possible to describe by an exponential evolution curve [1]. It means that every second, every minute and every day, humankind is facing new inventions bringing our development process towards a brighter future, changing the perspective of the world understanding and making our being more comprehensive and balanced. Sometimes there appear new ground-breaking algorithms and approaches, which allow solving routine problems and tasks in a more efficient way.

However, there exist problems, whose optimal solutions are exceedingly hard to find [2]. In other words, while applying a classical approaches to solve such a problem, the required amount of resources is extremely large and not acceptable by today's hardware or software limits. The complexity of these problems is possible to describe as an exponential or even factorial function, which reflects a correlation between the size of the input and the number of steps required to solve an instance of the problem [1]. In computational complexity theory, the previously mentioned problems, depending on their time or space complexity, are classified into several complexity classes, e.g. NP, NPC and NP-hard. The hardest ones are solved in exponential time using deterministic Turing machine [1]. It means that in a case of the relatively small input size, the process of finding an optimal solution would take decades or centuries. Special algorithms called approximation algorithms can be used for these cases. They work in a polynomial time and allow to find a well approximating, i.e suboptimal solution for hard, i.e. exponential time, problems [3]. Evolutionary algorithms, that are a part of the artificial intelligence, are commonly used for finding previously mentioned suboptimal solutions [2].

Genetic programming is a widely known optimization technique inspired by Charles Darwin's theory of evolution [1, 2, 4]. This technique together with genetic algorithms, evolution strategies and neuroevolution belong to evolution algorithms [1]. It is possible to describe the entire concept of evolution algorithms using Charles Darwin's famous quote: "*It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change*" [5]. In other words, the line above describes the classic evolution process, i.e. natural selection, where only an individual with better set of characteristics has better chances for survival [4, 6].

The optimization technique is processed iteratively and during its execution, there exist a set of possible solutions, which are analysed during the same period of time [1, 7]. As a result, this condition helps to find the solution, which is as close as possible to the optimal one [1]. The whole set of analysed solutions is called a population P and it includes a fixed number of individuals x_i during one step of the evolution process:

$$P = \{x_1, x_2, x_3, x_4, \ldots, x_n\}.$$

The set of operations is defined over the population and it includes mechanisms allowing realisation of biological evolution, e.g. selection s, mutation m, reproduction d and crossover c:

$$EA = (P, s, m, d, c, f).$$

Furthermore, there exists a fitness function f, which specifies how close the given solution is to the optimal one. The function f is the basis of evolutionary computing algorithms, because it allows to predict the success factor of each individual in the population without knowing its low-level implementation details [9]. The effectiveness of evolution process will be ultimately minimized in case the fitness function is not implemented properly [1, 8, 9].

Each individual x_i in the population is called a chromosome. It includes a set of parameters called genes [6].

elektrorevue
ISSN 1213 - 1539

They define its unique properties that means it is possible to define a mapping function, which maps chromosome's parameters to the problem solving parameters and this mapping is strictly defined before the evolution process starts. While processing one iteration of the evolution, chromosomes are mutated, selected, recombined and crossovered with each other [6]. Individuals with better parameters are chosen and brought to the next generation at the end of each iteration. It provides guaranteed information distribution between nearby populations, i.e. knowledge extracted from previous generation influences the new solutions in next generation [1, 6, 8].

For a better understanding of the discussed topic, the high-level description of a classic genetic technique is presented below in pseudocode [1]:

```
begin
    P ← initialize a default population;
    n ← initialize a population size;
    f ← initialize a fitness function;
    while suboptimal solution not found do
        X_(u,v) ← SelectPair(P);
        X' ← Crossover(X);
        P' ← P ∪ X';
        P'' ← Mutate(P');
        f ← Fitness(P'');
        P ← SelectFittest(P'', n);
    end
end
```

Previously mentioned techniques may be successfully applied on a wide range of NP-hard problems, whose final solutions are not possible to find in acceptable time $O(n)$ using linear or deterministic programming methods. The reason is that problems have a lot of possible solutions and there does not exist or is unknown any straightforward method which narrows problem's solution space including only the most promising ones. Today, genetic algorithms are highly used in solving problems such as data mining, process planning, machine learning, artificial intelligent and combinatorial optimization. [1]

2 Experiment

The experiment was focused on applying and comparing both genetic algorithms and genetic programming for source code generation with the goal to achieve the highest possible score while playing Robocode tanks [10]. Compared approaches are capable of finding a suboptimal solution in polynomial time. Both techniques have similar way of realizing the high-level principle of the evolution, but the main difference is in data structures that are used to encode the information in each chromosome. As it was mentioned in the Section 1, there exists a special mapping function, which defines chromosome's information about solution and chromosome's data structures to represent a possible solution. Every chromosome stores its information about solution using a list (vector) data structure in the case of genetic algorithms [8]. Genetic programming uses a tree based data structure at the core of each chromosome thus it achieves stronger expressive power and it allows to shape more complicated tank's behaviour. Nevertheless, the genetic algorithm approach was also implemented and tested to show all advantages and disadvantages described techniques (score comparison of both techniques is in Fig. 4 in the Section 3). It leads to a better understanding of both techniques that are viable and their usage makes sense in the specific cases.

2.1 Genetic programming

Genetic programming approach was chosen as a more perspective way for source code generation, because of its effectiveness and expressive power. This approach can be principally used for creating computer programs, which are directly encoded in artificial chromosomes, i.e. genetic programming creates other computer programs, which are able to solve a predefined problem [11]. Each program encoded in chromosomes is executed and evaluated by fitness function f during the evolution. It is important to note, that the genetic programming approach is based on the previously described evolution principles, and thousands of chromosomes are progressively evolved during the algorithm's execution [11]. The evolution process starts with creating an initial population containing randomly generated or manually adjusted chromosome's skeletons [11]. The fittest chromosome according to fitness function represents an artificially generated program, which transforms input parameters to the required output in the most effective way among other chromosomes in the population.

Tree based data structure used in genetic programming significantly influences the success factor of the evolution process, but unfortunately increases complexity to compute possible solutions [1, 11]. The tree is generally characterized in graph theory as an undirected connected graph without cycle[1]. There are plenty different types of trees in mathematics, but in the case of genetic programming, the direct rooted trees are mainly used. In rooted trees exists a vertex called a root, which has no parent and all its edges point away. The existence of a tree root allows easy tree traversal, which is equivalent to the execution of chromosome's encoded program with the motivation to apply its encoded solution to the problem and evaluate it.

Following the definition of genetic programming [11], chromosome's tree is formed by two different types of nodes (terminal, function), which are appropriate for the problem domain. The first type of node is called a terminal and it does not have any outgoing edge. For example, program which represents a mathematical equation – the set of terminals may include number constants and unknown input parameters: $T = \{7, 15, 13, \pi, x, y\}$ [1].

[1]Equivalently, tree structures may be described as an undirected and acyclic connected graph.

elektrorevue
ISSN 1213 - 1539

The second type of node is called a function and it contains at least one incoming and one outgoing edge. Applying to the previously described example, the set of functions may contain a various number of arithmetic operations and mathematical functions: $F = \{+, -, *, \div, \sin, \cos, \mathrm{mod}\}$ [1]. Furthermore, there exist additional requirements, which should be met during designing of a tree structure:

1. The root may be either the function or the terminal.
2. The terminal node does not have any children.
3. Every leaf is the terminal node.
4. The function may contain both types of nodes.

The way of source code representation introduced above allows to deal with more complicated problems and provides mechanisms to create the fittest computer programs. The disadvantage comes with using general data structures which are linked to the complexity of the chromosome's models undergoing adaptation [11]. In particular, their varying hierarchical structures could take different sizes and shapes, which may negatively influence the total complexity of the algorithm[2]. More detailed information about the genetic programming approach is available in sources [1, 11].

2.2 Robocode game

Robocode tanks is a computer game, which is similar to the classic and widely known game "Tanks". The story of Robocode begins in late 2000, when the game was presented by an experienced programmer – Matthew A. Nelson. The main purpose of the game was to teach young programmers how to write a code in Java language, understand its programming principles and at the same time do it in a fun and easy way. The Robocode game has become very popular and today it has a big community of developers and players. Since then, there have been many different features released, which have improved the game experience and allowed to use it even for scientific purposes especially for applying different artificial intelligence approaches and comparing theirs effectiveness to humans.

The game contains a battlefield with several tanks that fight each other and fight continues until one of the tanks lefts. The main difference between the games is that Robocode game provides indirect tank control. It means that nobody is able to control his own robot in real time. Instead of that, the user should come up with a tank's behaviour and code it using a Java language. After writing an artificially intelligent robot, the user may compile it and let him out in the battlefield to achieve the best score.

The tank's life cycle includes four main stages: programming the tank, code compilation, battle and death. Before being tested genetic programming approach on a real battle, each tank should meet certain requirements. The tank should inherit from a parent `Robot` class. Extending the parent class obligates a programmer to override all necessary functions in order to be compiled and accepted by a game engine. Next requirement comes from Java language that defines certain syntax rules, variables declaration, function's syntax, arguments, visibility, etc.

Every tank in the Robocode game has a precisely defined structure. Its class should contain the main `run()` function and may override different types of listener functions, such as `onHitRobot()`, `onHitWall()`, `onScannedRobot()`, etc. The `run()` function is profoundly important, because it defines tank's primary behaviour, when none of existing listeners catches any traceable event. A set of listeners inside a tank's class represents actions, which may occur while the game playing. For example an event, when the tank has scanned an enemy unit or has hit a wall.

2.3 Applying proposed method on the problem

While designing the genetic programming application it is important to follow the previously described tank structure. During the first steps of the application development all significant listeners and the `run()` function were defined as separate Java classes, which inherit from a generalized `Listener` class. This approach allows to simplify every listener, avoid code duplicity and defines polymorphic functions, i.e. `addSupportedEvents()` function and `getEventType()` function. The high-level example of the chromosome tank representation in the population is presented in the Figure 1.

It is important to understand that at the beginning of the tank designing it is necessary to generate a whole tank's Java class skeleton in a text (string) form. A syntax of the created tank's model and its logic is checked after the code generation step. If there is no mistake, the robot's class is prepared for the compilation. Overriding of `toString()` method is used for this purpose supposing of return the logically and syntactically correct textual representation of the function. This function should have polymorphic features, because of different Java structures should be written in a different form, i.e function definition, `for`, `while` cycle, `if` condition, etc.

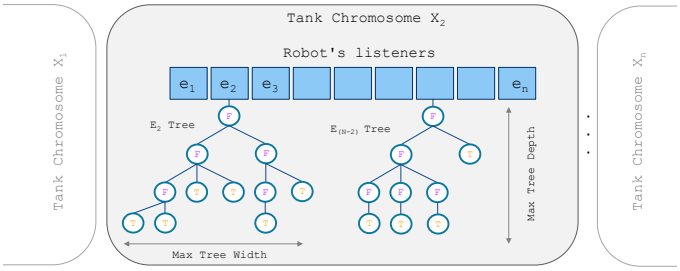

Figure 1: A chromosome representation in the population.

[2]For the successful algorithm execution it is important to provide operations, which allow to run tree traversal, to search concrete node, to copy required subtree etc.

elektrorevue
ISSN 1213 - 1539

Each chromosome x_i in a population is stored in a `PriorityQueue` data structure, which is based on a priority heap. Tanks are automatically compared and sorted by its fitness value in descent order. Every chromosome x_i contains a list of listeners, where each one has its own type of the triggered event e_i. The main `run()` function is placed at the end of the list.

Each listener within chromosome should contain its own behavioural rooted tree in the case of the Robocode game. The code structure and logic of the tank does not allow to generate one big tree, which contains all listeners and carries out complete information about all events. Furthermore, all listeners contain common inherited events, some of them define specific ones, which are not compatible with others. Individual trees are shown in the Figure 1. All trees are horizontally and vertically limited because of a good practice to fix tree's dimensions. The reason is that during initialization and mutation steps, individual sub trees are generated fully randomly. It can caused an enormously huge tree, which negatively affects both the training process and the efficiency of the tank actions.

Each tree contains a set of terminals and a set of functions, which are hierarchically arranged based on the rules described in the Section 2.1. The set of terminals includes possible tank actions, for example `ahead()`, `back()`, `fire()`, `turnGunLeft()`, `turnRight()`, etc. The set of functions consists of different events, which may occur in the tank's listeners. Typical examples of tree's functions are `e.getVelocity()`, `e.getTime()`, `e.getPower()`, `e.getDistance()`, etc. It is important to point out the reason why tracing event attributes is pertinent. The advantage of processing them helps to design more sophisticated and intelligent tank behaviour using logical `if` conditions. For example, the tank can analyse its health, power and damages or the tank can make decisions based on direction and distance to the enemies, i.e. it allows the tank to adjust its behaviour according to environmental changes.

The events (functions) should be as common regardless of listener type as specific for individual listeners. Actions (terminals) have their distinguishing input arguments or do not have any argument at all – for example an action `doNothing()`. Moreover, some of the terminals accept only a specific range of input values. Two described complications have reflected the design of the genetic programming application, where both events and actions are defined as separate Java classes. Each class contains information about defined limits, acceptable type of variable and polymorphic functions.

Every chromosome x_i has methods to clone itself, mutate its skeleton and the set of operations includes `compareTo()` and `toString()` functions. The first one provides a way how to compare[3] chromosomes for storing them in `PriorityQueue`. The `toString()` method allows easily and independently on its low-level implementation details to get a string representation of the robot class before the compilation process starts. The method works in a simple way – there exists a single loop over a list of robot's listeners. Each listener also contains polymorphic `toString()` method which is recursively called over nodes in a single tree.

The function `mutate()` is responsible for providing mutation operation. The function is implemented within a `Chromosome` class. A principle of the mutation process is shown in the Figure 2.

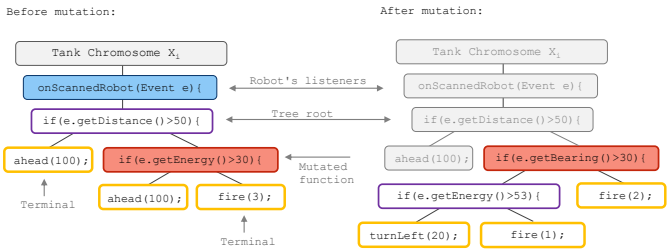

Figure 2: An example of the tank mutation process.

The chromosome x_i is randomly chosen from the population at the beginning of the mutation process. Next, one of the chromosome's listeners is selected. Referring to the previously mentioned examples, each listener contains a tree which includes distinctive terminals and functions. The `mutate()` function operates inside the listener's tree and randomly chooses either terminal or function node. According to the presented example (Figure 2), a function node containing the event `e.getEnergy()` is selected for the following mutation process. Next, the selected individual will be removed and replaced by a newly generated node or even a subtree (see Figure 2, the step after mutation). The mutated chromosome is returned to the population at the end of the mutation process and the evolution cycle continues.

The `Population` class contains two functions which are important in genetic programming – `crossover()` and `fitness()` function. The `crossover()` function is used to interbreed two different chromosomes, i.e. robots inside one population. The crossover process implemented in the application is presented in the Figure 3. The crossover operation is fundamentally important for the successful evolution process – it allows to transfer a knowledge (genes) between two individuals (x_i and x_j chromosomes), thereby affecting their future behaviour. It is important to notice, that the crossover operation makes sense only between two similar tank's listeners for providing event agreement. The crossover happens between the function node in the x_i chromosome and the terminal node in the x_j chromosome in the presented situation. The crossover operation may be finished in a destructive way in some situations – two pretty successful individuals crossover their trained subtrees that causes destroying of their success. The question is, how to choose two chromosomes in a smart way to minimize the probability of a negative impact on each other.

[3]The value used for the comparative purposes is given by an implemented fitness function.

elektrorevue
ISSN 1213 - 1539

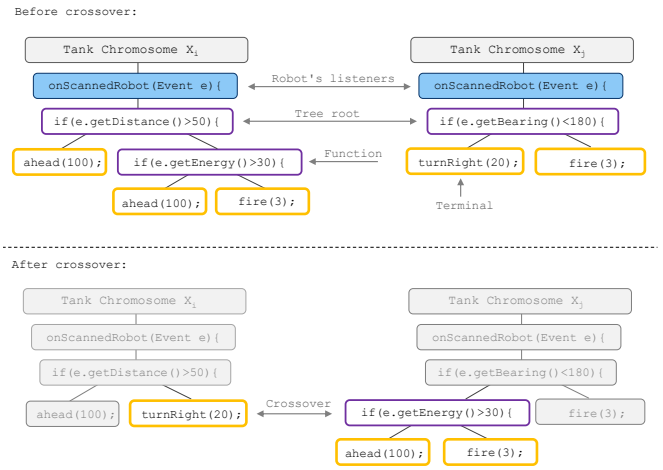

Figure 3: An example of the crossover process between two tanks.

The solution applied to the proposed method consists of choosing only the fittest tanks for the crossover.

In conclusion, in order to evaluate Robocode tanks cleverness the `FitnessFunction` class was implemented. The `fitness()` function returns an estimated real number reflecting robot's success factor (see the Section 1). The mentioned function is based on existing score mechanism directly provided by the Robocode game. Every tank gets its final score at the end of the battle, which directly reflects the tank's performance. Each tank gets positive or negative points depending on the number of victims, robot's remaining energy, health, damage done, etc. The final score is tank's attribute, which is next normalized regarding the population size and the number of accomplished battles. The result of the `fitness()` function is assigned to the chromosome and it is used for the following sorting in the population's `PriorityQueue`.

3 Results and Discussion

This part clears up the tuning and testing processes during developing of the proposed genetic programming application with the motivation to achieve a better result and make the evolution process more efficient. In order to provide better understanding of the applied genetic principle, the source code of the artificially generated tank is presented (see the Section 3.3). After that the focus is moved on the direct comparison between genetic algorithms and genetic programming (see Figure 4 in the Section 3.3). Finally, there is a comparison between the proposed genetic programming approach and the code written by a human (see Figure 5 in the Section 3.3).

3.1 Evolution process

During the execution of the genetic algorithm many steps are done randomly, so it is important to tune its parameters and optimize evolution process before the training starts [12, 13]. It means to set up the size of the population, to choose the amount of tanks for mutating, crossing, dropping, etc. Each tuned attribute may strongly influence the population's convergence. The most effective parameters, which were ascertained during testing and rerunning the algorithm were summarized into following points:

1. Each battle contains three rounds.
2. There are four tanks during one round.
3. The population of tanks contains 16 chromosomes.
4. Two best chromosomes (x_1 and x_2) automatically survive and are taken to the next evolution cycle.
5. Next four chromosomes randomly choose either x_1 or x_2 tanks and do the crossover operation with them.
6. Next two chromosomes are mutated.
7. Next two ones are crossovered between each other.
8. Other six tanks are removed and new tanks are generated to fit the population size.

3.2 Problems and complications

The first problem was related to the time required for simulating all robots inside one population. It was highly complicated to define the number of tanks during one battle and the number of rounds. In the case every tank fights with all others and the population size is relatively small, there was only one round and the time required for one evolution cycle exceeded 10 minutes. Finally, the population was divided into groups containing four robots and each battle includes three rounds. The amount of time has reduced to 80 seconds.

The second complication contributed to appearing the deviations during the evolution process. The reason was the general fitness function did not reflect a random position of the tank before every battle. It negatively influenced and complicated the final score computations. For example, a relatively good tank may appear surrounded by opponents that killed it in the moment independently on its source code. For avoiding a negative impact of the random position, it was necessary to simulate more battles, normalize a fitness function and follow a population median.

3.3 Results

The source code presented below directly reflects the proposed genetic application outcome, i.e. the code that has been automatically generated after realization of a crossover operation between two Robocode tanks regarding the Figure 3. There is a list of listeners at the beginning of the class and the run function closes Robot's class. As it was mentioned in the Section 2, all generated robots extend the common `Robot` class, which is directly provided by the Robocode game.

elektrorevue
ISSN 1213 - 1539

```
public class TankChromosome16 extends Robot {
    // Robot's event listeners
    public void onScannedRobot(ScannedRobotEvent e) {
        if (true) {
            if (e.getBearing() < 180) {
                if (e.getEnergy() > 30) {
                    ahead(100);
                    fire(3.0);
                }
                fire(3.0);
            }
        }
    }
    public void onHitWall(HitWallEvent e) {...}
    public void onHitRobot(HitRobotEvent e) {...}
    public void onBulletHit(BulletHitEvent e) {...}
    public void onHitByBullet(HitByBulletEvent e) {...}
    public void onBulletMissed(BulletMissedEvent e) {...}
    // Main run function
    public void run() {...}
}
```

The chart in the Figure 4 shows the difference between genetic programming and genetic algorithm that was also implemented and tested during the designing of the proposed method. It is noteworthy, that the genetic programming approach gets better final score after the 220th and it shows much better applicability on the Robocode game, because of its data structures that can express more complex chromosome's behaviour.

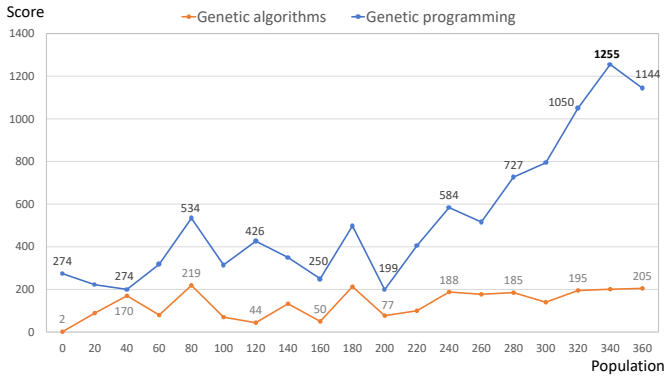

Figure 4: A comparison of genetic programming and genetic algorithm approaches.

Chart in the Figure 5 shows the difference between the genetic programming approach and a human. The automatically generated tank gets a much better score after 850 evolution cycles than the robot programmed by the human. Of course the reasonable question is, whether the evolution process should be continuing. Fortunately, a definitive answer does not exist. If the evolution process continues, two factors may affect the population's score growing. The first is the deviation, which has appeared during the 250th population and is described in the Section 3.2. The second one is related to the overstraining effect, which can occur in gradual fading of the population [1, 11].

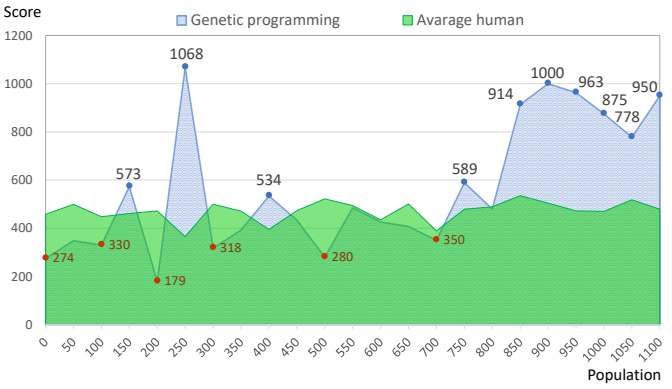

Figure 5: A comparison of genetic programming and tanks created by an avarage human.

4 Conclusion

Genetic programming is a very promising approach for solving NP-hard problems, where optimal solutions are not possible to find in real time, because of their exponential time complexityThis paper describes an overall method for source code generation using genetic programming technique. The proposed method was tested on the programming game Robocode with the motivation to generate the tank that defeats all opponents.

The paper summarizes a theoretical basics and includes a comprehensive approach to evolutionary algorithms. It clarifies the difference between genetic programming and genetic algorithm approaches, deals with basic methods and principles and presents an effective way of tuning genetic programming parameters. The experiment is directly focused on applying theoretically researched knowledge to the real NP-hard problem. The proposed method brings a complex solution for the source code generation using genetic programming technique. The advantage of the proposed method is the generality of its applications, functions and used data structures. It allows easy transformation of the invented method to solve other hard problems.

This work also brings a comparison of tested approaches and demonstrates the output of the method, i.e. founding a sub-optimal source code solution allowing the robot to achieve a better score as the human can. The demonstrated results evidently show the increasing trend in the robot's final score, which demonstrates the effectiveness of the proposed solution.

As a result of the comparison between genetic algorithms and genetic programming, the superiority of genetic programming is shown. The reason is that genetic programming brings more complicated expressive power using tree based data structures that allows to describe the behavior of entities in a more effective way.

In conclusion, the genetic programming approach is a great example of the approximation algorithm, which can be directly applied on a wide range of real-world problems that allow integration of optimization approaches.

elektrorevue ISSN 1213 - 1539

The main advantages of the proposed method are simplicity, generality and parallelization possibilities. Furthermore, the output of the genetic programming techniques returns a fully executable source code, which may be immediately processed.

Acknowledgment

For the research, infrastructure of the SIX Center was used. Authors also would like to show their appreciation to the group of students: Marek Mikulec, Pavel Mazanek, David Pecl, Lucie Popelova. They greatly helped with essentials for this paper within the Advanced Theoretical Informatics course supervised by doc. Radim Burget, Ph.D.

References

[1] BURGET, Radim. *Theoretical computer science*. Brno University of Technology, 2013. ISBN 9788021448971.

[2] PANCHAL, Gaurang and Devyani PANCHAL. *Solving NP hard Problems using Genetic Algorithm* [online]. Patel Department of Computer Engineering Chandubhai. Patel Institute of Technology, Changa, India, 2015 [cit. 2019-02-20]. ISSN 0975-9646. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.735.8064&rep=rep1&type=pdf>.

[3] RASHID BIN, Muhammad. *Algorithms* [online]. Department of Computer Science, Kent State University [cit. 2019-02-24]. Available: <http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html>.

[4] OBITKO, Marek. *Introduction to Genetic Algorithms* [online]. Hochschule fur Technik und Wirtschaft Dresden, 1998 [cit. 2019-02-20]. Available: <http://obitko.com/tutorials/genetic-algorithms/>.

[5] DARWIN, Charles. *The Origin of Species*. London, 1885.

[6] MALLAWAARACHCHI, Vijini. *Introduction to Genetic Algorithms* [online]. 2017 [cit. 2019-02-19]. Available: <https://towardsdatascience.com/>.

[7] KRAMER, Oliver. *Genetic algorithm essentials*. New York: Springer Berlin Heidelberg, 2017. ISBN 9783319521558.

[8] MITCHELL, Melanie. *An introduction to genetic algorithms*. Cambridge: Bradford Book, 1996. ISBN 0262133164.

[9] NELSON, Andrew, Gregory BARLOW and Lefteris DOITSIDIS. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems* [online]. 2009 [cit. 2019-02-21]. Available: <http://www.nelsonrobotics.org/paper_archive_nelson/nelson-jras-2009.pdf>.

[10] *Robocode* Robocode.sourceforge.io [online], [cit. 2019-02-01], Available: <https://robocode.sourceforge.io/>.

[11] KOZA, John. *Genetic programming and evolutionary computation* [online]. [cit. 2019-02-23]. Available: <http://www.genetic-programming.org/>.

[12] KOZA, John. *Genetic programming III: darwinian invention and problem solving*. 3. San Francisco: Morgan Kaufmann, 1999. ISBN 1558605436.

[13] LANGDON, William B. a Riccardo POLI. *Foundations of genetic programming*. New York: Springer, 2002. ISBN 3540424512.