Cooperation Evolved

Behavior Trees evolution by means of Genetic Programming

Miłosz Mazur

September 5, 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, developed in John Holland's Adaptation in Natural and Artificial Systems (1975), have been adapted for use in a vast variety of different problems, including optimisation 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, popularised through works of John Koza (most notably Genetic Programming: On the Programming of Computers by Means of Natural Selection, 1992).

This project is an attempt to combine the two subjects, resulting in a way to automatically produce well-performing behavior trees, ready to use when autonomy and certain performance level consistency is required.

The environment used to conduct the simulations will be based on FRAIL, an in-house project of Wroclaw's Univeristy of Technology, adapted for the purposes of genetic calculations.

Acknowledgements

Contents

1 Introduction

1.1 Thesis Goal

The main goal of the thesis is to answer the question: "Can we teach artificial agents to cooperate by forcibly putting them in conflicting situations?"

1.2 Thesis Research Goals

- Propose a way to deal with bad (botched) trees that are not going to produce an action each time they are evaluated.
- Explore the task of making GP operators content-aware.
- Measure and find a way to deal with the impact of the initial generation.

2 Literature Review

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 varius internal contructs. 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 [4]. 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 Begavior 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.
- Additionaly, **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 relativerly simple to work while maintaining versatility, flow-control nodes are possible to employ. These come in many varieties and often can be said to

mimick (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 [4]. 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 immidietly when one of its children return Success and Failure only when all of its children return Failure. Analogically, Selector is a BT counterpart to logical OR function.

Finally, **Parallel-One** and **Parallel-All** nodes introduce concurence 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 Execution 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 [4]):

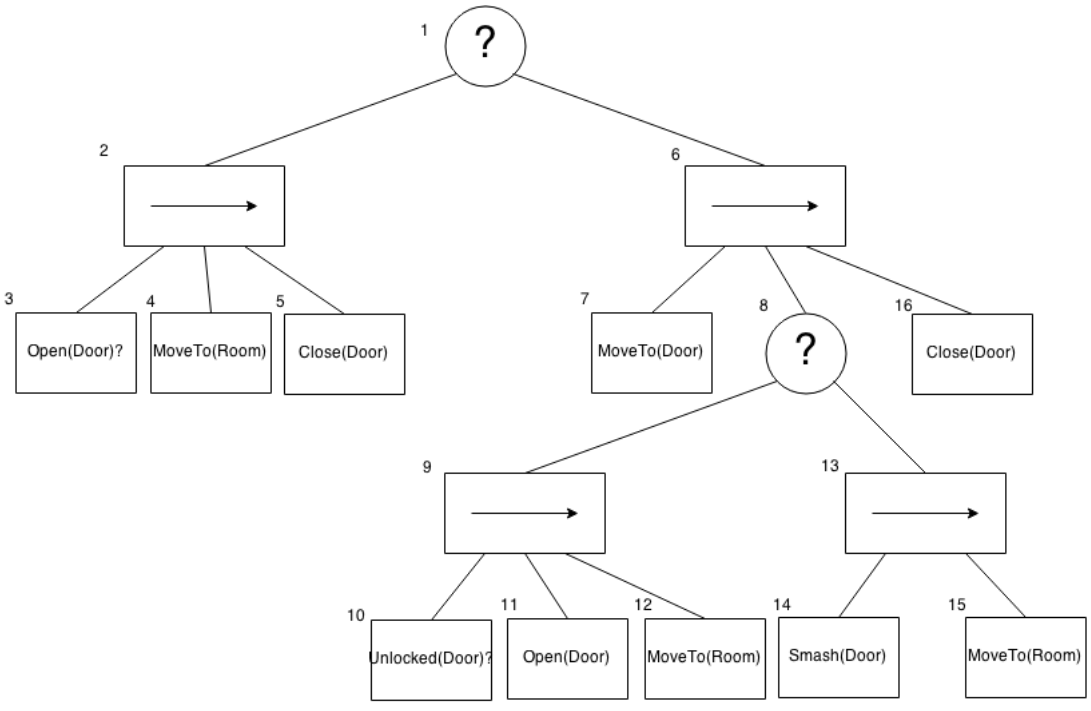


Figure 2.1: An example Behavior Tree

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 opening the door and closing them (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, desigining 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 du 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)[2]:

> (...) 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 differnet *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 seach space. [3]. With all steps finished, the whole process (sans the random initialisation 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 [2]):

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.

That being said, Selection, Crossover and Mutation strategies have a great deal of influence over the algorithm's result and for that reason, they 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 [3].

In **Fitness-Proportionate Selection**, also known as **Roulette selection** specimen are selected in proportion to their fitness: solutions with higher fitness value associated have a bigger chance of being seleced. Not only does it introduce an element or chance, in contrast to methods like **Elitism** or **Truncation Selection**, 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:

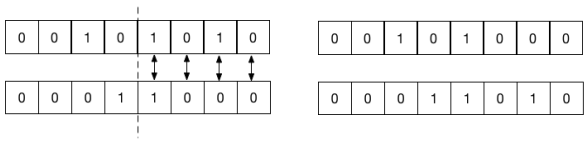
> They presume that the actual fitness *value* of an individual really means something important. But often we choose a fitness function such that higher ones are "better" than smaller ones, and don't mean to imply anything more.

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

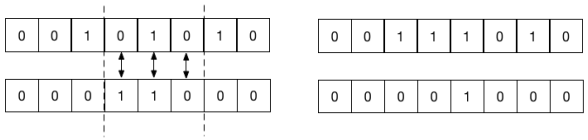
Tournament Selection deals with that problem in a curious way - instead of considering the fitness function values, it merely takes into consideration the ordering. The algorithms is definitely on the simpler side: a random sample of the population is chosen, and the best individual in that sample is 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 [3]. 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 inidividuals 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 ther genes in order to form two offspring (although a research by Eiben et al., 1994 mentioned in [4] 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 (or points, in latter case) along each speciman 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) One-Point Crossover



(b) Two-Point Crossover

Figure 2.2: Crossover Example

a destructive force. Namely, it has to do with *linkage* (or *epistasis*) between elements in specimen representation [3]. 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 a 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.

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 [2]:

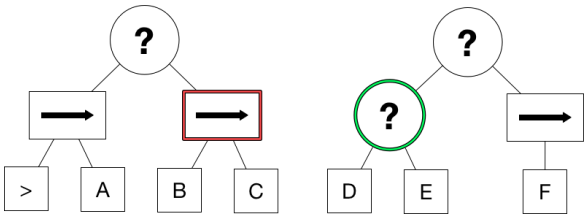
> The genetic programming paradigm continues the trend of dealing with the problem of representation in genetic algorithms by increasing the complexity of the structures undergoing adaptation. In particular, the structures undergoing adaptation in genetic programming are general, hierarchical computer 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

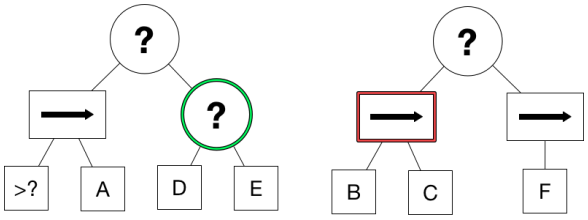
Apart from the issue of representation, Genetic Programming is, in its essence, merely an adaptation of Genetic Algorithm, with further changes to accomodate handling of a different data structure. As such, only said changes will be described here, while the remaining parts of the algorithm are assumed 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 Algorihm counterpart, selecting a random node in each parent and exchanging them (along with their respective subtrees). Figure 2.3 illustrates two specimen before and after crossover procedure.



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



(b) Step 2: performing the exchange.

Figure 2.3: 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 2.4 and 2.5 present an example of performing micro- and macromutation.

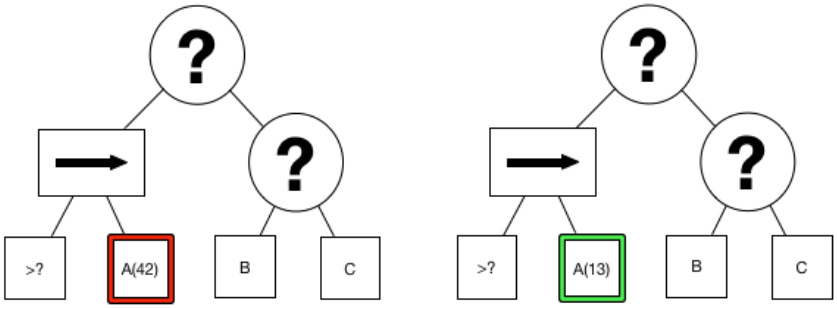


Figure 2.4: Micromutation: choosing a random element in a tree and changing its properties.

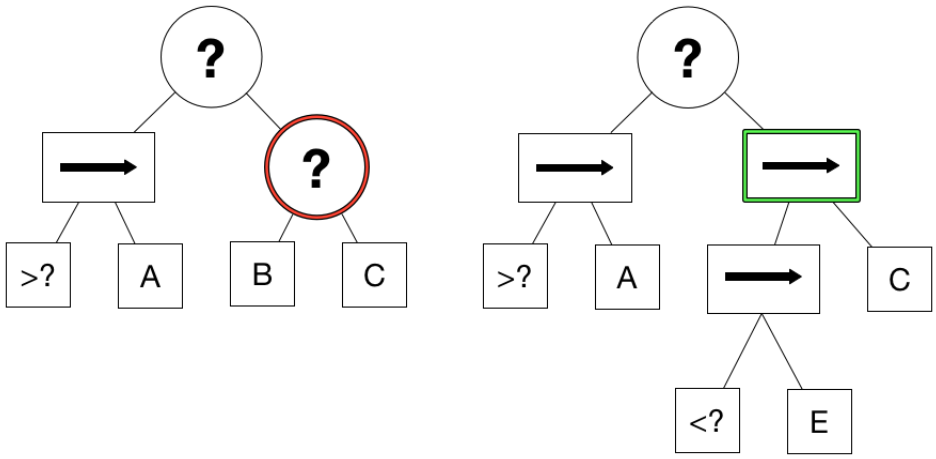


Figure 2.5: Macromutation: after a random node is selected, a new tree is generated and its contents replace selected node.

3 Environment

3.1 Introduction

FRAIL (Framework for AI Laboratories) is a platform developed in-house at Wroclaw's Univeristy of Technology (PWR) for purposes of testing artificial adversaries in shooter games. [1]

As an in-house project, users were able to freely access and change the source code, modifying existing contollers 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.

3.2 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 3.1 presents a 3D render of a finished map, while a top-down view, featured on Figure 3.2 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 secion, 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.

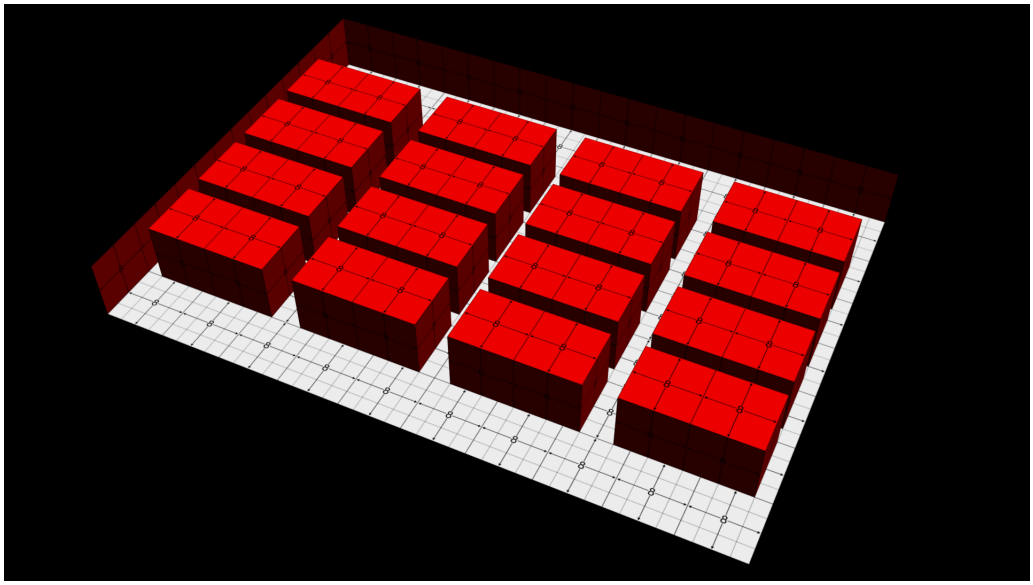


Figure 3.1: A rendering of scenery

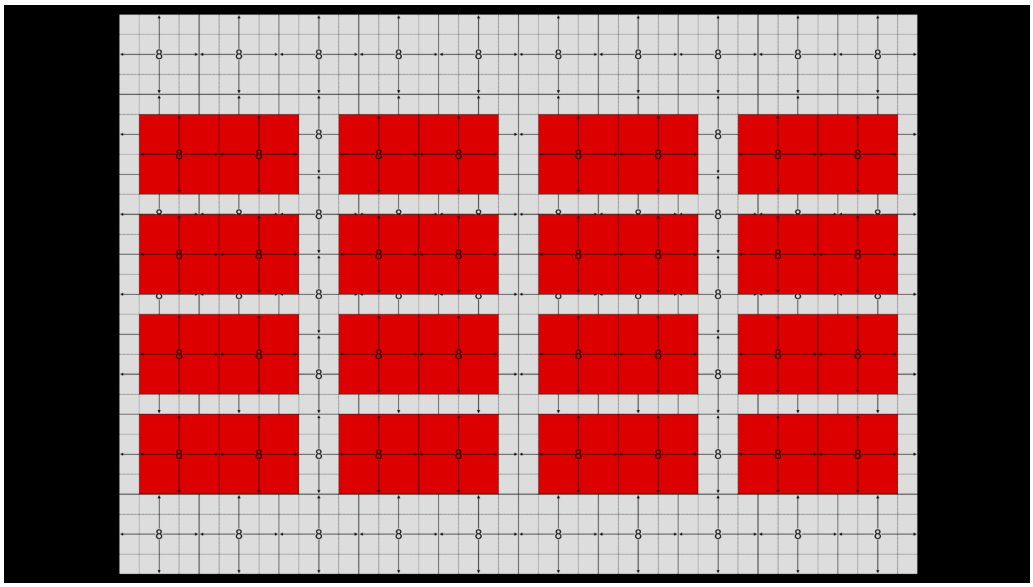


Figure 3.2: A top-down view of the map

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 endavor must be kept in mind. It would be beneficiary for a number of use cases to obtain a system whch, 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 unfesible 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 it against each other. As mentioned before, scenario type chosen for this was a resource gathering.

The premise of this particular scenario was as follows: a number of 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 said marker 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.

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

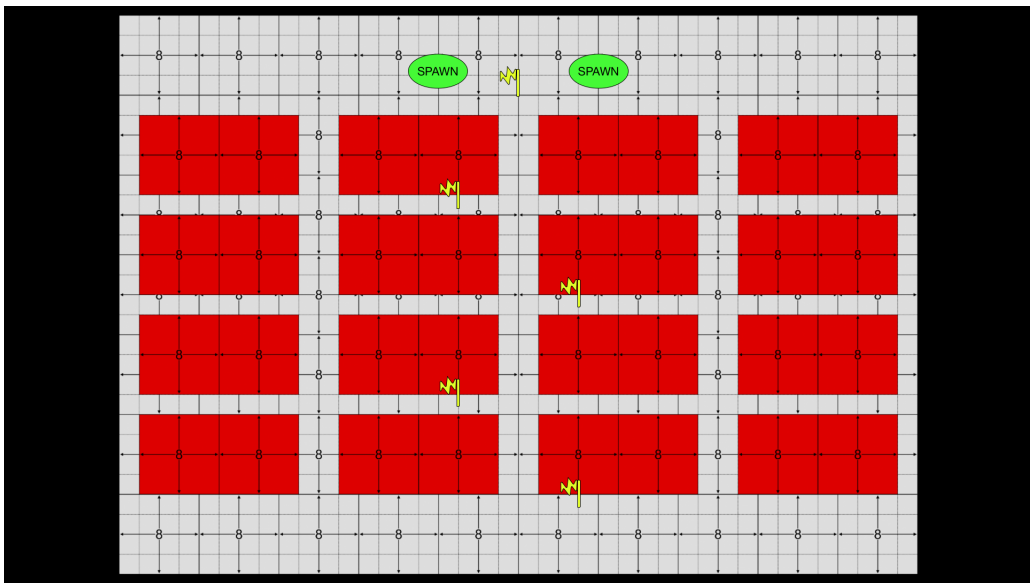


Figure 4.1: Initial positioning of agents' starting points and resource markers.

4.3 Introducing Genetic Programming

The Genetic Programming module was built on top of FRAIL's existing features, substantially changing the ususal flow of execution to make it suitable for learning processes. Initialisation, 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 then 48 hours to complete. A decision was made to increase update frequency of the simulations tenfold, thus guaranteeing simulations would complete afer 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), a specimen are instances of a specialized class, separated from the actual Behavior Tree implementation. These trees are then parsed during evaluation step to an actual 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 Operators

Initialisation

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 implemented as a exchange of randomly selected nodes (one from each tree) without minding their type nor location.

Mutation

Both available mutation methods 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 format of the thesis, used fitness function varied in different testing scenarios. Detailed explanations of each will be included in the appropriate sections of Results chapter.

5 Results

6 Conclusions and Future Work

List of Figures

Bibliography

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

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

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

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