Behaviour Tree Evolution by Genetic Programming

– Learning Novel Bot Behaviours in a 2D Top-Down Arena Shooter –

Martin Estgren <mares480@student.liu.se>
Erik S. V. Jansson <erija578@student.liu.se>

TNM095 – Artificial Intelligence for Interactive Media
at Linköping University, 20th March 2018

– Repository Link • Video Link –

Abstract

Behaviour trees are a popular model for representing the decision-making and plan execution process for NPCs in video games. These are built by hand, and often require expertise to craft into interesting and intelligent behaviours. These types of behaviour trees usually don't adapt well to new environments, and need many tailor-made trees for each situation.

In this paper we demonstrate how to generate the trees by using genetic programming; allowing us to essentially evolve novel behaviours automatically in our testbed. Instead of specifying low-level actions and conditions, we use high-level definitions. This leads to faster fitness convergence, and allows us to skip the bloat control and tree pruning steps used by other similar behaviour tree evolution methods. Results show that the evolved trees reliably defeats our hand-crafted behaviours by the 25^{th} generation.

Contents

1 Introduction

In interactive media such as video games, there is often a need for simulating seemingly complex agent behaviours in (soft) real-time. In a modern fast-paced computer game for example, the render, physics and artificial intelligence updates all need to complete in a total of less than 16ms to provide a half-decent experience. These resource constraint have spawned some very clever techniques to enable complex behaviours for autonomous agents while minimizing their required computational time.

These techniques often require the bot behaviours to be manually defined, and in some cases, are quite hard to extend when new behaviours are needed later in the game development process. This has been shown to be true by industry researchers, such as *Dawe et al* [4], regarding the *finite state machine architecture* (which we talk more about later).

In this project we have explored how one of the more widely used techniques, *behaviour trees*, can be extended to allow for not only hand-crafted complex behaviours, but for organic behaviours, tailored to the specific game domain by using a *reinforcement learning* approach. More specifically, we evolve the behaviour trees by using *genetic programming*. It's by no means a novel approach, but there still

has been relatively little research in the area. In some previous work, such as in *Ögren et al.* [3], this method of pairing behaviour trees with genetic programming is called *behaviour tree evolution (BTE)*.

Our work was initially inspired by and based on the research of *Ögren et al.* [3], but our approaches diverged later on in the project for a couple of reasons. First of all, the domain in *Ögren et al's* work allows for relatively simplistic agent actions and conditions (it's based on the *MarioAI* project), which isn't suitable for our domain since the actions and conditions our agent needs are a lot more complex. Therefore, it wasn't viable to have *low-level actions* with high granularity (e.g. move left, right or jump), but we instead needed *high-level actions* (e.g. move to the enemy), otherwise the behaviour tree would quickly grow too deep and become unmanageable. The second reason we diverged from *Ögren et al's.* work was because their method needs *tree pruning* and *bloat control* to remain viable, and we wanted our method to remain simple to implement in code.

Therefore, our method differs from *Ögren et al.* [3] in the aspect that we specify actions and conditions using a high-level specification instead of a low-level one. This allows us to skip the pruning and bloat control steps in *Ögren et al.* since the behaviour tree tends to converge faster and also not grow too deep, which gives us a somewhat simpler implementation as well. Another difference is that our selection, crossover and mutation steps in the GP are somewhat different than in *Ögren et al.*, and that we also didn't need to use simulated annealing as a pre-process step to get a good generation diversity.

Alongside the proposed method in Section 1.1, a testbed consisting of a top-down arena shooter was built. It features a sufficiently complex environment to support interesting behaviours by the enemy bot. We describe the features of our testbed in Section 4.

This report starts off in Section 2 by giving a brief overview of the relevant techniques and related work necessary to follow our reasoning and findings. In Section 3 we describe the details needed to implement these techniques in practice, along with the testbed architecture we have used. This gives the reader information about its potential pit-falls. We follow this by showing in Section 4 the testbed, and give some generated offspring generated by our method along with measurements of its efficiency. Finally, we reflect and give downsides of our method, and present directions for future work in Section 5.

1.1 Proposed Approach

The primary approach during this project is to examine how *genetic programming* applied to the generation *behaviour trees* can serve as a substitute for behaviours hand-crafted by game designers.

2 Background Theory

In this chapter we briefly describe the theory behind the three techniques used in this project: *behaviour trees*, *path finding* and *genetic programming*. These techniques are only briefly described here to make the report self-contained. However, we also give many references in the bibliography for that readers that want to read these topic more in-depth.

2.1 Behaviour Trees

Defining an accurate *automatic planner* model is often impractical and usually overkill for real-world applications. Especially in games, where very simple models are enough to give the illusion of intelligence. In games, the most well-known *behaviour selection algorithms* are *FSMs*, *GOAPs*, *HTNs* and *BTs* [4].

Finite State Machines (FSMs) are simple but hard to extend with additional states, resulting in an exponential increase of transitions [4]. While both *GOAP (Goal-Oriented Action Planner)* and *HTN (Hierarchical Task Network)* are powerful models, these don't allow agents to explore new task definitions, and "only" provide an agent with novel ways of solving an already defined set of tasks. Planning has seen some use in the games industry. Most notably in the game *F.E.A.R* [11] where adversarial agents used a planning system to combat the player.

Behaviour Trees (BTs) on the other hand, when comparing to FSMs, provide advantages in terms of modularity, reactiveness and scalability. And more importantly, since they are a type of tree, allow for integration with the genetic programming approach. They were first introduced in *Halo 2* [7], but are now applied in other area as well, such as robotics.

Behaviour Tree Evolution (BTE) is a technique which uses some form of genetic programming on BTs. It's a relatively new technique that has been applied to *strategy games* and *platformers*. Some of the previous work is due to *Lim et al.* [8], *Colledanchise et al.* [3] and a recent thesis by *Hoff et al.* [6].

In the *Behaviour Tree Starter Kit (BTSK)* [2] and in *Chris Simpson's Gamasutra article* [13], the BT is a tree-like data structure which describes the decision-making process of the agent. It's evaluated pre-order from the root of the tree, in each logic update tick. There are many other ways to define BTs, but the one above is the most common one, and is very similar to the original, fairly informal one, in the GDC '05 Halo 2 talk by *Damian Isla* [7].

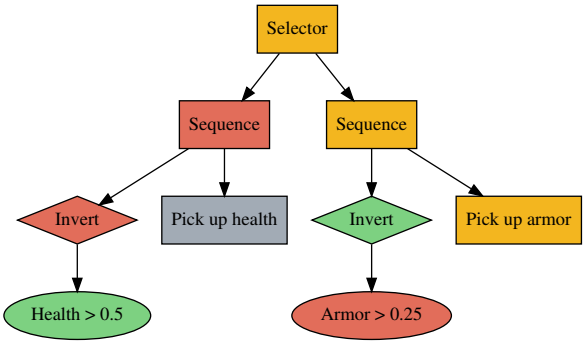


Figure 1: Example of Hand-Crafted Behaviour Tree

In the coming pages we attempt to summarize the most common types of nodes in a behaviour tree, in particular those found in the *BTSK* [2]. There are two types of nodes, *concrete nodes*, and *abstract nodes*. *Concrete nodes* are *leaf-nodes*, and are suspended at the bottom of the BT. Furthermore, concrete nodes are divided into two types: *action nodes* which allows the agent to change the world state, and *condition nodes* which allow the agent to query the world state. On the other spectrum are *abstract nodes*, which can either be *composite nodes* (which usually deals in the control-flow of the BT, by changing the order in which its children are executed) or *decorator nodes* (which modify existing child nodes).

Each node has a state attached to it: *RUNNING*, *SUCCEEDED* or *FAILED*. If its *running*, then this means the current action or condition hasn't been completely yet (this in turn also affects the parent node), but will eventually complete. If, once completed, the node *succeeded* or *failed* (according to some pre-defined convention), then the node's state becomes set to that (*FAILED* or *SUCCEEDED* state). We now summarize the common node types:

- **Selector (composite node):** evaluates its child nodes from left to right, until *one of the children* is *RUNNING* or has *SUCCEEDED*, and inherits its state. Otherwise, the selector receives a *FAILURE* state.
- **Sequence (composite node):** evaluates its child nodes from left to right, until *one of the children* is *RUNNING* or has *FAILED*, and inherits its state. If all children succeed, then the sequence also *SUCCEEDS*, otherwise it *FAILS*.
- **Parallel (composite node):** evaluates its children concurrently and can be specified to either behave as a *Selector* or a *Sequence*.
- **Inverter (decorator node):** evalutes its only child node and *inverts the resulting state once it leaves* the *RUNNING* state. i.e. *SUCCEEDED* state becomes a *FAILED* state, and vice-versa.
- **Succeeder (decorator node):** regardless of what the child's state is once it has *finished RUNNING*, we return only a *SUCCESS* state.
- **Condition (leaf node):** this node queries the world state, and if the query returns a positive answer (e.g. our agent has bullets for his weapon), we return a *SUCCESS*, if it's negative, we give a *FAILURE*. This node is usually instantaneous, and thus can't be in a *RUNNING* state.
- **Action (leaf node):** modifies the world state, and returns a *SUCCESS* if the action had the intended side-effect (e.g. the agent is shooting at the enemy), or returns*FAILURE* if it couldn't complete the action for some reason. If we are still performing the action on the world (e.g. aiming the gun), then the node is still *RUNNING*.

We find that it's easier to understand behaviour trees with an example. We'll be using the tree in Figure 1 as an example (it was produced within our testbed using a Graphviz library for Java). The *yellow* nodes means the node state is *running*, the *red* nodes have *failed*, and *green* nodes have *succeeded*.

At a high-level, this BT checks the agent's health and armour, and if it's below a threshold, it will go and pick up the crates which replenish these stats.

It does this by starting at the root node, *selector*, and executing the left child, the *sequence*. It will execute its left child, which is the *inverter*. At

last we reach a leaf-node, the *health condition node*. This will probe the world for the state of the agent, and return *success* because we've found that it's above 50%. This result will propagate upward to its parent, the inverter, which will set its state to *failed* (inverting the result of the child). We're back at the left sequence, and because one of the children have failed (the inverter), the sequence has *failed* as well, and won't run the right child (gray means the node has never been evaluated). Finally, because the selector's left child has *failed*, we execute the right branch too (remember, at least one child must succeed!). We use the same steps as before, but get stuck in *"pick up armour"* until that action has finished *running* in the world (has succeeded or failed).

2.2 Path Finding with A*

Agent path finding is done using the $A*$ graph traversal algorithm proposed by Peter E. Hart, Nils J. Nilsson, and Bertram Raphael [5]. This algorithm is an extension to the *Dijkstra's algorithm* which, given an *admissible heuristic* and non-negative costs, finds the path from a node n_0 to a node n_g with the lowest cost.

Calculating the expected cost of a path $n_0 \rightarrow n_k \rightarrow n_g$, from n_0 to n_g via n_k can be done with the following path finding cost function:

$$f(n_k) = g(n_k) + h(n_k)$$

where $g(n_k)$ is the true cost from n_0 to n_k and $h(n_k)$ is a heuristic approximation of the cost from n_k to n_g. This allows us to explore the potentially optimal paths first before considering the others.

In this project, the *Euclidean distance* is used as the heuristic $h(n_k)$. This, since the path finding is performed in a search-space where each node is defined as a point in the testbed's 2-D world-space.

Additionally, compared to regular $A*$, the path-finder utilizes an *influence map* to help the agent seem more "aware" of it's surrounding, even if the selected path is optimal given the heuristic function. Our *influence map*, like the ones shown in *Dave Mark* [9], take into account the visibility of each node in relation to the adversarial game agents' line-of-sight, and therefore updates the expected cost as:

$$f(n_k) = g(n_k) + h(n_k) + i(n_k)$$

where $i(n_k)$ is the cost introduced by the influence map. This gives the the result that nodes within the adversarial agent's line-of-sight have a much higher associated traversal cost, and should be avoided.

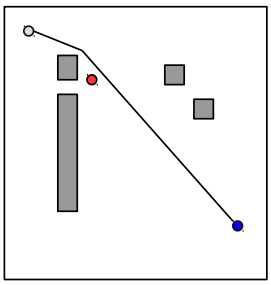


Figure 2: optimal path

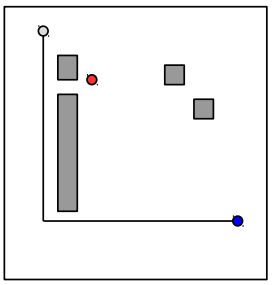


Figure 3: tactical path

Figure: Figure 2 shows an optimal path using regular $A*$, and figure Figure 3 shows the optimal path when taking into account the influence map; sneaking behind cover.

2.3 Genetic Programming

In many scenarios you try to optimize a function: $f(x_0, x_1, ..., x_n)$, where some or all of the inputs are of a discrete nature (ordinal or nominal values). These types of problems are often hard to optimize using techniques developed for continuous variables, such as *gradient ascent/descent*, e.g. in a *perceptron*.

Genetic programming is a technique that was introduced by *Barricelli et al* [1], as a way to optimize computer programs by encoding the parameters to be optimized as *genetic representations* which could be processed by traditional *evolutionary algorithms*.

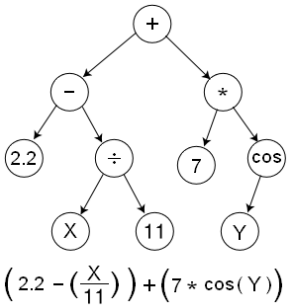


Figure 4: The *genetic representation* is often done using a tree-like structure in *genetic programming*. Image is taken from: `Wikipedia` (public domain).

The most general form of the *evolutionary algorithm* can be described in the following steps:

1. First generate an initial population by creating a initial pool $\mathcal{P}_0$, of individuals with randomized genetic representations (context-specific data).
2. At a given generation, evaluate the *fitness* for each individual $x_i \in \mathcal{P}_t$. This usually requires a domain-specific *fitness function*: $f(x_i) \rightarrow f_i \in \mathbf{R}$.
3. Remove individuals with *low fitness score* and *generate* new population : $\mathcal{P}_{t+1} = \Phi(\mathcal{P}_t)$ based on *proportional fitness selection*.

The fitness function for this project is composed of a *linear combination* of desirable behaviours collected during the *fitness evaluation* simulation, such as, amount of *damage dealt* to the adversary, whether the agent with a specific BT *survived*, and how many *weapons the agent has picked-up*.

Regeneration of the population pool for the next generation, $\mathcal{P}_{t+1}$, is done using the three functions:

Selection $\Phi_s(\mathcal{P}_t, \mathbf{R}) \rightarrow x_i \in \mathcal{P}_t$ individuals with high *fitness score* in the new population $\mathcal{P}_{t+1}$. In this project, *selection* is done proportionally to the *fitness score* for each individual. Each individual x_i has a probability $P(x_i|f_i)$ to be selected for the next generation $\mathcal{P}_{t+1}$ according to some random process $\mathbf{R}$.

$$P(x_i|f_i) = \frac{f_i}{\sum_{j=1}^{N} f_j} \text{ where } N = |\mathcal{P}_t|$$

Crossover $\Phi_c(x_i, x_j, \mathbf{R}) \rightarrow \hat{x}_i, \hat{x}_j$ individuals from the new population which have been crossed over, in the case of *genetic programming*, *crossover* is done by swapping random *sub-trees* in each BT. All sub-trees have an equal probability of being selected to crossover, creating new individuals in $\mathcal{P}_{t+1}$. $\mathbf{R}$ represents the random seed for the crossover process.

Mutation $\Phi_m(x_i, \mathbf{R}) \rightarrow \hat{x}_i$ a subset of the new population, but slightly tweaked with a random parameter (based on the random seed $\mathbf{R}$) or by adding/removing random child nodes. If no mutation is done, then the genetic programming might converge to some local optima because of monoculture, and never reach a global optimum.

Once these steps have been performed, a new generation of individuals have been produced, which can then be used to repeat the optimization process until we converge to a population with behaviours close the global maximum of the fitness function f_i.

3 Implementation Details

After presenting the theoretical background, we describe details specific to the concrete implementation for this project.

Both the top-down shooter game (target *testbed*) and our *behaviour tree evolution* (the *technique*) are written in *Java* and use the *libGDX* library for the entire simulation/visualization pipeline. This allows for quick prototyping, and while the game itself isn't the main topic of this paper, a rough outline can be found in Section 3.1 as it is necessary to understand for the main focus of this project.

Every *entity* in the game can be transformed into an intelligent agent with a `BotInputComponent`, which associates a given *behaviour tree* with said agent. It hooks together with the *path finder* to enable the entity to *traverse & interact* with the simulated environment. With *genetic programming*, additional BTs can be generated and used.

3.1 Game Architecture

Given the very limited implementation time of this project, the architecture of the game was designed to allow for fast prototyping iterations, rather than based on strong and scalable software design philosophies. The structural design is based on the *entity component system* design, which is highly popular use in the contemporary game industry, for example in the *Unity* game engine. The primary reference implementation is taken from the book *Game Programming Patterns* written by *R. Nystrom* [10] remixed with the component system from *M. Boströms' Speed Coding Zelda*[1]. The resulting implementation can be seen in the source code in the package: `se.sciion.quake2d.level` and its sub-folders.

An *entity component system (E-C-S)*, as the name suggest, consists of three interacting class types. The *entity*, the *component*, and the *system*.

The *entity* is defined as a class consisting of a series of *components*, together with functions for

[1] `https://github.com/MilleBo/SpeedCodingZelda`

querying information about the entities' components. This allows the entities' components to query their parent (the entity itself) regarding other components within this entity. Meaning, components that heavily depend on each other have high coupling, while independent components don't have any coupling. This gives us a very modular design, and gives the expressive power to easily add new components to an entity on-the-fly. There are several other ways to solve the problem of system-system communication, like message-passing, but we've decided to take the "easy way out" and just pass references around to the relevant objects.

Figure 5 shows three different types of entities in our game: *user-controlled players*, *AI-controlled agents* and *health/armour/weapon crate pickups*. Adding functionality to each entity is just a matter of adding/swapping a component. The only difference between a *player* and an *agent* is the method of input, *UserInput* vs *BotInput*.

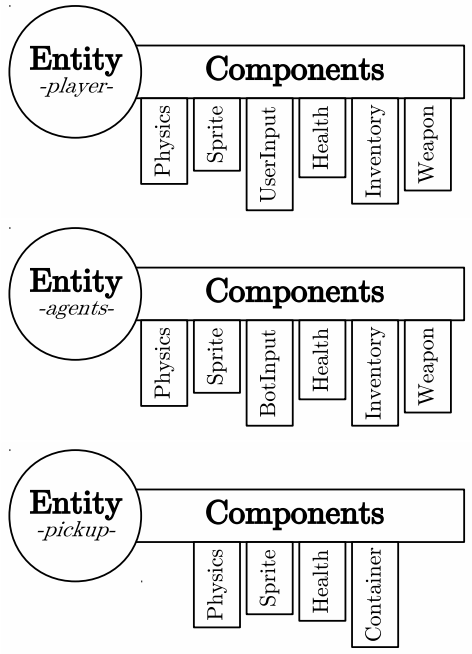


Figure 5: Entities with corresponding components.

The game mechanics which spans more than inter-entity communication are defined as individual components not bounded to any entity, so called *systems*. Example of such *systems*are *path-finding*, *physics*, and *gameplay*. As we touched upon before, these systems are passed as references to the components that are interested in them, allowing for the components to decide what to do with the data provided by the systems without caring about the system's implementation details.

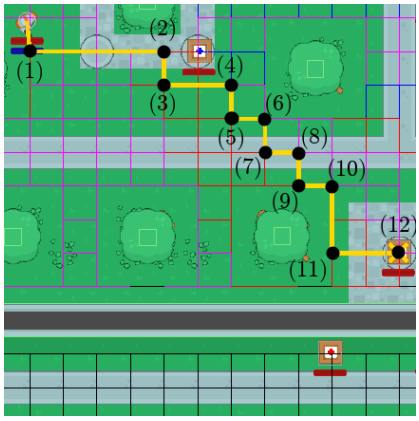


Figure 6: Visualization of the path finding stack $\mathcal{F}$. If there are no obstacles in the way (line-of-sight check), the agent takes a straight-line path instead.

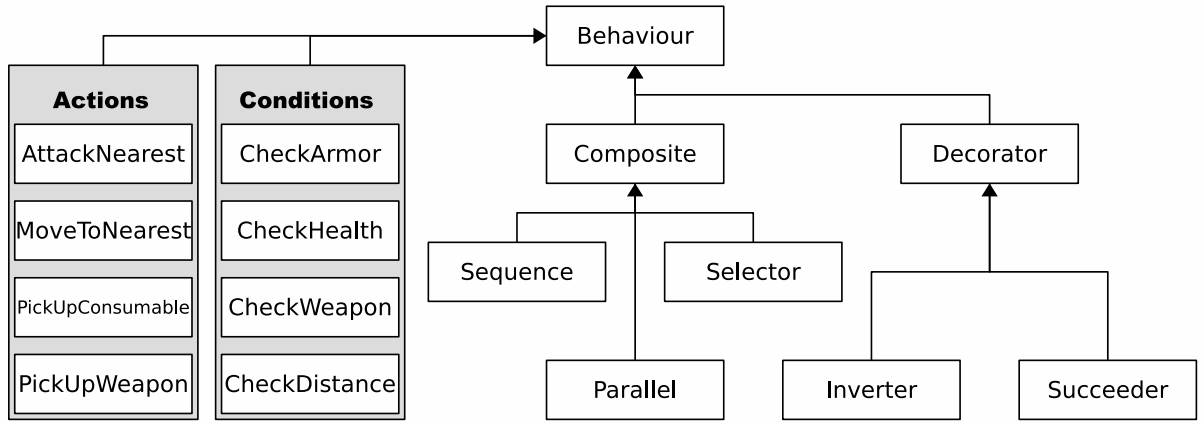


Figure 7: UML-diagram of the class structure for behaviour nodes.

3.2 Behaviour Trees

Inside of the *Java package* `se.sciion.quake2d.ai.behaviour` (can e found in the source code repository) you'll find the implementation of our BTs. They are specified using inheritance, as in the figure above. In our variant, only the leaves above are non-abstract.

A behaviour tree is represented as a tree of behaviours where each node contains links to their respective child nodes. The tree is evaluated in a recursive fashion which makes it possible for each behaviour tree to only care about the root-behaviour.

All nodes are derived from the class `BehaviourNode`, which provides the framework for each node. Each node is processed for each step (tick) of the simulation though the *tick* function.

Algorithm 1 Pseudo-Code for the BT's *Tick* function

Require: some initial *State* to the behaviour tree
  if $State \neq Running$ **then**
    $doBehaviourEnter()$
  end if
  $State \leftarrow doBehaviourUpdate()$
  if $State \neq Running$ **then**
    $doBehaviourExit()$
  end if

The tree is evaluated recursively by the root-nodes *tick* function being called, which in turn calls the *tick* function in its children.

The composite nodes already defined have enough expressiveness to allow for very complex modelling. This means that the most focus is put on lead-nodes which models concrete actions and conditions highly coupled to the specific game mechanics used in this project. We've adopted a fairly simple solution to communication between leaf-nodes and game mechanics, where the leaf-node is given a reference to systems that it needs communication with. The leaf-nodes also have a reference to the entity owner of the input component which the specific behaviour tree is attached to. This reference provides a interface for querying other entity components for entity specific data such HP, armour, or carried weapons.

For example, if a BT wants a *AttackNearest* node beneath a *Succeeder*, we'd first have to hand over a reference to the *path finder* and a *target tag* to the node's constructor, such as: *AttackNearest(path-finder, tag)*. Only after that are we allowed to execute: *SucceederNode(attack-nearest)*, which we can utilize to construct the next parent layer above it.

Below is a list of all actions and conditions we've implemented in our game, along with a description:

- **AttackNearest(*tag*):** finds the closest entity with *tag* and *path find* to it. Shoot with weapon. *Success* if we were able to fire, *false* otherwise.
- **MoveToNearest(*tag*):** finds the close entity

with *tag* and *path find* to it. Stops when close. *Success* when we're close, *fails* if no valid path.

- **PickUpConsumable(*type*):** finds the closest consumable of *type* (*health*, *armour* and *boost*), *path find* to it and *pick it up*. *Fails* if it couldn't.
- **PickUpWeapon(*type*):** finds a close weapon of *type* (*shotgun*, *rifle* & *sniper*). *Path find* to it & try to *pick it up*. *Succeeds* when picked up.
- **CheckStatus(*ratio, type*):** checks the entity owner's *status* (*health* or *armour*), and returns *Success* if above the *ratio*, or *fail* if its below.
- **CheckWeapon(*type*):** checks entity owner's *inventory* for weapon *type*. *Success* if it gets it.
- **CheckDistance(*tag, r*)** checks if entity with *tag* is above r units away. *Succeeds* query if so.

3.3 Genetic Programming

Since the genetic programming has to be done on the trees during runtime, we need a way to dynamically generate trees with varying structure (changing composite nodes) and behaviours (changing leaf nodes). The solution used in this project is to create a set of *prototype behaviour nodes* $\mathcal{S}$, one for each node type. This set can then be used to create the trees by building them from the root. The prototypes are implemented using the *prototype design pattern* as presented by *R. Nystrom* [10], meaning, we have a set of correctly pre-specified instances which are then cloned & modified to suit our needs.

Tree Generation

Each tree starts by first generating a random node $n \in \mathcal{S}$ as the *tree root*, and if the node is a *composite*, continue recursively until a complete tree is formed.

All nodes have a *randomization function* which for each node type randomizes it in different ways. *Leaf nodes* have a trivial randomization function which only tweaks parameters such as *thresholds* (e.g. 0.5 or 0.2) and *game tags* (e.g. *player* or *crate*). *Composite nodes* are randomized by picking a random number between 1 and 6 which represents the number of children. Each child is picked randomly from all possible behaviours. Finally, each child is randomized.

Tree Mutation

As with *tree randomization*, the *mutation of trees* is done in a recursive manner, where each node mutates itself, and in the case of composite nodes its children. The mutation only differs from the tree randomization in that nodes are only slightly changed with a certain probability, whereas the randomization can completely change all nodes in a tree. Figure 8 and Figure 9 shows the mutation of a *leaf* and *composite* node, respectively.

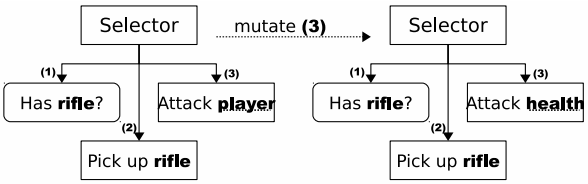


Figure 8: Mutation for the Leaf Node

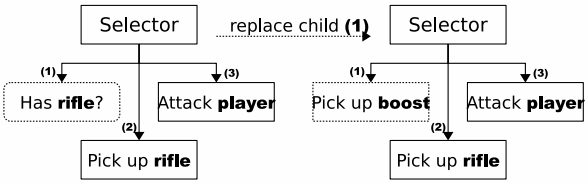


Figure 9: Mutation for Composite Node

Tree Crossover

During *tree crossover*, the tree is flattened to a list. This enables us to *select two random nodes*,where each node in the trees has the same probability to be selected, from *two different trees*, and swap them.

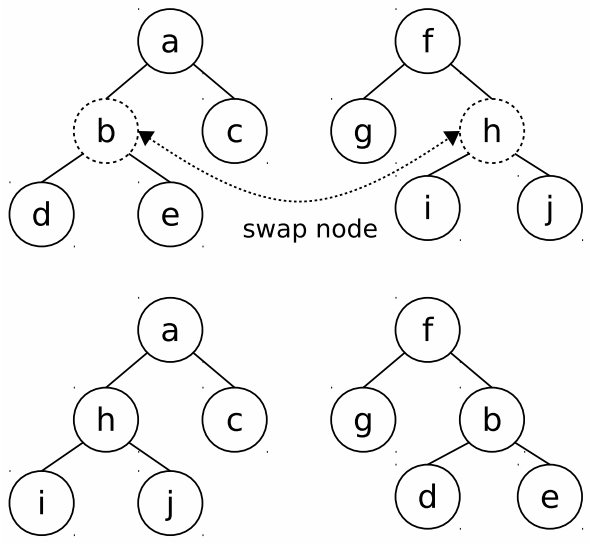


Figure 10: Crossover between two trees.

Fitness Function

As alluded in the theory section, the fitness function f_i is defined as an arbitrary linear combination of game related parameters. The full implementation is described in pseudo-code on the next page.

Algorithm 2 Pseudo-Code for the fitness function

Require: *damage* being the damage given
Require: *armor* being the armour picked-up
Require: *weapon* to be the weapons picked-up
Require: *kills* to be the number of agents killed
$Score = 5 * damage + 2 * armor + 50 * weapon + 1000 * kills$
if Individual Won **then**
return $Score + 500$
else
return $Score$
end if

Since individuals can be tested more than once each generation, the final fitness score is averaged over all test. This allows for a more fair fitness evaluation since only testing each individual once might make some test unfair. For example a bad tree might score very high against an opponent which is broken and can't move.

4 Results and Screenshots

Below are a couple of in-game screenshots from *Quake 2D*, our testbed for *behaviour tree evolution*. Here we've chosen to show two bots playing against each other in the arena, each of them loaded with a basic hand-made behaviour tree shown in Figure 13.

As can be seen in Figure 11, the environment is composed of *static collidable walls*, *dynamic pick up crates*, *user/agent controlled players*, and *graphical scenery details* based on *tiled textures*. Maps are specified using the *Tiled* editor, which means we can build new environments to test our solution. We've created a serializer for our behaviour trees, that enables us to save and load behaviour trees to disk.

Additionally, we've developed several tools for showing our agent's behaviour in real-time. As can be seen, Figure 12, we're able to *visualize path finding information* for the intelligent agents. Here the *yellow line* is the *target path* specified by the *path finding* system given a *target location* (shown as a *yellow cross*) by the *behaviour tree*. In this example, the target is the *rifle pickup crate*. You'll see a coloured grid being displayed. This tells us something about the state of the *influence map*, where the *blue and red lines* are grids that each agent is able to shoot towards, while *magenta lines* are locations where both agents are able to shoot towards.

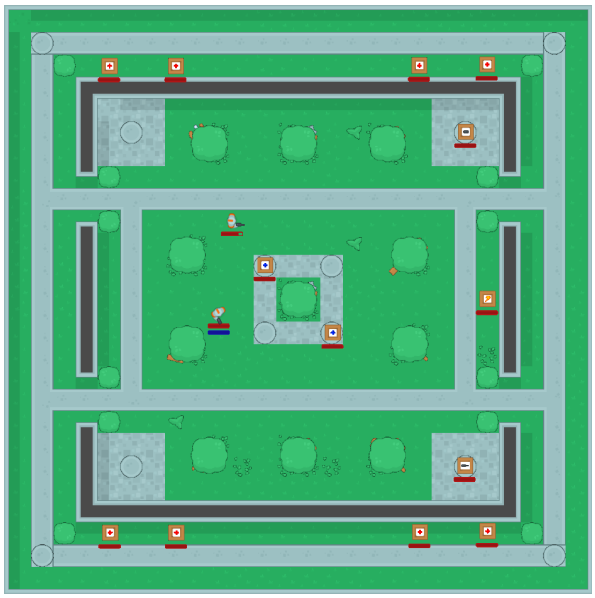


Figure 11: Screenshot of a Bot vs Bot in Quake 2-D

For debugging and analysing agent behaviour, we've implemented a way to visualize BTs, the one in Figure 13 from the Figure 12 situation, in real-time by using a *Graphviz*[2] library. It opens in a separate window and runs in another thread, where we can see instant changes in the BT's nodes' state.

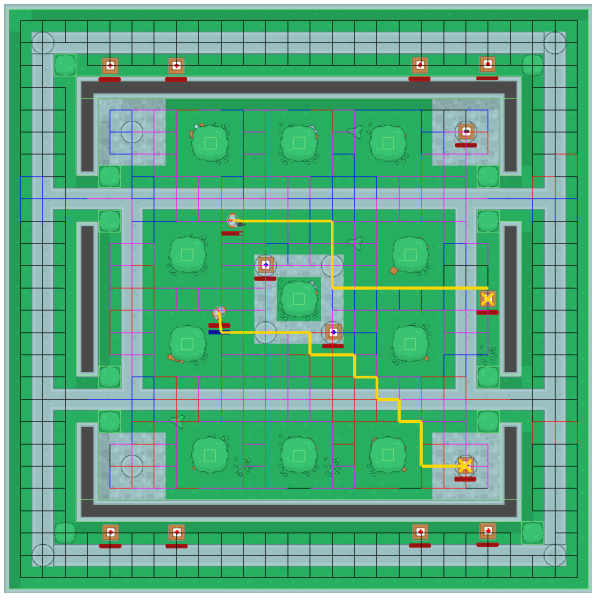


Figure 12: Screenshot of Path Finding in Quake 2-D

[2] https://github.com/nidi3/graphviz-java

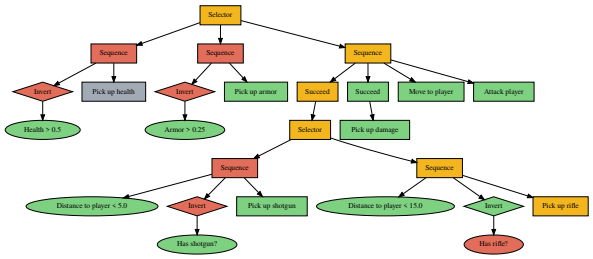


Figure 13: Hand-crafted BT (larger in Appendix A)

We've uploaded a video[3] of our solution in action.

Finally, we present the results from our solution. After generating fifty random individuals for $\mathcal{P}_0$, we applied genetic programming until we had $\mathcal{P}_{50}$, the resulting pool of *evolved behaviour trees*. The fitness for each individual in each generation was found by having a match against our crafted behaviour tree, and recording match statistics, used to calculate f_i.

Each individual (BT) has been evaluated for each generation by pitting them against other individuals in the same generation (we call them *peers* below). Therefore two results will be presented, one where the fitness f_i is evaluated against our *hand-crafted tree* and one against its *peers* in the same generation.

In Section 4.1 we provide two behaviour trees generated with our solution at $\mathcal{P}_{50}$ when playing against hand-crafted behaviour trees. We then give in Section 4.2 graphs of the evolution of the fitness function per generation, for both the peer and the hand-crafted evaluation methods from $\mathcal{P}_0$ up to $\mathcal{P}_{50}$. In Section 4.3 we measure the effectiveness of the behaviour tree by looking at the amount of individuals in the generation that have survived against our hand-crafted behaviour trees as the number of generation progresses forward.

4.1 Generated Behaviours

Each fitness evaluation was performed on a randomly selected hand-crafted level to prevent the behaviour trees getting overfitted to a specific level.

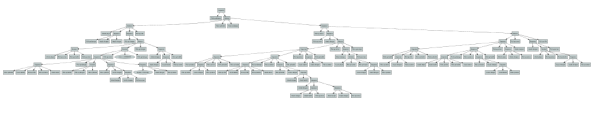


Figure 14: Generated tree with a very high fitness score (4750). This behaviour tree is 15 levels deep.

[3] https://vimeo.com/242262802

As can be seen in figure 14. This tree is way too big to provide any significant insight, unless interactively evaluated during run time. Therefore we've chosen not to provide a scaled-up variant of it. The fitness score is very close to the limit for the theoretical limit possible for the levels used during evaluation.

As the number of generations increase, the trees grow very large. This happens since no pruning is performed for the trees, neither as a post-process step nor in the disguise of reduced fitness score.

While the depth of this tree is unsatisfactory, the performance it achieves against our hard-coded bot is quite good. It is able to consistently beat it on all different map arenas. In fact, it's actually hard for even the authors of this paper to beat it consistently.

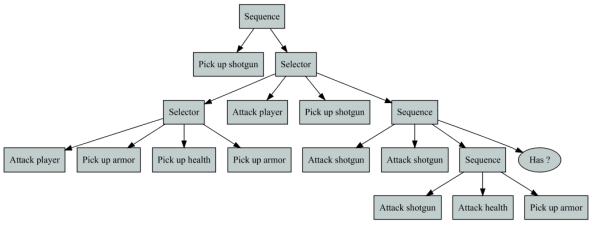


Figure 15: Generated tree with high fitness (3650). This tree on the other hand is only fours levels deep.

In figure 15 smaller tree of the same generation as the tree in figure 14 can be observed, it manages to defeat the hard-coded bot on all levels without bloating to unmanageable size. Prior exploration of generated trees indicated that as long as the tree has a *sequence* of *pickup shotgun* and *attack player*, it would reach a fitness of around 1000. Because this kind of sub-tree is quite common, it tends to appear in many BT crossovers.

Since this behaviour tree is a lot smaller, it's quite easy to analyse the expected behaviour in comparison to the one in Figure 14. At the end of this paper we've therefore provided a scaled-up version in Appendix B so the reader can look through it in more detail.

4.2 Behaviour Fitness

Looking at the general fitness over generations, we can observe that once the behaviour trees routinely starts defeating its adversary, the mean fitness plateaus and the variance fills the entire range of possible fitness values. When evaluating each tree against random peers, we see something similar but it converges with a slower rate. One possible hypothesis of why this happens is that the amount of "junk" behaviours that get by in Figure 17 at the start are all low-performing BTs. Therefore, there is no real competition at the start until one of them gets a good mutation or crossover. After that, this better BT will very likely pass on its genes onto the next generation and generate a lot better BTs than before. In our hard-coded behaviour tree, this happens quickly because the competition at the start is already quite high.

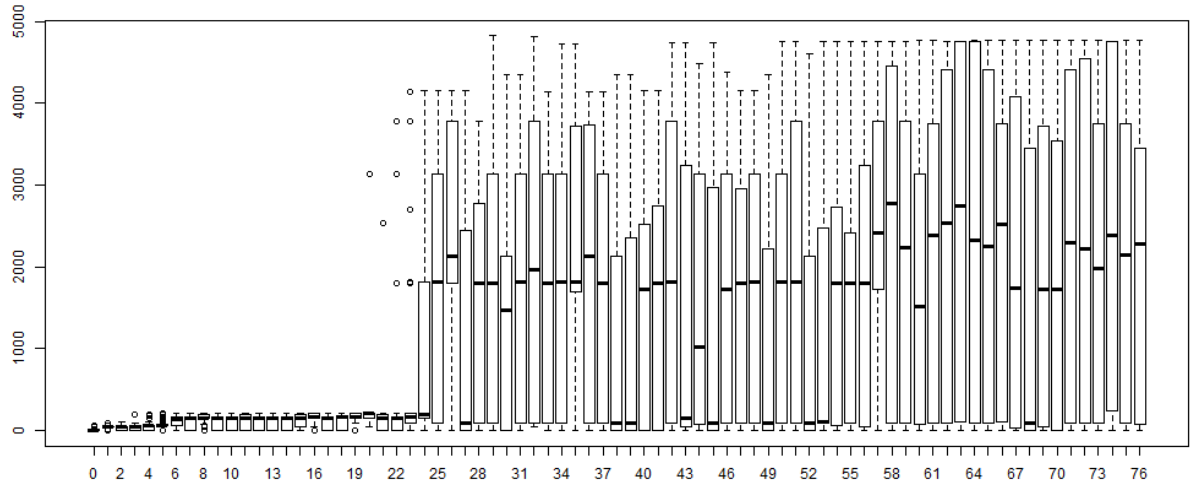


Figure 16: Fitness over generations when evaluating against a hard-coded bot. Higher score is better. The solid bar represents 50% of the fitness variance. The bar in each box represents the mean value and isolated points represents significant outliers.

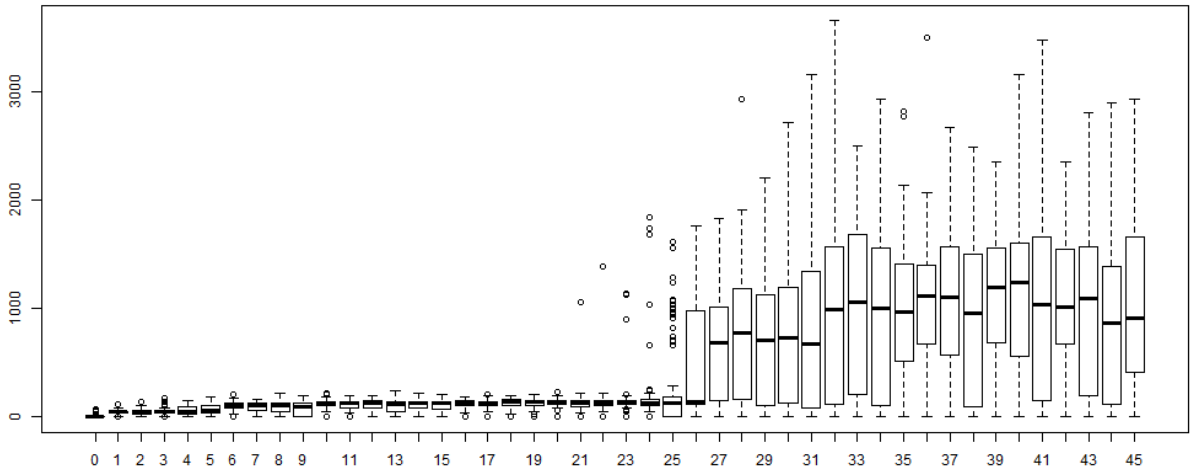


Figure 17: Fitness over generations when evaluating against 2 random peers. Higher score is better. The solid bar represents 50% of the fitness variance. The bar in each box represents the mean value and isolated points represents significant outliers.

4.3 Survival Ratio

In the ideal case, the survival rate follows as a consequence of the fitness score. In the two figures below we can see the same trend as in the fitness, the higher the fitness, the higher the survival rate. If this wasn't the case, our fitness evaluation wouldn't provide a good metric for evaluating combat performance. When playing against the hard-coded behaviour tree, it takes around 25 generations until 70% of the individuals in the population are victorious against it. These results then seem to vary between 60% and 90% win-ratio in each generation. On the other hand, when playing against itself, it takes around 28 generations until the same can be said about it. However, it's actually able to achieve a high maximum win-ratio, and has less variation between each generation, usually staying between 76% to 95% win-ratio.

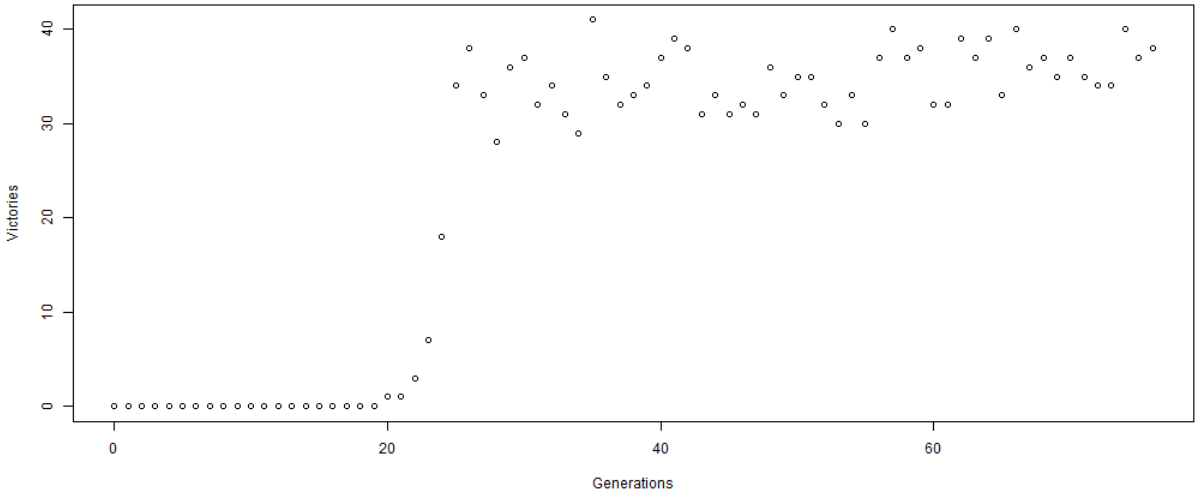


Figure 18: Victories for each generation when evaluating against a hard-coded bot. Higher value is better.

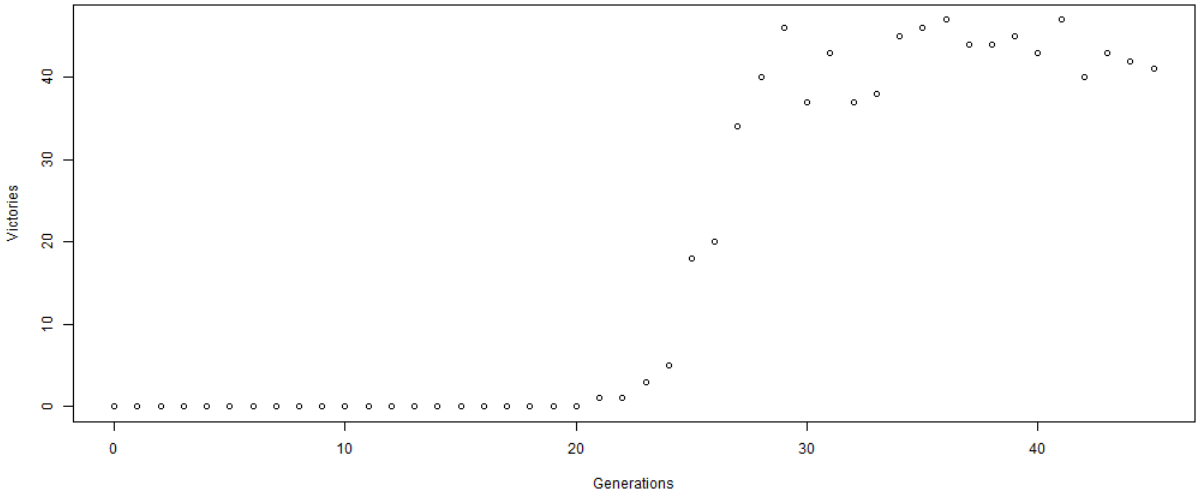


Figure 19: Victories for each generations when evaluating against 2 random peers. Higher value is better.

5 Discussion and Outlook

There are still some areas in the project which merit further analysis. We'll start with the results section, and see what we can extrapolate from the fitness and survival scores. Afterwards, we look at some limitations of our solution, compare it to related work, and then end with some future improvements.

Convergence Rate

As can be seen in the plots in the result section, the evolutionary algorithm converges to a high game proficiency in just a couple of generations, and manages to stay that throughout the evolution process.

The fact that it manages to do this may reflect poorly on our skills as game designers. Since one expects that the more complex behaviours that are required to proficiently play the game, the more generations it will take until it produces such well-tuned behaviour trees. As we mentioned before, all it takes to beat our hand-crafted behaviour trees is to play very aggressively. And the evolved trees which are generated clearly reflect this fact.

It can be viewed as the fitness function, tree mutation, and pool of possible actions being well tuned to the specific requirements of this project. Which interpretation is correct is left to the reader.

A possible explanation to the different convergence rates when evaluating against a hard-coded tree or adversarial peers is the stochastic nature of both level selection and the randomized adversary. The adversarial peer should be much closer in performance compared to the hard-coded tree. Once the trees which can defeat the hard-coded bot have been generated, additional fitness can only be reached by overfitting to the given level. The trees with peer evaluation on the other hand, can't find a solution on how to reliably defeat the enemy. The adversary co-evolves with them which makes it very hard to find a "holy-bullet" behaviour since almost all individuals in the next generations will have it.

When tested against a human player the evolved behaviours becomes very hard to defeat. Even for behaviour trees with a fitness score of around 3000. One of the reasons could have to do with the agent's superhuman accuracy when shooting, but it has to do with how greedy the agent is when it comes to pickups, sometimes it can play really safe, and keep refilling it's health and sometimes denying pickups!

Limitations

Since we decided to have high level actions, the tree's behaviour often appears clunky compared to a human player. As an example, two bots might decide to walk to the same health pickup when both have a very low health instead of just shooting at each other. Current fitness evaluation is bound to the render thread, which severely limits the number of evaluations that can be done during one second, the amount of actions, object types, and levels have been fairly limited in this project to produce faster results and ease debugging. This leads to the BTs having a limited repertoire of behaviours.

Related Work

As we mentioned before, our genetic programming solution was initially based on the work by *Colledanchise et al.* [3]. The solution proposed in the paper did not pan out in our case since they use an selection heuristic during the mutation phase, allowing for higher fitness increase over fewer generations. The paper used primitive actions which would not scale to the type of game used in this project without *very* large behaviour trees. With this said, it seems like low-level actions might produce more "clutch" behaviours in this kind of fast-paced games.

Also, even though the behaviour tree in Figure 14 is not unusable in practice, it still became deep. The bloat control and pruning steps in *Ögren et al.* [3] might not have been such a bad idea. However, our solution is algorithmically simple, and gives acceptable results. Thus, we recommend starting simple, and then follow *Ögren et al's.* [3] steps when needed.

Future Work

Some improvements are to extract the behaviour tree and genetic programming parts as an independent library and to use heuristics during the creation of each new generation. To add additional behaviours and mechanics to the project and have it compete with itself instead of a hard-coded adversary, which has been tested to some extent during the project but not enough to yield interesting results. It would be good to run fitness evaluation in headless mode and become physics independent.

If you are interested in our solution, feel free to look at our code and apply some of the additions. [4]

[4] Repository: `https://github.com/sci10n/Quake2D`

References

[1] N. A. Barricelli et al. Esempi numerici di processi di evoluzione. *Methodos*, 6(21):45–68, 1954.

[2] A. J. Champandard. The behaviour tree starter kit. *Game AI Pro*, pages 73–91, 2014.

[3] M. Colledanchise, R. Parasuraman, and P. Ögren. Learning behavior trees for autonomous agents. *arXiv 1504.05811*, 2015.

[4] M. Dawe, S. Garolinski, L. Dicken, and T. Humphreys. Behavior selection algorithms: an overview. *Game AI Pro*, pages 47–60, 2014.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] J. W. Hoff and H. J. Christensen. Evolving behaviour trees: Automatic generation of ai opponents for real-time strategy games. Master's thesis, NTNU Dept. of Info. Science, 2016.

[7] D. Isla. Managing complexity in the halo 2 ai system. In *Proceedings of the Game Developers Conference*, volume 63, 2005.

[8] C.-U. Lim, R. Baumgarten, and S. Colton. Evolving behaviour trees for the commercial game defcon. In *European Conference on the Applications of Evolutionary Computation*, pages 100–110. Springer, 2010.

[9] D. Mark. Modular tactical influence maps. *Game AI Pro*, 2:343, 2015. [Online; 2017-11].

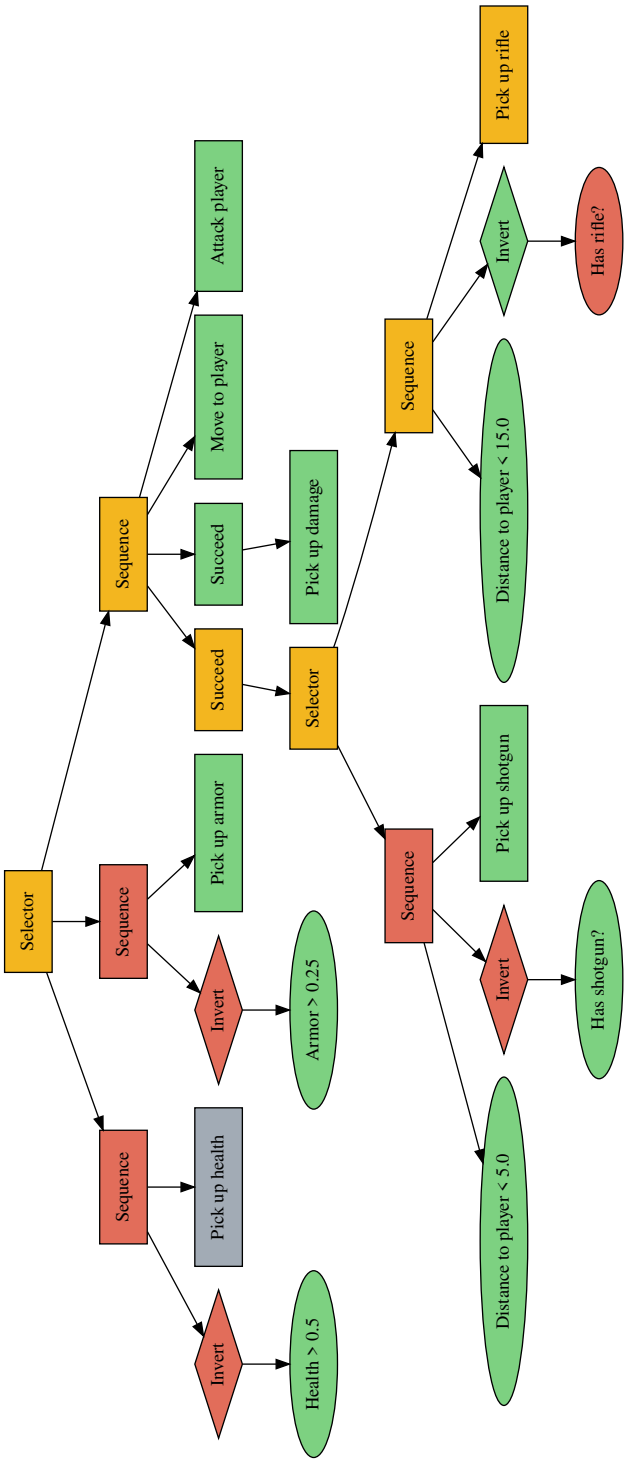
[10] R. Nystrom. *Game programming patterns*. Genever Benning, 2014.

[11] J. Orkin. Three states and a plan: the ai of fear. In *Game Developers Conference*, volume 2006, page 4, 2006.

[12] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon. Evolving behaviour trees for the mario ai competition using grammatical evolution. *Applications of evolutionary computation*, pages 123–132, 2011.

[13] C. Simpson. Behavior trees for a.i. `https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php`, 2014. Accessed: 2018-03-16.

A Appendix: Our Hand-Crafted Behaviour Tree



B Appendix: A Generated Behaviour Tree