Cooperation Evolved

Behavior Trees evolution by means of Genetic Programming

Miłosz Mazur

September 21, 2015

Abstract

Behavior Trees are a method for AI programming that consists of a tree of hierarchical nodes controlling the flow of agent's decision making. They have proven, while being a pretty straightforward means to implement an AI, to be incredibly powerful way of obtaining autonomous agents, both due to a fact that the development can be iterable (one can start with implementing simple behavior and gradually improve the tree by adding and modifying nodes and branches) and allowing for, so to say, "fallback tactics", should the currently executed action fail. Born in the game industry, they have since gained fair amount of popularity in other domains, including robotics. Evolutionary algorithms, largely popularized by John Holland [6], have been adapted for use in a vast variety of different problems, including optimization issues and decision handling, often through introducing serious changes to both the algorithm structure and data structures used. Arguably, one of the most valuable modifications was Genetic Programming, popularized through works of John Koza ([7]).

This thesis documents the work on combining Behavior Trees and Genetic Programming in order to study and observe cooperative and adversative behaviors between agents controlled by genetically generated Behavior Trees. Evolving two kinds of agents in two contrasting scenarios, this thesis focuses on *feasibility* of cooperation and its evolution.

Acknowledgements

Contents

List of Figures

List of Tables

1 Introduction

1.1 Thesis Goal

The main goal of the thesis is to answer the question: "How selfish and utilitarian behaviors between artificial agents influence the performance in multi-agent task-oriented environments?"

1.2 Thesis Research Goals

- Provide a platform that can be used to automatically generate and evaluate artificial agents.
- Measure and find a way to deal with the impact of the initial generation.
- Measure the effect cooperative behaviors had on task execution.

1.3 Challenge

The true challenge in presented work is to provide a valid way to compare two drastically different approaches to the task in order to prove whether changing said approach is tied to a change in efficiency of the task's execution. Before the grounds for the comparison can be defined, a common goal must be stated for agents in both approaches, thus formulating aforementioned task.

The technical challenge lies in handling the incorporation of Genetic Programming to the problem. Evolving Behavior Trees will be subjected to various transmutations, some of which could potentially destroy some of them. The task here is to minimize the occurrence of this happening.

1.4 Thesis Structure Overview

In order to provide the answer to a question stated in the thesis goal, following chapter will first provide essential background information on technologies used in the thesis. Chapter 3 will complement that information with recent developements and related work in appropriate areas. Chapter 4 will acquaint the reader with the platform the research was done on, as well as the map used in specimen evaluation, detailed description of research conditions, restraints put on the environment and actually combining the two

technologies described in previous chapters. The last chapter presents the details of experiments done as part of the thesis, as well as their results.

The attempt to answer the main research question of this thesis is then presented in conclusion, along with ideas for possible future research.

2 Background

2.1 Behavior Trees

2.1.1 Synopsis

Behavior Trees are Directed Acyclic Graphs that are used to represent an agent's decision process. From a syntax perspective, they're usually represented as a tree structure constructed from multiple nodes representing various internal constructs. Each node has a return status, commonly: Success, Failure or Running, used to determine a result (or lack of one, in case of Running status) of the node's execution [3]. In turn, said nodes' function is to either redirect a flow of a process or to represent a decision resulting from one. Consequently, we can derive three kinds of these nodes:

- **Action** nodes reside exclusively in leaves of a tree and are, as mentioned above, a resulting decision, a set of instructions for the agent to execute. The degree of granularity is arbitrary - one can imagine both *Move Forward by 1 meter* and *Perform Engine Repair* to be examples of Action nodes, with relevant code belonging to Agent's implementation. Alternatively, Action nodes might point to further decision processes: for instance, *Perform Engine Repair* could be a complicated procedure with separate Behavior Tree, detailing diagnostic and the actual repair.
- **Composite** nodes, however, are tasked with controlling how the tree is actually executed. For the purposes of this thesis, Selectors, Sequences, Parallel-All and Parallel-One types will be considered and reviewed. Note that although the ones are arguably the most widely used, many more Composite nodes types exist, fitting various use cases.
- **Condition** nodes are used to query a specific property of the environment, testing an internal condition and returning Success or Failure applicably - representing, in this case, True and False.

While selection of Action and Condition nodes are (often) entirely implementation-dependent, as they must be programmed to perform specific tasks (or query the environment for specific condition), most of the Composite nodes are not dependent on the platform and may be freely reused (with the possible exception in Parallel type nodes).

2.1.2 Composite Node Types

In order to remain relatively simple to work with while maintaining versatility, flow-control nodes are possible to employ. These come in many varieties and often can be

said to mimic (or at least resemble in action) logic functions.

- **Sequence** nodes will test its child nodes in defined order - executing them from left to right is the common practice, although introducing a different metric (for example, task priority) is possible [3]. Sequence node will return Success if and only if all of its children return Success, and Failure otherwise. In this, Sequence works identically to logical AND function.
- **Selector** node, for all intents and purposes, is a operational opposite of Sequence. Execution order remains unchanged (along with all modification possibilities), but Selector node will return Success immediately when one of its children return Success and Failure only when all of its children return Failure. Analogically, Selector is a Behavior Tree counterpart to logical OR function.

Finally, **Parallel-One** and **Parallel-All** nodes introduce concurrence to Behavioral Trees: execution of all child tasks is simultaneous, the node returning when any (in case of Parallel-One) task or all of them (Parallel-All) return Success.

2.1.3 Behavior Tree Example

Consider the task of getting to a room through the door. For this, a behavior tree depicted in Figure 2.1 was proposed (adapted from [12]):

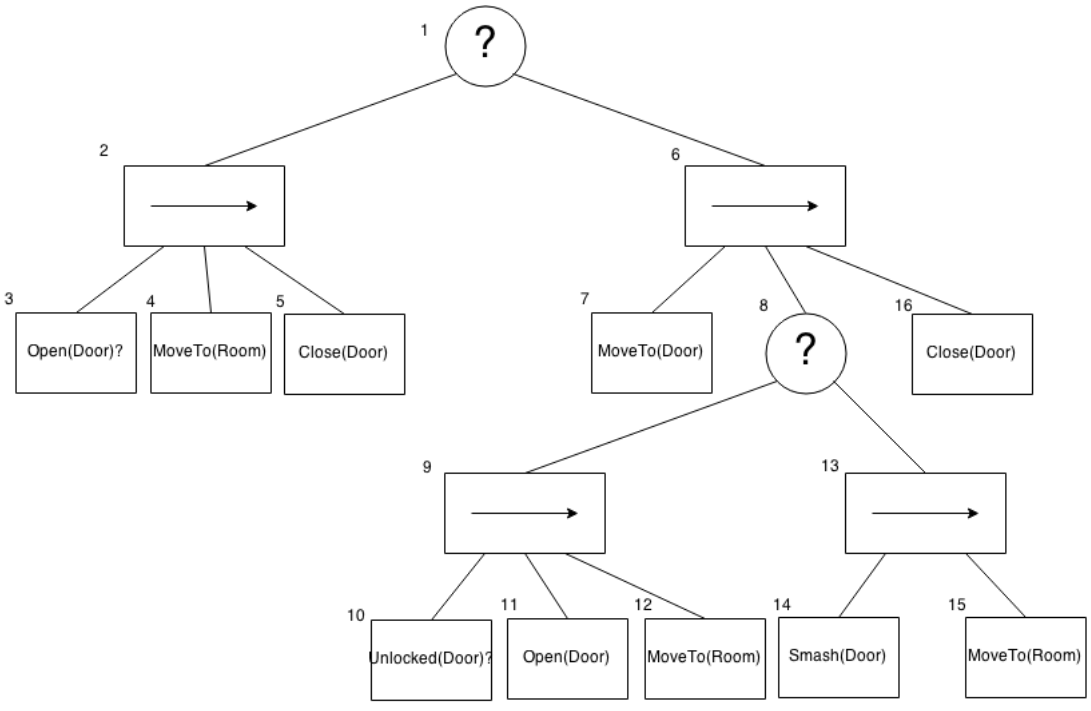


Figure 2.1: An example Behavior Tree

In the above tree consider:

- nodes 1 and 8 to be Selectors
- nodes 2, 6, 9 and 13 to be Sequences
- nodes 3 and 10 to be Conditions
- nodes 4, 5, 7, 11, 12, 14, 15 and 16 to be Actions

Both Action and Condition nodes here have parameters (Door and Room, which we consider constants denoting, respectably: door in question that we need to get through and a room behind it).

In case when the door is already open, Selector 1 will go to node 2 (Sequence) and, after verifying that the Door is open (Condition 3), will proceed to going through it and closing them afterwards (Actions 4 & 5).

In case when the door is closed, but not locked, Selector 1 will go to node 2 as well, but since Condition 3 will fail, next node evaluated will be Sequence 6. After moving to the door (Action 7) we first(Sequence 9) confirm that the door is, in fact, unlocked (Condition 10) and proceed to open it and move to the room (Actions 11 and 12). After returning from Sequence 9 (with Success) and Selector 8 (since 9 returned Success), it will proceed to closing the door (Action 16).

Finally, if the door is both closed AND locked, Selector 1 will go to node 2 first, but since Condition 3 will fail, next node evaluated will be Sequence 6. After moving to the door (Action 7) and finding out that the door is locked (Condition 10), we will go straight to Sequence 13, first smashing the door (Action 14), then moving to the room (Action 15) and eventually, after returning from nodes 13 and 8, closing the door (Action 16).

2.2 Genetic Algorithms

2.2.1 Motivation and History

Genetic Algorithms are a family of search algorithms directly inspired by Charles Darwin's theory of evolution. His influential book, "On the Origin of Species"(1859) led to a birth of the term "survival of the fittest" by Herbert Spencer ("Principles of Biology", 1864), who understood it as "(...) the preservation of favoured races in the struggle for life".

Over the years, many scientists proceeded to capitalize on that phrase, designing algorithms in which potential solutions' quality (*fitness*) was the deciding factor in deciding if they survive to the next round of choosing.

Genetic Algorithms' births and growth into a popular search heuristic was largely due to John Holland's "Adaptation in Natural and Artificial Systems". In his book, he proposed a method to apply an evolutionary process in nature to artificial systems by genetically breeding populations of fixed-length character strings (posing as solutions

to a given problem) using Darwinian natural selection model and genetic operation of recombination. He was also able to conclude the following about the model problem in the book (a multi-armed slot machine)[7]:

> (...) the genetic algorithm is a near optimal approach to adaptation in that it maximizes expected overall average payoff when the adaptive process is viewed as a multi-armed slot machine problem requiring an optimal allocation of future trials given currently available information.

In following years, and after introducing some modifications, the method evolved into a shape summarized below.

2.2.2 Synopsis

The Genetic Algorithm process starts with generating a population of specimen: candidate solutions to the problem. They are then graded using user-specified, problem-specific *fitness function*, and a fitness value is bound to each solution. Afterwards, the process of creating a next generations starts: specimen, based on their fitness, are chosen to be parents to new solutions. Children resulting from such union have a combination of genes from both parents, although the exact algorithm varies between different *crossover* operation implementations. The children are produced until a new population's size is equal to the previous one. Additionally, new population is then subject to a process of *mutation*, meaning that each child has a chance to have parts of his genes changed; this introduces some variation in the population as well as helps breaking out of local optima and broadening the search space [9]. With all steps finished, the whole process (sans the random initialization part) repeats until an end condition is met.

Genetic Algorithm can be thus summarized with following series of steps:

Initialisation – start by producing (randomly or otherwise) a population of candidate solutions.

Evaluation – assign each specimen in current generation a fitness value determined by fitness function.

Recombination – choose (using chosen selection method) two parents and combine their genes in *crossover* process, producing two children. Repeat until new generation is equal in size with parents' generation.

Mutation – subject current population to a *mutation* process providing a chance to change individual genes of a specimen.

Furthermore, one has to make several key decisions during preparation to using Genetic Algorithms. These are commonly (after [7]):

specimen representation – a mapping of the solution information to a data structure handled by the algorithm.

population size – number of specimen in each population.

fitness function – a function to assess the value of the specimen.

number or generations – an amount of iterations to execute before the process is finished.

additional end-conditions – an early option (or options) for early termination of the algorithm.

selection method – a strategy to choose parents for the next generation.

crossover and mutation methods – an exact implementation of genetic operators.

mutation and crossover rates – a rate by which said genetic operators affect the population.

Selection, Crossover and Mutation strategies have a great deal of influence over the algorithm's result and will be discussed further in sections below.

2.2.3 Selection

Selection process is a entry point to the recombination process - it's here where two specimen are chosen from the population pool to either be copied to the next generation, or become parents to (traditionally) two children, which are then added to a it instead (parents, however, are "returned" to the old one, able to be selected again). Therefore, candidates should be seen as individuals competing for a possibility to extend their lineage and pass their genes on to the next generations. Following Darwinian law, individuals with higher fitness value associated to them should have the higher chance to be selected - that was the original premise behind **Fitness Proportionate Selection**, largely popular during early developements of Genetic Algorithms [9].

In **Fitness-Proportionate Selection**, also known as **Roulette selection** [9] specimen are selected in proportion to their fitness: solutions with higher fitness value associated have a bigger chance of being selected. Not only does it introduce an element or chance, in contrast to methods like **Elitism** or **Truncation Selection** [9], where one simply copied n best specimen (repeating that operation until new population was the right size), it also doesn't restrict the option of "weak" individual being selected to carry through genetic material that potentially can prove to be interesting in future. Using Roulette Selection, however, one makes a big assumption:

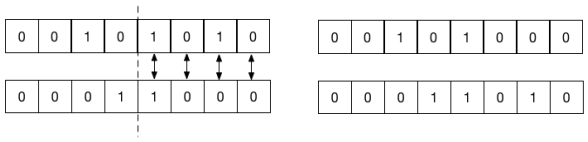
> They presume that the actual fitness *value* of an individual really means something important. But often we choose a fitness function such that higher ones are "better" than smaller ones, and don't mean to imply anything more.

[9]. This can pose a significant problem during later iterations of the algorithm, when most of the solutions may have similar fitness values, effectively turning a theoretically fitness-proportional choice to strategy only slightly better then random one.

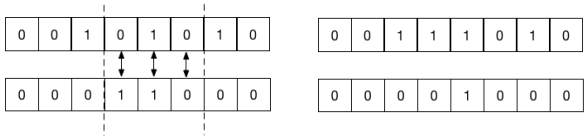
Tournament Selection deals with that problem in a curious way - instead of considering the fitness function values, it merely takes into consideration the ordering. The algorithm is definitely on the simpler side: a random sample of the population is chosen, and the best individual in that sample is returned as a result (hence the "tournament"). That invulnerability on fitness function's particulars, along with the simplicity are some of the reasons why Tournament Selection has become the primary selection technique used for Genetic Algorithm [9]. Furthermore, this method offers additional great advantage - its behavior easily adjustable. By setting a tournament size small, for instance, one can expect it to be less *selective*, since "weaker" individuals might still win in smaller instances of the tourney. By setting it to a large value, one can expect the method to strongly prefer top performers. Setting it to a *too large* value, however, will guarantee that the new population will be constructed from one specimen.

2.2.4 Crossover

Unlike other evolution-inspired techniques, in GA selected individuals are not always explicitly copied to the next generation. Instead, selected individuals have a chance (dictated by **crossover rate**) to undergo a process that will recombine their genes in order to form two offspring (although a research by Eiben et al., 1994 mentioned in [12] suggests that using more than two parents might generate higher quality children). The common method of doing that seems to be One- or Multi Point Crossover, which involves choosing a random point (points) along each specimen representation and exchanging genetic material between them. Figure 2.2 illustrates the process of One- and Two-Point Crossover operations. In some cases crossover might prove to be quite a



(a) One-Point Crossover



(b) Two-Point Crossover

Figure 2.2: Crossover Example

destructive force. Namely, it has to do with *linkage* (or *epistasis*) between elements in specimen representation [9]. Elements that are feasible in current setting might, and will be, broken up, and one can easily imagine crossover operation producing a number of "broken" or otherwise flawed individuals.

2.2.5 Mutation

Mutation operation serves as an assurance that the algorithm won't converge prematurely by introducing that much-needed variance. It's usually performed on a whole population, where each individual has a chance of his genes randomly changing (provided by **mutation rate**). Depending on the actual representation of the chromosome, popular implementations include a bit-flip (in case of boolean strings, for example) or replacing a value with one randomly generated with given distribution. Figure 2.3 presents the process of mutation on a fixed-lenght bit string. In this particular example, two flips are executed.

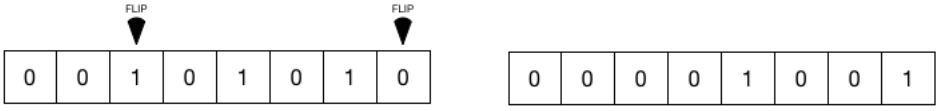


Figure 2.3: Mutation Example

2.3 Genetic Programming

2.3.1 Motivation

John R. Koza, creator of Genetic Programming, wrote the following on motivation and scope of this field [7]:

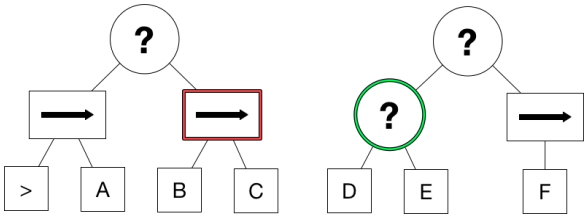
> The genetic programming paradigm continues the trend of dealing with the problem of representation in genetic algorithms by increasing the complexity of the structures undergoing adaptation. In particular, the structures undergoing adaptation in genetic programming are general, hierarchical computer programs of dynamically varying size and shape.(...) I claim that the process of solving these problems can be reformulated as a search for a highly fit individual computer program in the space of possible computer programs. When viewed in this way, the process of solving these problems becomes equivalent to searching a space of possible computer programs for the fittest individual computer program.

2.3.2 Key Differences

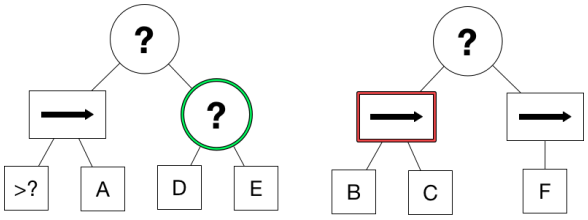
Apart from the issue of representation, Genetic Programming is, in its essence, merely an adaptation of Genetic Algorithm, with further changes to accommodate handling of a different data structure. As such, only said changes will be described here, while the remaining parts of the algorithm are assumed to be unchanged. Note here that in interest of convenience, all examples below have been presented on Behavior Tree structures.

Crossover

Crossover works in similar way to its Genetic Algorithm counterpart, selecting a random nodes in each parent and exchanging them (along with their respective subtrees). Figure 2.4 illustrates two specimen before and after crossover procedure.



(a) Step 1: selecting nodes to be exchanged.



(b) Step 2: performing the exchange.

Figure 2.4: Crossover Example

Mutation

Changes to mutation, however, took a more significant turn. Genetic Programming mutation is defined in two ways: one still could be rather similar to Genetic Algorithm mutation (**micromutation**). The other one, called **macromutation** (or **"Headless Chicken"**) is an entirely new way of introducing variance to a population. While micromutation consists of randomly selecting a node in a tree and changing its parameters, macromutation is executed by replacing a random node with a new, randomly-generated, tree - in certain cases, it may result in replacing the whole tree. Figures 2.5 and 2.6 present an example of performing micro- and macromutation.

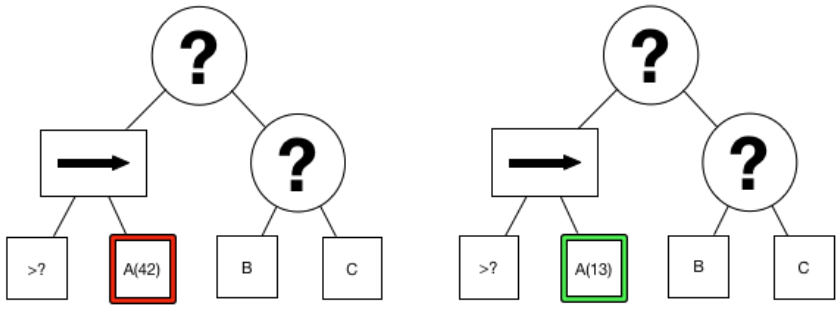


Figure 2.5: Micromutation: choosing a random element in a tree and changing its properties.

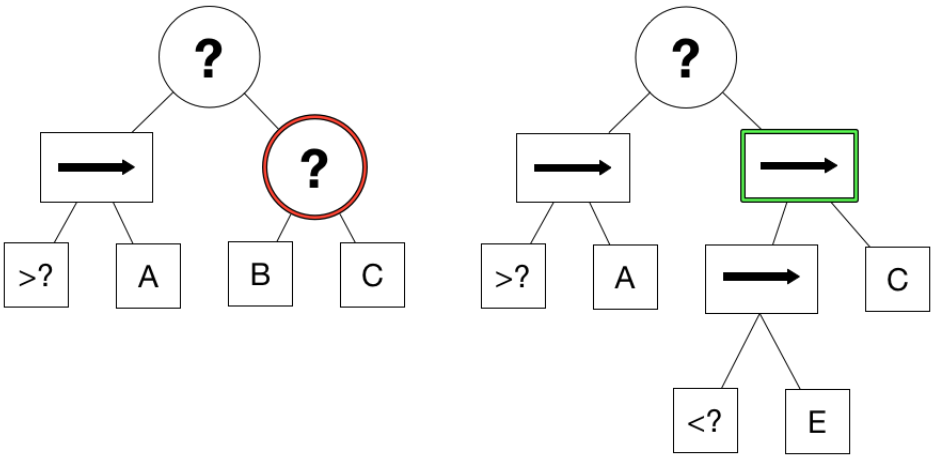


Figure 2.6: Macromutation: after a random node is selected, a new tree is generated and its contents replace selected node.

3 Related Work

The idea of evolving Behavior Trees using Genetic Programming method is not an entirely new concept. Similarity in representation of the specimen (both Behavior Trees and originally proposed by J. R. Koza in [7] lisp programs are represented as a tree structure) allowed for nigh painless adaptation of the GP algorithm to the problem, while the popularity of Behavior Trees as the AI representation of choice has grown past game programming areas exclusively, venturing into previously unexplored use cases.

Earlier implementations of this idea include evolving Behavior Tree agents for a turn strategy game DEFCON in [8]. In this paper, authors applied Genetic Programming method to evolve Behavior Trees acting as an artificial agent playing the game against existing AI developed by Introversion Software Ltd. (creators of the game). The goal was producing a automated agent able to beat the reference bot more than 50% of time over a sufficiently large number of games. Their experiments resulted with an evolved AI-bot able to beat the reference agent 55% of the time in span of 200 games. The chosen method involved manually defining a Beavior Tree, then evolutionary swapping and modifying its branches and node parameters. The finished agent was the sum of partial Behavior Trees (utilizing the modularity feature) that were evolved for a specific behavior, a sub-goal, like resource positoning or fleet production.

Another research applied the method to evolve a controller for DelFly Explorer UAV, tasked with locating and flying through a window placed in a simulated (and later on, physical) room [12]. In his thesis, Scheper constructed and used a framework of constructing and evolving Behavior Trees to obtain an agent with 88% success rate in a simulated environment and 54% in a real one. Furthermore, Scheper was able to reencounter and better describe some of the [8] findings related to evolving Behavior Trees specifically, like the low number of generations needed achieve a fit solution, indicated by mean fitness reaching plateau. This was also the first time that evolutionary Behavior Trees were introduced into Robotics area of research.

Cooperation among artificial agents in multi-agent system environments is a well-known subject, with multiple papers detailing state-of-the-art techniques [11], [5]. The agents in multi-agent systems can present a wide range of behaviors, depending on the environment: their behavior can be classified as either cooperative, competitive, or (in case when said behavior is not apparent enough to be classified) indistinguishable. [5] classifies agents as *selfish* (maximizing their private utility function) or *utilitarian* (maximizing the group utility), in order to focus on their intention rather than actual behaviors presented. Selfish agents can still present behaviors that could be classified as cooperative, as long as it is a viable option, especially in trivial or no-conflict games (per [5] classification).

A classic example of a matrix game with cooperative behaviors possible, and a model

subject for many behavior-related studies (among them, referenced in this work [5], [11], [1], [2]) is Prisoner's Dilemma. The game, though simple in nature, posed a serious problem to the existing game theory - the most rational action (to deflect) wouldn't neccesarily bring the highest payoff. [1] formalized an iterated version of the game with unspecified number of interactions between the two prisoners to present how cooperation based on reciprocity, when introduced to the population, might evolve and finally establish itself, resisting invasion from other strategies in the population. Study by van den Berg & Weissing [2], however, discovered and related how changes in evolutionary strategy and mutation regime can bring a vastly different in evolutionary outcome (average level of cooperation, specific cooperation strategy). The methods used in the study were 1:1 genotype-phenotype mapping and a simple neural network.

Finally, there seems to be a constant, unsatiated need for capable testing environments. Undoubtedly, modifying and fine-tuning features and code of the testing environment are the most sought-after features, this being the reason why so many researhers turn to in-house developed solutions. There were, however, a number of attempts of introducing a general-purpose testing environment ([10], [13], [4]) or, at the very least, *generalized* testing environment. Still, openess of the code and ease of modifications seem to prevail being the deciding factors.

4 Method

4.1 Motivation

Albeit the goal stated in the introduction chapter is focused at the method of obtaining quality agents, the motivation for such endeavor must be kept in mind. It would be beneficiary for a number of use cases to obtain a system which, when provided with a sufficient description of an environment and capabilities of agents, could be used to produce ready to be implemented Behavior Trees capable of performing given assignments. Arguably, additional overhead due to implementing the environment & agents specifications and task description must be taken into account when considering using such system, potentially making the method unfeasible in scenarios when the time is of essence. Even then however, one has to weight initial investment into such project against manually designing and adjusting the AI.

Additionally, there is also one critical advantage to using evolutionary learning methods: the possibility of early termination through end condition. With the right fitness function the option to take an undeveloped tree as a template and improve it becomes perfectly possible. This can be useful in more sophisticated problems, when manual tuning will complement evolutionary method's solution.

4.2 Formulating a Task

In order to properly assess feasibility of the system in making, a common task had to be constructed to compare different variations of a system in-making against each other. Scenario type chosen for this task was a resource gathering.

The premise of this particular scenario was as follows: a certain amount of numbered resource markers (portrayed as flags in the simulation) were scattered across the simulated environment resembling a warehouse. The agents, starting from designated spawn points, are tasked with claiming as many of them as possible against a fixed time limit. *Claiming* in this case was a process consisting of approaching a certain vicinity of a marker, at which point it was despawned and marked off as *captured* by an agent initiating the claiming. Making the process instantaneous ensured that markers would be claimed on FCFS (*First-Come, First-Served*) basis. Accompanying the evolutionary agent was the human-defined tree, filling a role of a rival - in a competitive scenario - or a partner - in the cooperative one.

While the particulars were a subject to fine-tuning numerous times, Figure 4.1 presents the initial schema of agents' and markers' spawn points.

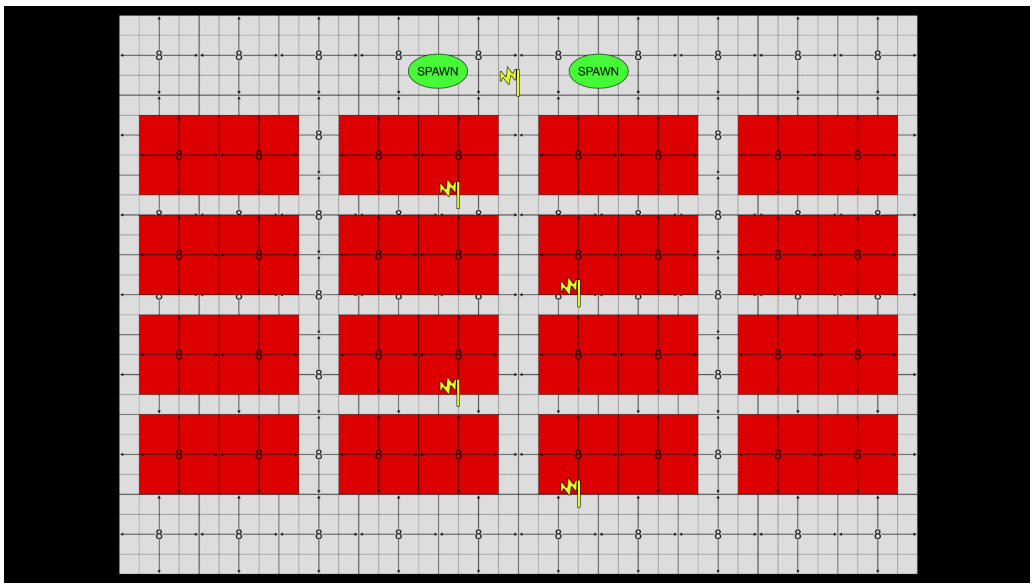


Figure 4.1: Initial positioning of agents' starting points and resource markers.

4.3 Incorporating Genetic Programming

The Genetic Programming module was built on top of FRAIL's existing features, substantially changing the usual flow of execution to make it suitable for learning processes. Initialization, evaluation and reproduction steps are all done internally, with evaluation being a step when the actual simulations are run. Since the population was being tested sequentially and the maximum time one could run was 20s (with the possibility to end early, in case all markers were gathered before that time), some testing would take more than 48 hours to complete. We decided to increase update frequency of the simulations tenfold, thus guaranteeing simulations would complete after maximum of 2s.

4.3.1 Representation

In the interest of avoiding integrity requirements and semantic incoherences (which would be undoubtedly caused by genetic operators), specimen are instances of a specialized class, separated from the actual Behavior Tree class responsible for tree execution. These trees are then parsed during evaluation step to a "regular" Behavior Tree that is injected to an AI Controller in the simulation.

This simplified implementation allowed for easier manipulation of the trees' components. The nodes are distinguished by their type, but otherwise have all parameters required to operate. While this poses a serious data redundancy, the possibility of freely adding additional node types and ease of operations made this option highly desirable.

4.3.2 Genetic Operations

Initialization

The population starts as a set of randomly generated specimen of varying size and depth. Each tree generation starts with a random Composite node in root, then the decision is made a at each further step to either add a random Action / Condition node to current root's children or recursively create a random subtree, all bound by a maximum size and maximum depth allowed. This approach provides a varied population while maintaining good balance of Action / Condition and Composite nodes.

Selection

The selection method of choice is a regular implementation of Tournament Selection. At each step, a random sample of a given size is selected from the current population and sorted according to a fitness value of the specimen. The first individual from this set is selected for Reproduction purposes.

Crossover

On each iteration, two selected specimen have a chance (determined by crossover rate parameter) to produce two children which will be put into the next generation instead of them. The crossover itself is a single-point version, implemented as a exchange of randomly selected nodes (one from each tree) without minding their type nor location.

Mutation

Genetic Programming framework offers two methods of mutation: micro- and macro-mutation, explained in chapter 2. Both were implemented: initially, only the possibility for the tree to undergo micromutation is tested - that is implemented by resetting randomly chosen node's parameters to a random values. However, each tree selected for a micromutation process (probability of which is dictated by a mutation rate) has a chance to undergo a macromutation process instead - that, in turn, is determined by a separate, macromutation rate. In this case, micromutation step is replaced entirely with "Headless Chicken" mutation.

Fitness function

Due to the format of the experiments, where solutions to two different scenarios were explored separately, used fitness function varied between experiments. There is, however, a number of factors that were considered relevant to the fitness of the specimen that can be presented now. These were:

- **Number of markers gathered** - a number of resource markers gathered.
- **Time to completion** - how much time did it take to finish the simulation (i.e. how much time has elapsed until all the markers were claimed).

- **Size of a Tree** - how many nodes were in the tree.

The first characteristic ended up being relevant only in the competitive scenario, where agents were to compete in claiming the markers: the number of claimed resources was, in fact, the defining characteristic. The relevance of the two following features, however, was common to both cases: the best solution is expected to finish the task as close to the shortest time as possible, while pressuring the algorithm to keep decreasing the size of a tree was done in the interest of both readability (and, with that, ease of modification) and execution time.

4.4 Behavior Tree

The solution produced by the algorithm had to be *translated* to an actual Behavior Tree representation used in the simulations for the reasons discussed earlier in this chapter. The process was, however, a part of the usual workflow and required no interaction from the user.

Specimen were built from only three kinds of nodes: *Selector*, *Sequence* and one Action - *goToFlag(flagNumber)*. This decision was made based on relative simplicity of the task, not requiring elaborate choice in actions. *goToFlag* automatically chooses the optimal route to the selected path and, when the actor approaches capturing distance, attempts to claim the marker. Note that there is no information provided on the marker still being in the chosen place, as well as no knowledge at all about the state of the board. Agents operate thus in an environment with incomplete information, with the position of the markers being the only information available.

Furthermore, *goToFlag* Action provided additional two variants, tied with the identifier argument. While values of 1-7 would select one of the existing markers, values -1 and 0 held special meaning: -1 would set the course on the furthest *available* (non-claimed) marker, while the latter would select nearest one. This was done in order to see if that level of abstraction would be preferable in certain situations, acting with critical timing against unchanging reference agent. The listing below presents pseudocode algorithm of *goToFlag* Action.

```
actionGoToFlag(int identifier) //checks if identifier is in range [-1, 7]
{
    if (identifier == -1)
        goToFurthestFlag()
    else if (identifier == 0)
        goToNearestFlag()
    else
        goToFlag(identifier - 1)
}
```

4.5 Environment

4.5.1 Introduction

FRAIL (Framework for AI Laboratories) is a platform developed in-house at Wroclaw's University of Technology (PWR) for purposes of testing artificial adversaries in shooter games. [4]

As an in-house project, users were able to freely access and change the source code, modifying existing controllers or creating new ones. Since its launch, FRAIL has seen use as a didactic aid during Artificial Intelligence classes and bootstrap for various AI-related projects, including evaluation by a video game development and publishing company, Techland for internal use.

In order to fully utilize the platform's capabilities, substantial changes have been made to FRAIL's engine in one of the previous projects, allowing for the entire process of Genetic Programming to be executed on the platform itself. A Behavior Tree implementation and representation related modules were taken from current version of FRAIL and the project was reverted to one of the earlier versions to reduce bloating. Genetic Programming module was implemented as a "building block" that could be placed on any map operating in "Capture the Flag" mode. Exploiting FRAIL's scripting behaviors and usage of RTTI (Run-Time Type Identification), a number of convenience changes were introduced to control the environment and algorithm parameters.

4.5.2 Map overview

The task planned for the agents can be described as a resource retrieval. To simulate this scenario, a map resembling a warehouse has been planned and prepared for use. Figure 4.2 presents a 3D render of a finished map, while a top-down view, featured on Figure 4.3 will be referred to further describe the design.

The map was designed with two to four agents in mind, with the possibility of scaling up when needed. Its shape is rectangular, measuring 80 on 56 simulation units. Over the middle section, 14 cuboid structures were placed symmetrically to act as storage containers / shelve-holding space, with access points along every edge. This placement allowed to make use of narrow alleys between containers in order to create potential conflict points, and positioning resources right next to them would hopefully allow for more faithful representation of a real-world resource gathering problem.

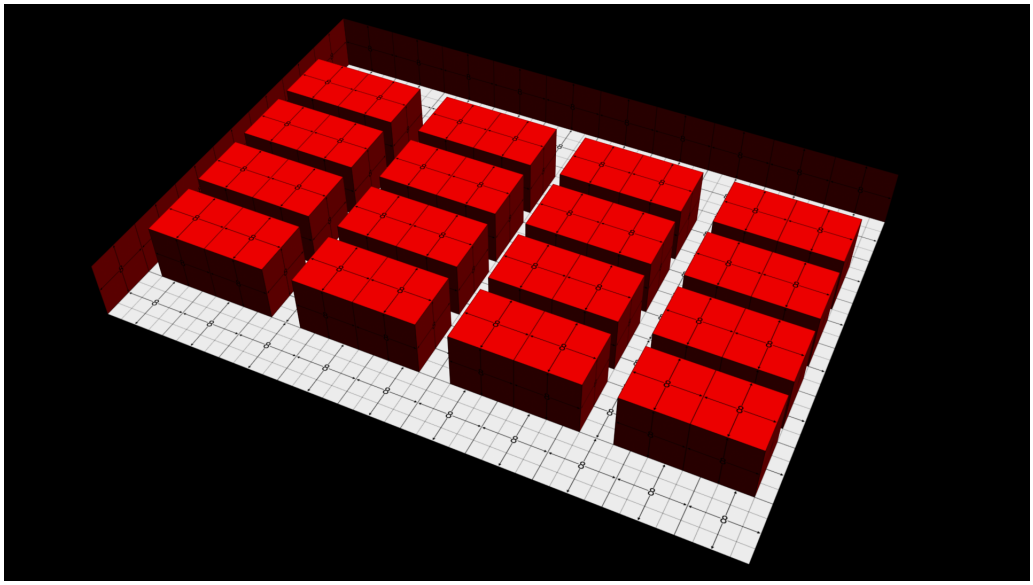


Figure 4.2: A rendering of scenery

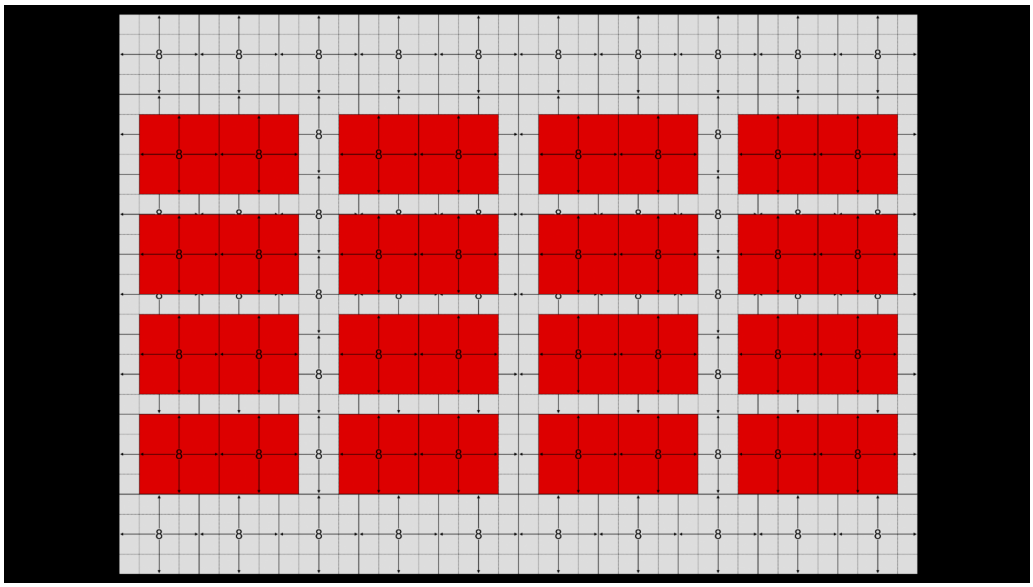


Figure 4.3: A top-down view of the map

5 Results

To learn how cooperative behaviors may influence success ratio in a particular task, the decision was made to first observe the evolution in case where agents work against one another, competing for a higher score. Only then, comparing the results with those of a cooperative case, conclusions regarding cooperation influence may be drawn.

The scenarios were tested in parallel with each change, were it to complicate the problem or put emphasis on certain components of the solutions.

The two scenarios shall be described in their dedicated sections below, focusing on key contrast points. Following, however, are the elements common to both cases.

Evaluation process of candidate solutions was made in the same environment, with identical starting points and resource markers' locations. In both scenarios, trees produced by Genetic Programming are accompanied by a unchanging, user-defined tree in the simulations. This reference agent uses one type of Action, *goToFlag(flagNumber)*, visiting each resource marker in turn, capturing them from first to last, never changing path between simulations. Figure 5.1 presents the Behavior Tree schematic of the reference agent. Existence of such tree could fill the function of both a pressuring component (reflected in the fitness function in the appropriate cases) and essentialy a time-gate, as gathering all markers would end the simulation. Furthermore, algorithms in both cases had access to identical Behavior Tree Components to build the trees from, thus ensuring equal sophistication of solutions in each scenario.

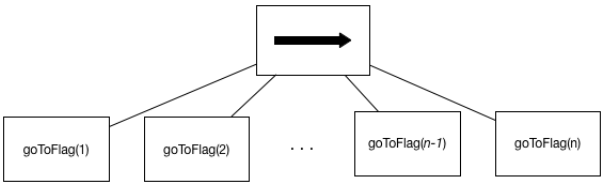


Figure 5.1: A schematic of reference agent used in simulations.

5.1 Competitive Scenario Specification

5.1.1 Synopsis

Competitive case assumed the optimal solution to the problem is maximizing the agent's personal gain. The markers gathered were counted for each agent separately, thus turning the task into a race - evolutionary agent was forced to get to as many markers as possible before the reference agent would claim them for himself.

After initial testing, the scenario was modified to feature 7 resource markers (instead of initial 5) and modified spawn points of agents - readjusted to move evolutionary agent further back from the first marker. While increasing the number of markers served as an attempt to complicate the problem, moving evolutionary agent further from the first flag was done in the interest of being able to reproduce the results: since both agents were in the equal distance of the first flag and moved with the same speed, the matter of which one would claim the first flag was comparable to a coin-toss. Figure 5.2 presents the final view of the competitive scenario map.

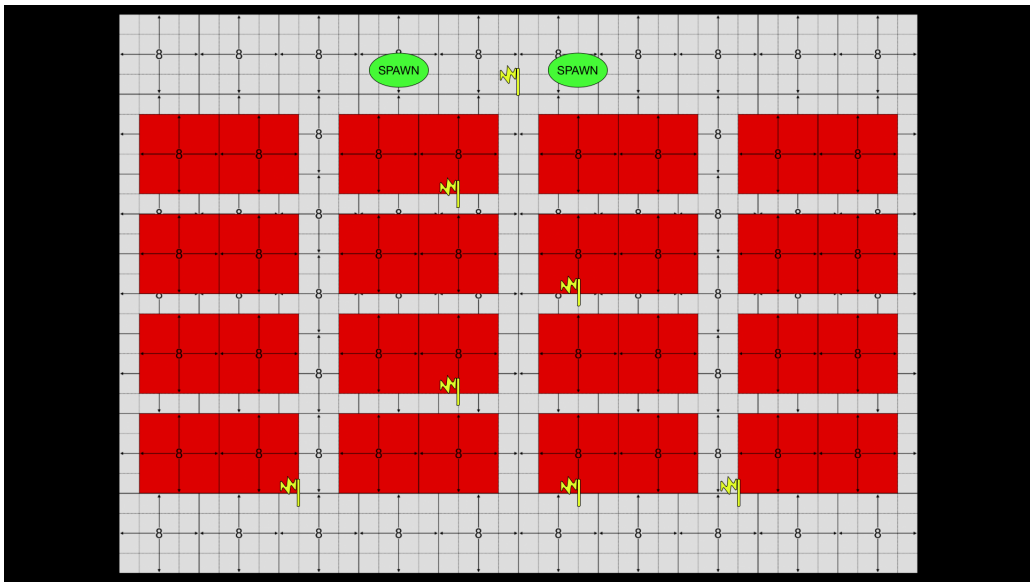


Figure 5.2: Finished map of the competitive scenario environment

5.1.2 Fitness Function

The competitive scenario is effectively a zero-sum game: a gain in utility of one agent means (prospective) loss of utility of the other's. Considering that, the initial fitness value of the $i-th$ specimen in the population was a sum of n markers gathered by an agent controlled by a tested tree, multiplied by a resource point constant C_1. This, however, wasn't feasible as a long-term solution. To introduce differentiability between the specimen scoring the same amounts of markers, a factor of time in milliseconds multiplied by constant C_2 was then subtracted from the sum. Additionally, to coerce the algorithm into preferring smaller trees, a tree size factor with weight C_3 was further subtracted from this value. The finished formula for fitness function is presented on Equation 5.1.

$$f(i) = C_1 * n - (\frac{t}{C_2} + \frac{s}{C_3}) \tag{5.1}$$

5.1.3 Success Criterion

With the goal of maximizing personal gain, which is the case with the considered scenario, desirable trees are those that claim every possible marker. However, since evolutionary agent's spawn position put it at the disadvantage (making it impossible to gather the first flag), only solutions with $n - 1$ flags gathered out of n possible would be considered "successful". The true goal of a ideal solution was then to create a big enough difference in time taken to visit the markers, allowing it to reach and claim the last one.

5.2 Cooperative Scenario Specification

5.2.1 Synopsis

Cooperative scenario, in contrast to the "selfishly" motivated competitive one, takes a broader perspective. Instead of competing, agents were expected to work with each other towards claiming all 5 markers. However in time, the layout of the scenario was modified to maintain uniformity between the two cases' layout - that included both increasing the number of flags to 7 and modifying the agents' spawn positions - for comparison reasons.

5.2.2 Fitness Function

Due to competitive aspect being gone, a perspective on performance shifted substantially. The number of markers claimed by an agent became an impractical characteristic, with the time factor providing a much more considerable view on specimen performance. However, the decision was made to include the *total number of markers* in the fitness function, serving as a constant, to ensure that the results from both scenarios are of the same order of magnitude. Aforementioned time and tree size components remained in untouched form, all the more critical in case at hand. Equation 5.2 presents the fitness function used in evaluating specimen in cooperative scenario.

$$f(i) = C_1 * (n + m) - (\frac{t}{C_2} + \frac{s}{C_3}) \tag{5.2}$$

5.2.3 Success Criterion

In the cooperative case, the underlaying goal is to divide the markers between two agents in the most efficient way. To effectively operate, the desired outcome is achieving completion time shorter than the time it takes for the reference agent to complete his route. In other words, a successful solution will be able to successfully adapt itself to a potential partner's route and improve upon it in terms of time.

5.3 Fitness Function Parameters

The aforementioned point constants of the fitness function, defined as follows:

- C_1 - resource marker point value
- C_2 - time factor weight
- C_3 - tree size weight

Were tuned by using *grid-search* algorithm with a number of possible values of C_2 and C_3. C_1, however, remained unchanged. The decision was made that, for the purposes of the competitive case, the number of resource markers should remain the deciding factor. The values of the parameters after grid-search tuning are presented in tables 5.1 and 5.2.

Parameter Name	Parameter Value
C_1 - resource marker point value	1000
C_2 - time factor weight	0.1
C_3 - tree size weight	100

Table 5.1: Selected Fitness Function factor weights for Cooperative Scenario

Parameter Name	Parameter Value
C_1 - resource marker point value	1000
C_2 - time factor weight	0.01
C_3 - tree size weight	10

Table 5.2: Selected Fitness Function factor weights for Competitive Scenario

5.4 Genetic Programming Parameters in Experiments

After observing the results from a number of initial test runs as well as using *grid-search* algorithm to help determine *crossover rate*, *mutation rate* and *population size* parameters, the parameters used in the experiments are presented in table 5.3.

After a number initial runs, it was clear that keeping a maximum number of generations above one hundred served no function - there were no instances that did not converge before that point and following were long periods of stagnation or, even worse, loosing best specimen. The method responded positively to increasingly higher number of specimen in the generation: totaling at 200, this ensures a good variety of genetic material for the algorithm to make use of. Crossover and mutation rates were kept low due to the risk of destroying relatively low number of good specimen each generation. On the other hand, they could not have been set *too low*, lest the algorithm would

Parameter Name	Parameter Value
Maximum Number of Generations	100
Population Size	200
Crossover Rate	25%
Micromutation Rate	15%
Macromutation Rate	2%
Tournament Selection Tourney Size	10
Starting Generation Minimum Tree Size	20
Starting Generation Maximum Tree Size	30
Maximum Tree Depth	6

Table 5.3: Selected Genetic Programming parameter values.

not explore the search space properly. It is worth pointing out that macromutation ("Headless Chicken") was kept at extremely low occurrence chance specifically for that reason - there weren't any instances suffering from too early convergence, something that macromutation would have been a remedy to. Choosing a 5% value of tourney size ensured it would still be possible to get specimen with mid-range fitness value into the next generation while keeping the selection pressure relatively high.

Table 5.4 presents selected parameters used during tree generation steps in the algorithm. The tree related parameters were chosen after a careful consideration of desirable

Parameter Name	Parameter Value
Starting Generation Minimum Tree Size	20
Starting Generation Maximum Tree Size	30
Headless Chicken Generated Tree Minimum Size	10
Headless Chicken Generated Tree Maximum Size	20
Expected *Action-to-Composite* node Ratio	3:1
Maximum Tree Depth	6

Table 5.4: Selected Behavior Tree generation parameter values.

feats both in the initial generation as well as during rarely-observed macromutation. Generating the tree between 20 and 30 nodes would create a varied populations with different shaped trees, while maintaining decent complexity bound by maximum tree depth. The need for larger number of Actions in relation to Composite nodes was dictated by a simplicity of the task and in the interest of further coercing the algorithm to prefer simpler structures.

5.5 Exemplary Specimen

In this section, a sample specimen representation is presented and described. The listing below presents the text representation of a tree, while figure 5.3 contains a (arguably)

more human-readable form.

```
Sequence
  ActionGoToFlag 0
  ActionGoToFlag 3
  ActionGoToFlag 4
  ActionGoToFlag 4
  Selector
    Sequence
      Selector
        Sequence
          ActionGoToFlag 7
          ActionGoToFlag 5
          ActionGoToFlag 6
        ActionGoToFlag 6
      ActionGoToFlag 0
      ActionGoToFlag 3
      ActionGoToFlag 4
    ActionGoToFlag 3
    ActionGoToFlag 7
  Sequence
    ActionGoToFlag 2
    ActionGoToFlag 3
  ActionGoToFlag 4
  Selector
    ActionGoToFlag 1
    ActionGoToFlag 1
  ActionGoToFlag 5
  ActionGoToFlag 2
  ActionGoToFlag 3
```

This particular specimen was generated during one of competitive scenario experiments and his evaluation placed it on value 5911. Knowing both the value of fitness value constants it can be proven that the specimen claimed 6 resource markers out of 7 possible.

The tree has a very cluttered look and many unnecessary placed nodes, but the route should be clear: going from the top, it's going to visit the closest marker, which is the second one (sic!), then the third, proceeding to the fourth one (twice, in fact, but the second call will return instantly) and head for the last one, only to return to the fifth and sixth, probably meeting a reference agent nearby fifth marker. The rest of the nodes, as mentioned before, serve no function in this particular scenario. Unfortunately, the tree didn't follow the expected strategy (gaining a bit on each marker only to make a final push from sixth to seventh) instead opting to devise a new one. Although - as seen above - the strategy of choice seems to be at least equally viable.

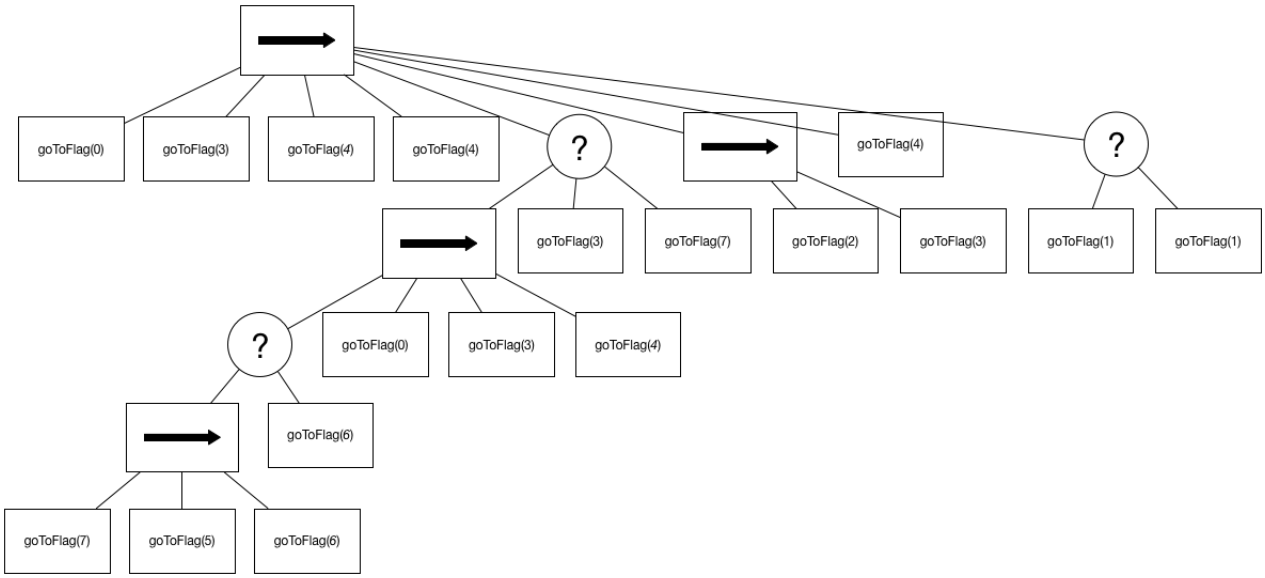


Figure 5.3: Example diagram of a evolutionary generated specimen.

5.6 Optimization Results

In this section, results of model cases (one from each scenario) will be presented. When analyzing the results, following factors were considered:

- The shape of the algorithms' best specimen fitness value curve and its relation to average fitness value one.
- An average size of a tree throughout the iterations.
- Difference in node count from a *minimal* successful solution
- A number of iterations needed for the best specimen to achieve success.

The shape of fitness plots indicate the general state and health of the Genetic Programming algorithm. While the plot indicating top fitness value in the population is certainly important (the success measure is directly tied to a top specimen), it's the average fitness value in the population that provides the much needed information on population growth, stagnation or even oscillations, indicating local plateaus.

The average size of a tree is also a useful indicator of a general perceived value of the generation: while larger trees are not necessarily always worse (since the execution time of a tree is virtually negligible), they most definitely contain needless constructs. Apart from being nigh unreadable, disregarding tree size would certainly result in need of introducing post-processing to prune a number of never-to-be-executed branches.

The task was designed in a way allowing for the optimal solutions to be relatively simple in structure: such solution, written manually, would be composed of no more than 7 (in competitive scenario) or 5 nodes (in cooperative one). These solutions would

be deemed *minimal*. Since tree size is one of the factors determining a specimen fitness, it is reasonable to expect the algorithm to converge on a tree that is close to that size.

Finally, the number of iterations needed to produce a first solution that is deemed "successful". Looking into the matter from the user's perspective,the only feasible characteristic is the speed of achieving a *fit* specimen, able to deal with the task at hand with given restraints.

Figure 5.4 presents a plot of the highest fitness value and average fitness in the population in respect to iteration in the competitive scenario. Figure 5.5, however, illustrates the change of average tree size in the population in the course of the algorithm's run in the same scenario.

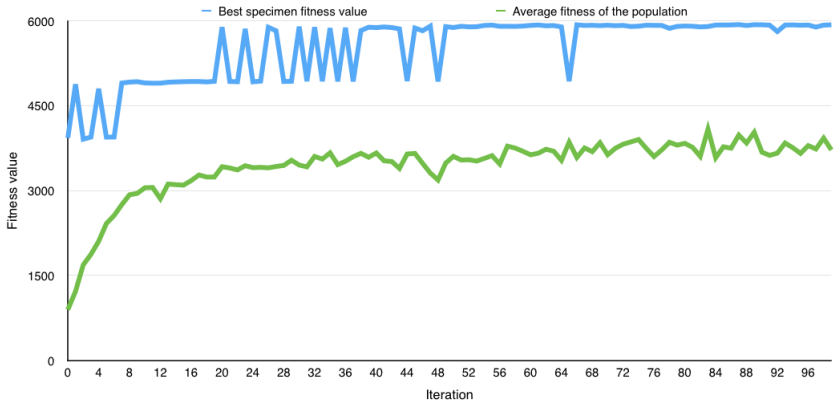


Figure 5.4: A fitness value with respect to iteration plot in the competitive scenario.

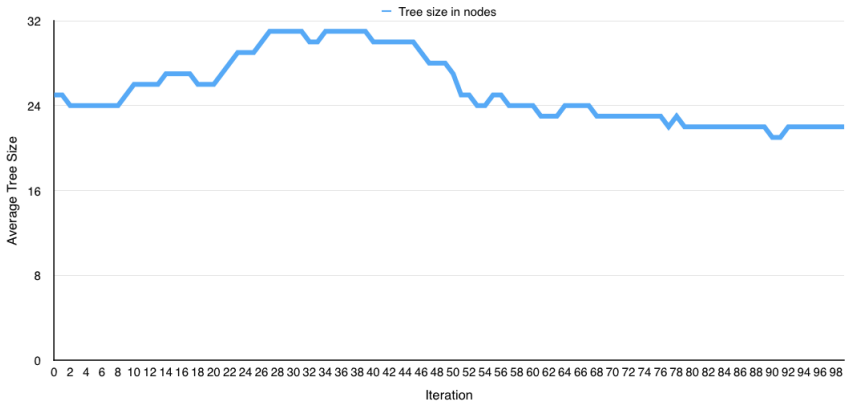


Figure 5.5: An average tree size with respect to iteration plot in the competitive scenario.

The algorithm's run in this particular case started on a relatively high note, with the best specimen claiming 4 markers for himself. In the next few iterations, best fitness achieved increased, then oscillated around 4200 value, to finally stabilize on a value closer to 5000. Up to this point, one can see the average fitness in the generation

rising, indicating that, while there were no radical increases in value, the algorithm is producing more and more feasible options. Then, surely due to particularly good mutation or crossover, a breakthrough is made and the population achieves its first successful specimen. This one, however, is instantly lost, having not been copied over to the next generation. The curve then starts oscillating heavily, with the algorithm producing further successful specimen only to lose them few cycles later. Finally, around iteration 50 mark, the best specimen is kept for a little longer. Note that the first successful individual marked the decline in average fitness growth. That proves that by this time, the population has been saturated with useful gene structures.

The size plot is intended to be considered and analyzed together with the previous one. Having done that, one can observe that with the first successful specimen produced, the population started intensively increasing in node count. Having achieved large enough pool of fit solutions, size started to be the differentiating factor, successfully lowering the number. This is the perfect example of larger trees being promoted in the initial stages (since they have a larger chance of containing a useful node structure), but having their value decreased as the useful genes from them are passed into further generations. This also poses an interesting question - whether the first found successful specimen should be accepted, or will processing more generations rewarding?

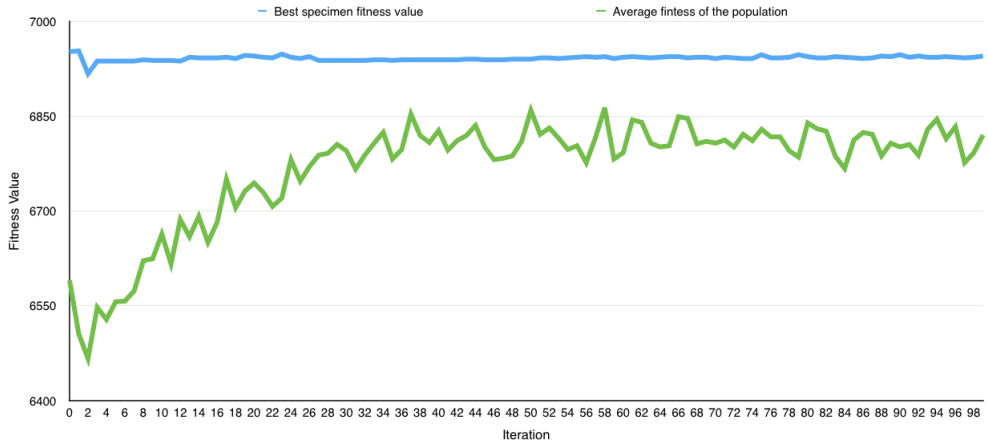


Figure 5.6: A fitness value with respect to iteration plot in the cooperative scenario.

Having seen the competitive scenario, the fitness plot in figure 5.6 might not appear as telling at first. Having produced a successful specimen in the first generation, the individual is instantly destroyed, its genes diluted in the next generation. Another one, however, is produced as quickly as the fourth iteration of the algorithm. The value then remains virtually unchanged, oscillating only slightly to the last generation. Average fitness value curve reflects that progress well, with the oscillations starting only when the other one stabilizes.

The feeling disappears, however, looking at the average tree size change presented in figure 5.7. On 30th generation, when the fitness value of the best specimen appears to enter stagnation, abrupt decrease of an average specimen node count begins. Over the

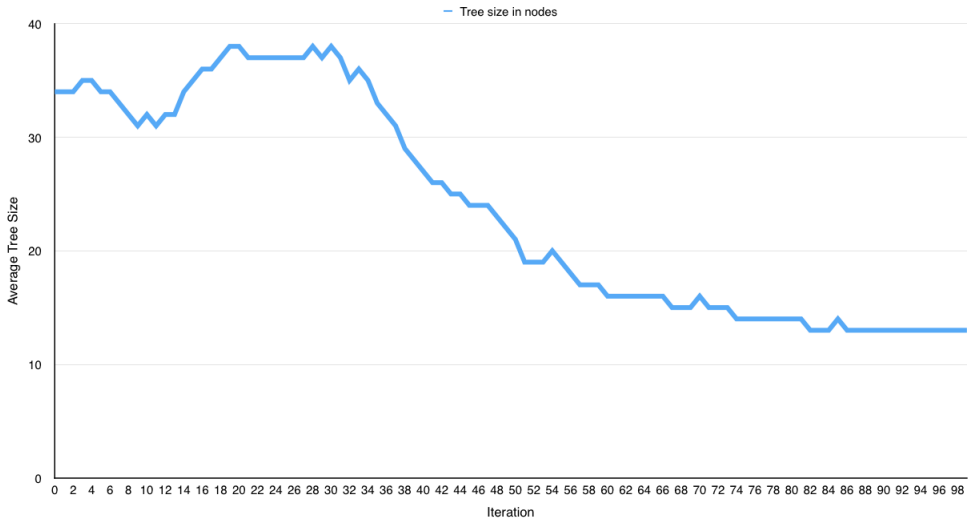


Figure 5.7: An average tree size with respect to iteration plot in the cooperative scenario.

next 50 iterations of the algorithm, the average size is reduced almost thrice - from 36 nodes to 13.

6 Conclusions and Future Work

The aim of this thesis was answering the research question: "How cooperative behaviors between artificial agents influence the performance in multi-agent task-oriented environments?". On the grounds of success criteria defined in the previous sections: the increase of speed in reaching the solution, the size of a successful candidate and speed of task execution itself all testify to cooperative behaviors having a considerable influence in tested task. Specimen in the cooperative scenario achieved quicker completion time compared to those evolved in the contrasting, competitive one. Furthermore, later iterations in the cooperative case would feature specimen of much smaller size executing the task with equal effectiveness.

These results prove that cooperative behaviors are, in fact, *feasible* to evolve. With that said, the work here merely touches the surface of genetically evolving Behavior Trees and the behavior types they produce. Future work will focus at deeper analysis of the population, with aim to possibly make genetic operators work with Behavior Tree representation seamlessly. Developing more sophisticated methods of crossover and mutation, aware of constraints of Behavior Tree encoding, would doubtlessly increase the effectiveness of evolving trees. Furthermore, the concept of adaptation of the agent to a given environment should be explored. The task considered in this thesis - arguably simple - has ben proven to be solvable with the simpliest of elements. A natural next step would be increasing both the dificultness of the problem as well as the resources available to the algorithm.

Bibliography

[1] R. Axelrod and W. D. Hamilton. *The Evolution of Cooperation*. 1981.

[2] P. van den Berg and F. J. Weissing. "The importance of mechanisms for the evolution of cooperation". In: *Proceedings of the Royal Society of London B: Biological Sciences* 282.1813 (2015).

[3] A. J. Champanard. *Behavior Trees for Next-Gen Game AI: an on-line presentation*. `http://aigamedev.com/insider/presentations/behavior-trees/`.

[4] *FRAIL Official Website*. `http://frailpwr.github.io/`. Accessed: 2015-08-15.

[5] P. J. Hoen et al. "An overview of cooperative and competitive multiagent learning". In: (2005), pp. 1–46.

[6] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[7] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[8] C. Lim, R. Baumgarten, and S. Colton. "Evolving behaviour trees for the commercial game DEFCON". In: (2010), pp. 100–110.

[9] S. Luke. *Essentials of Metaheuristics*. second. Available for free at http://cs.gmu.edu/∼sean/book Lulu, 2013.

[10] S. Luke et al. "Mason: A multiagent simulation environment". In: *Simulation* 81.7 (2005).

[11] L. Panait and S. Luke. "Cooperative multi-agent learning: The state of the art". In: *Autonomous Agents and Multi-Agent Systems* 11.3 (2005).

[12] K.Y.W. Scheper. *Behaviour Trees for Evolutionary Robotics: Reducing the Reality Gap*. 2014.

[13] *The AI Sandbox*. `http://aisandbox.com/`. Accessed: 2015-08-15.