# Physical Swarm behavior using evolved Behavior Trees

Liam R. Koehler
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
koehl238@morris.umn.edu

## ABSTRACT

Behavior trees are a modular and human readable structure for modeling agent behavior. Their tree structure allows them to be evolved using Genetic Programming. Creating agents via evolution is desirable because it allows the creation of custom agents with minimal human input. This paper will discuss the methods, findings and results of *Evolving behaviour trees for swarm robotics* by Jones *et al* [1], wherein behavior tree controllers for a swarm foraging task are evolved using genetic programming.

## Keywords

Behavior Trees, Genetic Programming, Agent Behavior, Swarm Modeling, Foraging

## 1. INTRODUCTION

Many automated processes contain the concept of an *agent*. Agents are autonomous, self-contained units that react to the world in a convincing and predictable way. Agent modeling is the task of creating the behaviors and control structure the agent will use to navigate its environment. Agents appear in a wide range of industries, including video games and robotics. Agents are special because they fully encapsulate desired behavior into self contained units. This allows the agent to operate independently without relying on an overhead control structure.

A novel use of agents can be found in the task of swarm modeling. Swarm modeling is the process of taking a simple agent and duplicating it many times, allowing many copies of the same agent to operate together in a shared space. This is called a *homogeneous swarm*. Swarms are interesting because of the possibility that complex behaviors can be created out of simple agents. This a phenomenon that is observable in the natural world: bees, termites and others are all naturally occurring swarms with complex behavior.

Behavior trees (BTs) is an agent modeling schema that grew out of the video-game industry, but is now also used in the robotics world. Behavior trees are a hierarchical tree-based system for modeling agents. Behavior trees can be used to model individual agents in a swarm. This represents the first use of behaviors trees for this task [1].

An appealing way to create agent behaviors is through the use of genetic programming (GP). GP is the process of evolving programs based on a supplied language, and a supplied task. This paper will cover the evolution of behavior tree controllers for foraging swarm robots in *Evolving behaviour trees for swarm robotics* by Jones *et al* [1]. Foraging is the task of collecting an item, such as food, and returning it to a home region. Foraging is a common swarm-modeling task because it mimics the natural behavior of many bio-swarms, such as ant-colony foraging.

Section 2 will provide an in-depth and comprehensive overview of behavior trees, including their structure, function, and history. Section 3 will examine genetic programming through the lens of evolving behavior trees. Section 4 will synthesize these topics by examining the methodologies and results of *Evolving behaviour trees for swarm robotics* by Jones *et al* [1].


*UMM CSci Senior Seminar Conference, October 2019* Morris, MN.

## 2. BEHAVIOR TREES

Behavior trees are useful for agent modeling because they break agents down into simple pieces which can be assembled as desired. As an example, let us take a foraging insect, such as a termite. At a high level, we would like our termite to **search for food** until it is *carrying food* at which point it should **walk home** and **drop food**.

The **bold** elements represent agent behaviors. These are well-defined, actionable behaviors that the agent can perform in any given moment. The *italic* element represents an interaction between the agent and the world such as, *am I currently carrying food?*

To correctly model this agent we need to create a control structure that allows the agent to switch between these behaviors as its environment changes (such as discovering food). Behavior trees, as described in this section provide such a control structure. Figure 1 shows a behavior tree that captures this termite behavior. This example behavior tree will be referenced as the structure of behavior trees is explained.

### 2.1 Structure

Behavior Trees are represented as a directed tree, made up of *action nodes* and *composition nodes*. Action nodes represent actionable behaviors for the agent, and the composition nodes represent the control structure for running these behaviors. The *root node* of a BT is unique, and conventionally has a single composition node child. Figure 1 represents the root node as $\omega$.

A behavior tree is evaluated from top to bottom, filtering down through the composition node control structure, and

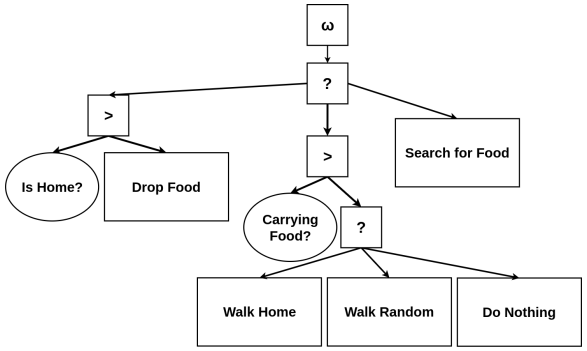


**Figure 1: A sample behavior tree, showcasing a simple termite controller**

*ticking* various behaviors until all the relevant nodes have been tried. Then, evaluation will begin again at the top, and a new selection of behaviors will be ticked. A *tick* represents one *cycle* of a behavior. Behaviors such as *move forward* need to be ticked multiple times to make meaningful forward progress.

Creating a behavior tree from scratch comes in two steps: first, create the set of composition and action nodes available to your system. This is called the *node set*. Second, place these nodes into a proper behavior tree structure. It should be noted that the same node set can be recombined in an infinite number of ways. For example, our termite node set (the set of nodes shown in Figure 1) could be reused to create behaviors for an ant, a bee, or with some re-labeling, a dog playing fetch. This modularity and re-usability is an advantage of a behavior tree system.

The remainder of this section will go over the structure of a behavior tree, including behavior nodes, composition nodes, and memory capabilities (called blackboard). Special attention will be paid to the composition node set, which is relatively static compared to action nodes.

### 2.1.1 Action Nodes

The leaves of a behavior tree are called action nodes, and represent the desired behaviors of the tree. The following action types exist:

- *Behavior*: Proper action nodes represent agent behaviors, such as "Search for Food" or "Walk Home". When reached, the action node will be ticked. Figure 1 represents those nodes as rectangles.

- *Blackboard*: Blackboard nodes are "false behaviors" that sit at the leaves of the tree, and are used to query the internal memory of the agent, as discussed later in this section. Blackboard nodes come in the form of questions, such as "Is Home?". Figure 1 represents blackboard nodes as ovals.

Action nodes are tricky because they are typically problem dependant. Behaviors such as **search for food** (from Figure 1) make no sense in the context of a self-driving-car agent for example. For this reason, it may be helpful to think of action nodes as *pointers* to behavior functions. Each behavior node simply contains a reference to the function it should call when its reached. This encapsulates the idea of both proper behavior nodes, which perform an action, and blackboard nodes, which return a Boolean result. It also reinforces the idea that BTs are nothing more than a structure for picking behaviors. The behaviors themselves still have to be supplied by some outside system.

### 2.1.2 Composition nodes

Composition nodes are contained within the tree, and represent the control structure for a BT. While behaviors change based on the problem, composition nodes are significantly more static. Composition nodes can be thought of as the *language* for a behavior tree. Different languages exist, but they all share commonalities.

Composition nodes are somewhat abstract, which can make them hard to understand at a glance. It can be helpful to think of composition nodes as replacing logical constructions in a traditional programming language. Concepts such as *if, then, else, repeat* are all achievable using the following node types: [2]

- *Select*: Select nodes will try each child from left to right, moving control to the first successful child. A select node will return false if all children fail. This can be thought of as an *if/then/else* block. The select node is used to pick a child subtree, in order of importance. A select node will return *Success* if it can find a valid child to run, and *Failure* otherwise. Figure 1 represents Select nodes as '?'.

- *Sequence*: Sequence nodes will try each child from left to right, ticking each child until a child fails. Sequence nodes can be used to run multiple behaviors in sequence. Sequence nodes can also be used to create protected behaviors, which can only be run if the behaviors to its left succeed. For example, in Figure 1, *drop food* will only run if *is home* is true, insuring that the termite only drops its food when it is home. A sequence node will return *Failure* if a child fails, and *Success* otherwise. Figure 1 represents sequence nodes as '>'.

Select and sequence nodes make up the backbone of a decision node implementation, but additional control types might be provided. These auxiliary control types are used to more conveniently represent complex behaviors. Examples include *always-true*, which simply evaluates to true without an associated query, or *random-select*, which will randomly select from its children.

### 2.1.3 Tree Traversals

BT implementations come with the concept of a *tick*. A tick is a small, measured time interval, often tied into a system clock. Each tick, evaluation of the BT begins at the root node, and continues through the composition nodes until all relevant composition nodes have been queried.

When an action node is reached during traversal, it is ticked by its parent. Each action node will handle this tick request independently. For example, when the **Search for Food** task in Figure 1 is ticked, it will fire off subtasks to begin a search procedure in the area.

When an action node is ticked, it returns a *tick state* to its parent. This state is used by the composition nodes to continue the behavior tree traversal.

The following behavior node states exist: [4]

- *Running*: The behavior node will return running if it is still processing its current behavior. This is used in scenarios where a behavior cannot be interrupted.
- *Success*: The behavior node will return success, and set its internal state to running, unless it is unable to handle the behavior for some other reason.
- *Failure*: Failure is a fallback state, and represents a behavior that is unable to be run at that time.

In each case, an iteration of the tree should conclude with a number of ticked action nodes. The set of ticked nodes determines the agents behavior in that tick.

#### 2.1.4 Blackboard Memory

A certain amount of memory is reserved for use by the BT. This is referred to as the *blackboard*. *Blackboard values* are specific memory slots (variables), which the behavior tree can query during traversals. These blackboard values can be affected by the outside world as well as by agent activities. The blackboard represents not only internal memory, but also the communication medium between the BT and the outside world.

In Figure 1, blackboard node types are represented as ovals. These nodes come in the form of questions. For our termite, those questions are **is home?** and **carrying food?**. **is home?** is an example of querying the world: the termite doesn't decide it's home, but rather checks its location against its knowledge of where home lies. **carrying food?** however represents a memory location directly set by the tree. When food is found, the food should be picked up, and the value of **carrying food?** should be set to true.

## 3. GENETIC PROGRAMMING

*Genetic programming* (GP) is a subset of artificial intelligence research that involves *evolving* programs. Genetic programming works by taking a set of randomly generated programs and applying evolutionary pressure. Evolution is achieved by iteratively combining existing programs, rating the new programs, and a filter which allows the best rated individuals to propagate from one generation to the next. After a sufficient number of generations, successful genetic programming will result in agents who have hopefully "learned" how to perform a supplied task. Genetic programming is desirable because it allows the creation of unique agents with little human intervention. [5]

While it is a mistake to take the comparison too far, it is convenient to use biological evolution as a proxy to talk about genetic programming techniques. Much of the terminology used in genetic programming is co-opted from biological evolution. Terms such as *generations*, *children*, *parents genes* and *lineages* are all more or less equivalent to their biological counterparts. Genetic programming can be thought of as a sped-up version of biological evolution, where programs are evolved instead of organisms.

When applying genetic programming to behavior trees, programs are represented *as* BTs. This means that the structure of each behavior tree can be thought of as its genetic code; the composition and action nodes, and their ordering within the tree structure *define* the individual. Each BTs rating is determined via a simulation.

The tree structure is amenable to evolution because trees are recursive, and resilient against breaking during the combination or mutation stage. In other words, changes can be applied to the tree without breaking the fundamental structure of the tree. This can be compared against a programming language such as *Java*, where very small changes are very likely to cause a compilation error. Since BTs are already trees, they are especially convenient to evolve.

### 3.1 Fitness

*Fitness* is the numeric measure of how we judge the viability of a program during evolution. Fitness can always be thought of as the "rating" or "quality" of a program. Sorting programs by their fitness is referred to as the *fitness heuristic*. What constitutes a fit individual depends on the problem. For this reason, defining a fitness heuristic is an important part of applying genetic evolution to a problem. For the purpose of evolving behavior trees, the fitness of an individual agent can be thought of as the performance of the agent in a simulation.

A good fitness heuristic has a relatively smooth gradient. This property allows even small improvements to the programs to be rewarded, and insures that the fittest programs are surviving. Large plateaus or other deformities could cause fit programs to be overshadowed by inferior ones.

### 3.2 Mutation, Crossover and subtree-shrink

For genetic programming to progress, new children need to be created each generation. These children should be based on their parents in such a way that the desirable genetic material is passed on. The tree-structure of behavior trees allow this process to occur in various ways, without breaking the structure of the tree itself. These methods are outlined below.

*Crossover* is the process of combining existing programs to create new programs. This is done by taking a sub-tree from one BT, and splicing it onto another BT from the same generation. This procedure can be thought of as combining the *genetic information* from two parents, and passing that genetic information onto a new child program.

Alongside crossover, there is the concept of *mutation*. Mutation begins by taking a *clone* of one BT, and applying one or more modifications to its structure. This can be thought of as asexual reproduction. For the purpose of evolving behavior trees, modifications come in the form of *node swaps*. A node swap is the process of replacing any given node on the tree with another node that fits. For example, a mutation of Figure 1 could be replacing the uppermost *select* node with an *always fail* node.

The example mutation above would cause the termite to enter a failure feedback loop. The termite would no longer be able to move or collect food. This is a convenient example because it shows that mutations and crossover are not strictly beneficial. Just like in nature, many mutations are in fact harmful, or fatal. Genetic programming will address this failure by rating our termite very badly, which will most likely prevent that agent from passing on its genetic material.

A common problem in evolved tree structures is uncontrollable tree-size growth. This is a multifaceted issue that is beyond the scope of this paper to address. The important thing to realise if that the mutation and crossover of trees has a tendancy to trend towards larger and larger trees. *Subtree-shrink* is used to combat this issue. Sub-tree shrink is a form of mutation where instead of preforming a node-

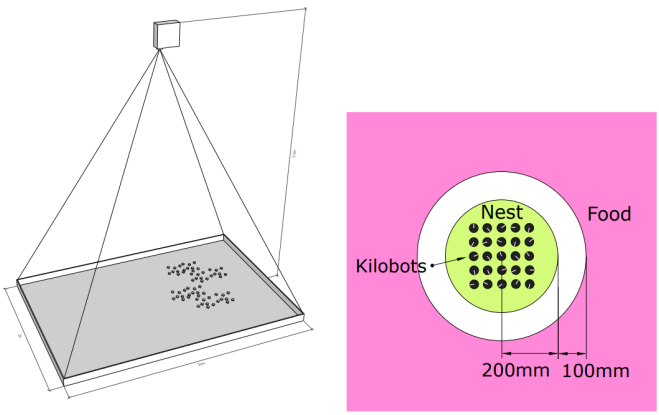


**Figure 2: Foraging, taken from [1]**

| Node | *Success if* | *Failure* if | *Running if* |
|---|---|---|---|
| Composition nodes | | | |
| Sequence: Tick until failure | N Ch S | 1 Ch F | 1 Ch R |
| Select: Tick until success | 1 Ch S | N Ch F | 1 Ch R |
| Repeat: Repeat subtree 1 time | 1 Ch S | 1 Ch F | Ch R |
| Always Succeed | Always | Never | Never |
| Always Fail | Never | Always | Never |
| Behavior nodes | | | |
| Move forward 1 tick | t - 1 | never | t - 0 |
| Move left 1 tick | t - 1 | never | t - 0 |
| Move right 1 tick | t - 1 | never | t - 0 |
| Always Succeed | Always | Never | Never |
| Always Fail | Never | Always | Never |

**Table 1: Action nodes, based on [1]**

swap, a node is cut from the tree, including all of its children.

# 4. EVOLVING BEHAVIOUR TREES

This section will combine the concepts of behavior trees and genetic programming in a concrete example. In *Evolving behaviour trees for swarm robotics* by Jones *et al* [1], the researchers evolve behavior trees for a robotics swarm-modeling task.

## 4.1 Swarm robotics and Foraging

Swarm robotics is the field of research focusing on small, autonomous robots interacting with each other, often on a large scale. The inspiration for swarm-robotics comes from colony-representative species such as bees or termites [6]. Swarm robotics uses the emergent properties of swarms to create useful robotic behaviors out of relatively simple agents.

The bio-inspiration for swarm behaviors, such as ant colonies, showcase these emergent properties well. Ant colonies are able to create vast underground nests, harvest massive amounts of food, and defend against enemies much larger than an individual ant. These feats are a cooperative consequence of many "dumb" ants working together in a swarm. [7].

Rubenstein *et al.* introduced the concept of a *Kilobot* [3]. A Kilobot is a small, physical robot which represents a single agent in a swarm. Each Kilobot is equipped with two vibrating motors, which allow the bot to turn and move forward. Light patterns projected from an overhear light-source are used by the researchers to create pseudo-physical environments, where different light patterns denote different regions. An upwards facing photo detector attached to each Kilobot detects these light patterns, and provides local location sensing. This light projector setup is shown on the left side of Figure 2. The combination of these physical properties allow the Kilobots to navigate their physical environments in a swarm-like way. These traits make them a valuable tool for modeling and studying swarm behavior.

A common task for modeling swarm behaviors is *foraging*, which is the task of moving away from a *home* area, collecting *food* and returning home. This task is interesting because it includes many agent interactions, and because it encourages cooperation among the swarm as opposed to conflict. Jones *et al.* use a photo-projector to create physical foraging environments for the kilobots. This foraging environment is shown on the right side of Figure 2. The foraging environment consists of three concentric circular regions: a *nest* region with radius 200mm is followed by a 100mm wide air gap. The rest of the accessible region is the *food* region. A Kilobot that enters the food region instantly picks up food, and a Kilobot that is carrying food when it enters the nest region instantly drops the food. Kilobots are roughly rated on the amount of food they collect in a fixed period of time. An efficient Kilobot will move rapidly back and forth betwen the nest and food regions.

## 4.2 Why behavior Trees

Jones *et al.* chose to model Kilobot behavior using behavior trees. This represents a novel use for behaviors trees. To the authors' knowledge, behavior trees have never been used to model swarm behaviors. The authors motivate their choice of agent modeling solution in the following way:

> [Behavior trees] are human readable. They are hierarchical, all subtrees are themselves behaviour trees, encapsulating a complete behaviour that can exist within a larger tree, offering possibilities for modularity and building block reuse. Finally, they can be created and optimised using the techniques of Genetic Programming. [1]

Essentially, the limited set of inputs, and the constrained way in which a BT must be built allows genetic programming to search within the space, and find successful solutions for the BT construction. Additionally, the BT maintains its human-readable characteristics, which allows the agents to be qualitatively evaluated as well as tested in both real and simulated environments.

## 4.3 Structure

Part of the complexity of designing and implementing a new genetic programming solution is deciding which components to supply the algorithm. This is a non-trivial process. Deciding which components to include and which to exclude is not always obvious. For for the purpose of evolving BTs, the input components are built using action nodes, composition nodes, and blackboard values. These elements are combined and trained using genetic programming. This section will outline the behavior tree structure that Jones *et al.* used to create Kilobot behavior.

### *4.3.1 Action Nodes*

| Name | Access | Description |
|---|---|---|
| *motors* | W | 0=off, 1=left, 2 = right, 3 = forward |
| *scratchpad* | RW | Arbitrary state storage |
| $detected_{food}$ | R | 1 = Light sensor showing food region |
| $carrying_{food}$ | R | 1 = Is carrying food |
| *density* | R | Density of kilobots in local region |
| $\Delta density$ | R | Change in density |
| $\Delta dist_{food}$ | R | Change in distance to food |
| $\Delta dist_{nest}$ | R | Change in distance to nest |

**Table 2: Composition nodes, based on [1]**

Several of the action nodes for the behavior tree are laid out in the upper section of Table 1. Recall that action nodes are *explicit, actionable behaviors*. For a Kilobot, these actionable behaviors consist of three movement based behaviors: move forward, move right, and move left. These behaviors are provided so the Kilobot can navigate its environment. The Kilobot is also provided with two auxiliary behaviors: instant fail, and instant succeed. These behaviors do not cause any actionable behavior in the Kilobot, but instead only exist to pass a Boolean value from the leaf to its parent.

As explained in section 2, nodes in a tree can return one of three states when queried. These columns are displayed as *Success if*, *Failure if*, *Running if*. The rows can be read as: N = any, Ch = child, F = Failure, S = Success. For example, the sequence node is read as; Fail if any child fails, Running, if 1 child running, otherwise Success. This information matches the description of the sequence node in section 2. This information is provided for all composition nodes, and all action nodes. Using this information will be invaluable for reading the constructed behavior tree in Figure 4.

### 4.3.2 *Composition nodes*

The composition nodes for the behavior tree are laid out in the upper section of Table 1. The *sequence* and *select* nodes operate as described in 2.1.2. Three additional composition nodes are also provided. *Always fail* and *Always succeed* operate very similarly to their action node counterparts. They are simply used to prematurely evaluate a subtree to a Boolean value. The *Repeat* node is used to repeat a subtree $n$ times. This node type is utilized by the best evolved Kilobot to repetitively move left until the change in the distance to the nest is less than some threshold.

### 4.3.3 *Blackboard*

The blackboard values available to each Kilobot are listed in Table 2. Access for each value is denoted as either write (W), read (R), or RW (read write). RW values can be thoughts of as proper memory cells: these are values that the Kilobot can read and write, and use to store information about the world. R values however, should be thought of as a communication path from the outside world. Information such as *carrying_food* or *detected_food* is read from the blackboard using *blackboard nodes*, but is not directly set by the Kilobot.

Three delta values are supplied. These are used to track the change in various properties. The blackboard can be queried to gain these values. The best Kilobot use $\Delta dist_{food}$ and $\Delta dist_{nest}$ to determine whether it is facing its respective region, after a series of turns.

## 4.4 Evolution

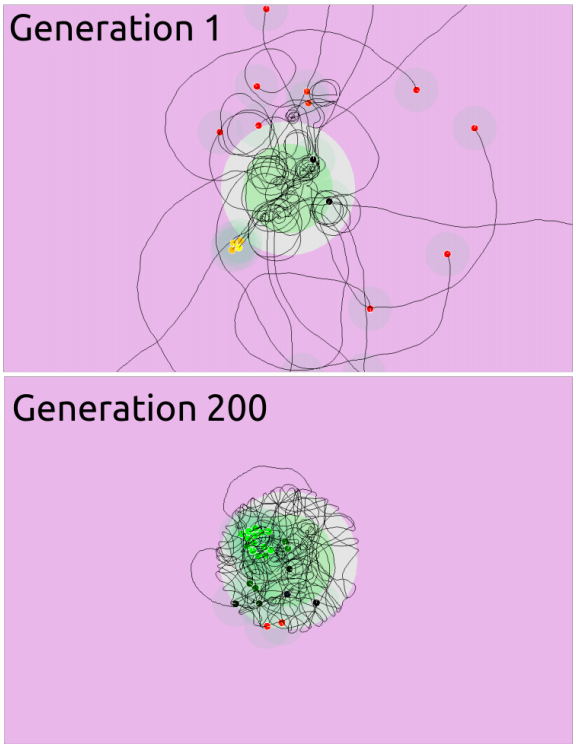


**Figure 3: Tracked Kilobot paths, modified from [1]**

A population of 25 simulated Kilobots were evolved for 200 generations. Each individual represents a unique behavior tree controller. Fitness was determined by the amount of *food gathered* over the length of a 300 second simulation, and by the size of the behavior tree. As discussed in section 3.2, uncontrollable tree growth is a problem that genetic programming faces. This is partially addressed by penalizing programs which grow too large. After evolution took place, the fittest individuals were tested in the physical environment described in Section 4.1.

Fitness rose very quickly after the first generation, as can be seen in Figure 5. The authors note that this is because a Kilobot that does nothing more than move in an arcing line is still able to gather a small amount of food, by randomly meandering in and out of the food region. The tracked paths of the Kilobots for generation one and generation 200 are shown in Figure 3. This graphic clearly shows that evolution took place, and that the Kilobots were able to learn efficient tracking back and forth between regions.

The authors note that the best lineage performed significantly better than the average lineage. This can be seen clearly in Figure 5 – the top line of the fitness box-plot rises well above the average. The authors suggest that this is because the program isn't exploring the space very efficiently, and that the average Kilobot was getting stuck in a local minima.

## 4.5 The fittest evolved behavior tree

The fittest behavior tree is studied in some depth by Jones *et al*, who simplified and analyzed the evolved behavior tree. This is one of the great advantages of evolving a behavior tree based system. Even after the complexity of evolution, the resulting agents are still potentially understandable by

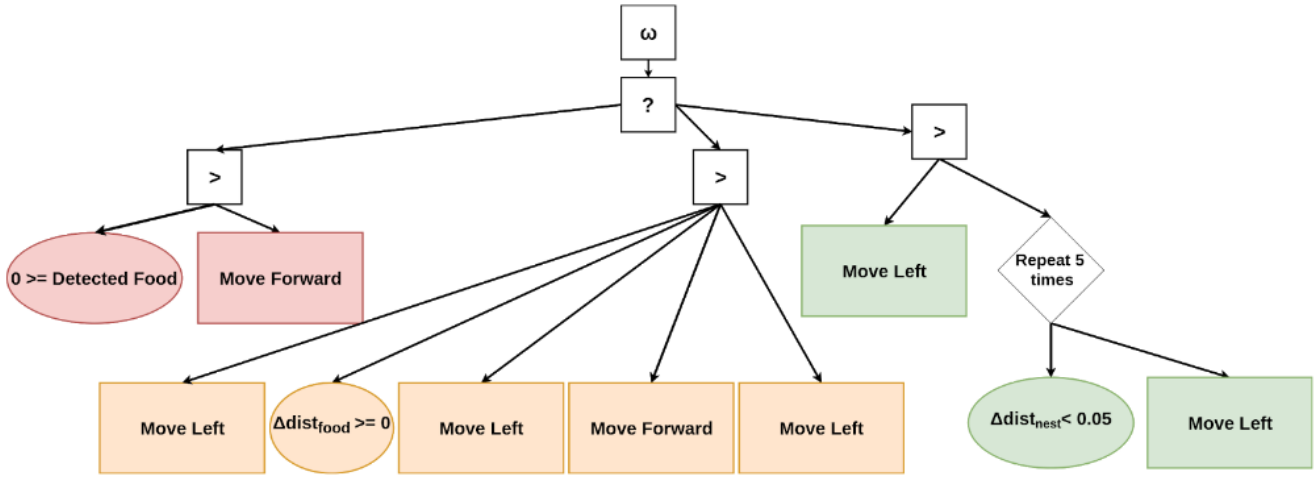


**Figure 4: The best kilobot behavior tree, adapted from [1]**

humans. Evolved behaviors can be pulled apart, analysed, or tweaked to the satisfaction of the researchers. This accessibility is a powerful tool.

The simplified code for the fittest Kilobot is shown in Figure 4, reconstructed into a behavior tree. The tree contains three predominant branches, each of which can be considered as a *sub task*. Each subtree sits as a child to a single select node. Recall that a select node is used to pick one child to run. In other words, each tick, exactly one sub task will be performed.

- The left branch contains a sequence node. Recall that sequence nodes are used to run a number of tasks in a row, until a failure is reached. This sub task can be interpreted as "move forward, until in the food region".
- The middle branch is also made up of tasks stacked under a sequence node. This sub task can be interpreted as "move left and forward until out of the food region".
- The rightmost branch is also made up of tasks stacked under a sequence node. This sub task can be interpreted as "move left until turned towards the food region".

The combination of these three sub tasks allows the Kilobot to travel efficiently back and forth between the food and the nest regions, as showcased in Figure 3.

## 4.6 Results

Figure 5 shows the results of the simulation of evolved Kilobots. The graph shows the best individual fitness across all 25 evolutionary runs. A box plot is provided every 5 generations. The main takeaway is that evolution did occur. As supported by Figure 3, the kilobots were able to "learn" how to efficiently navigate between the home region and the food region.

It is also interesting to note how the *Max fitness* line in Figure 5 rises well above the average. This shows that the best Kilobot performance was significantly better than the

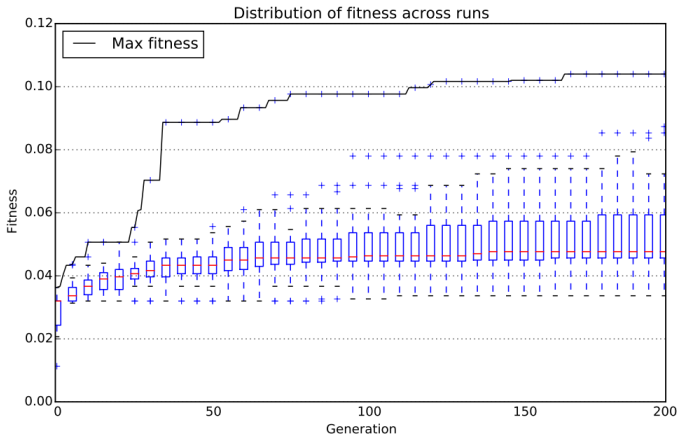


**Figure 5: Results, taken from [1]**

average. This is also supported in 3, where it can be seen that even by generation 200, some Kilobots were still meandering inefficiently.

# 5. CONCLUSIONS

Behavior trees are a powerful and simple structure for modeling agent behavior. Their modular and human readable structure makes them appealing for swarm modeling tasks. As shown in *Evolving behaviour trees for swarm robotics* by Jones *et al* [1], behavior trees can be evolved using genetic programming. The evolved behavior tree is able to succeed at a physical foraging task. Due to behavior trees human readable structure, evolved behavior trees can also be studied and tweaked after evolution has occurred.


## Acknowledgments

I would like to thank Nic McPhee, Kristin Lamberty and the students of Senior Seminar 2019.

# 6. REFERENCES


[1] S. Jones, M. Studley, S. Hauert, and A. Winfield. Evolving behaviour trees for swarm robotics. In *Distributed Autonomous Robotic Systems*, Springer Tracts in Advanced Robotics, pages 487–501. Springer, 3 2018.

[2] jschomay. Behavior Tree. `https://hexdocs.pm/behavior_tree/BehaviorTree.html`, 2018. [Online; accessed 06-December-2019].

[3] M. Rubenstein, C. Ahler, and R. Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. *2012 IEEE International Conference on Robotics and Automation*, 2012.

[4] Wikipedia. Behavior tree — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Behavior\%20tree&oldid=922038836`, 2019. [Online; accessed 15-November-2019].

[5] Wikipedia. Genetic programming — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Genetic\%20programming&oldid=921953038`, 2019. [Online; accessed 15-November-2019].

[6] Wikipedia. Swarm robotics — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Swarm%20robotics&oldid=921430950`, 2019. [Online; accessed 03-November-2019].

[7] E. Şahin. Swarm robotics: From sources of inspiration to domains of application. *Swarm Robotics Lecture Notes in Computer Science*, page 10–20, 2005.