Master's Thesis

**Master's Degree in Automatic Control and Robotics**

# Evolution of Behaviour Trees for Collective Transport with Robot Swarms

## REPORT


**Author:** Guillermo Legarda Herranz
**Director:** Dr. Simon Jones
**Speaker:** Dr. Ramon Costa Castelló

**Call:** June 2021



Barcelona School of Industrial Engineering


# Summary

Swarm robotics, inspired by natural swarms, studies how simple robots with only local sensing capabilities and no centralised control may cooperate to achieve a common goal in a robust, flexible and scalable way. A robotic system with such properties constitutes an interesting alternative to the platforms currently used in warehouses and distribution plants, where workers are at risk of injury and the space and budget available for complex infrastructure is limited. Swarm behaviours are emergent, which makes the task of designing the controllers of the individual robots particularly challenging.

In this work, we propose a method for a swarm of industrial robots to collectively transport items that are too heavy for a single agent to carry. We use artificial evolution to evolve behaviour tree controllers for the swarm agents and we conceive a decentralised coordination strategy based on local messaging. The method is developed and tested in a simulated environment, using a combination of freely available open source libraries.

The results show that a homogeneous swarm equipped with our solution is able to successfully find the items placed in the environment and transport them back to a nest region. We suggest further tuning of the evolutionary parameters and the introduction of noise in the simulator in order to improve the observed performance of the controllers in simulation and their expected performance the real world.

# Contents

# 1 Acronyms

| | |
|---|---|
| AABB | Axis-aligned bounding-box |
| ANN | Artificial neural network |
| BT | Behaviour tree |
| DOF | Degrees of freedom |
| EA | Evolutionary algorithm |
| FOV | Field of view |
| FSM | Finite state machine |
| GA | Genetic algorithm |
| GP | Genetic Programming |
| GUI | Graphical user interface |
| IMU | Inertial measurement unit |
| LED | Light-emitting diode |
| NPC | Non-player character |
| PFSM | Probabilistic finite state machine |
| RDPSO | Robotic Darwinian Particle Swarm Optimization |
| ROS | Robot Operating System |
| TOF | Time of flight |
| XML | Extensive Markup Language |

# 2 Introduction

## 2.1 Swarm robotics

Swarm robotics is the study of how a group of relatively simple robots can cooperate to achieve a common goal [1]. The term *relatively simple* is task-relative and refers to the inability of a single robot to perform the required task. The robots that compose the swarm might be homogeneous or heterogeneous [2]. They are controlled with a decentralised architecture and are asynchronous, meaning that they do not share a common time [3]. In order to complete the task, each robot within a swarm interacts locally with its environment and with other neighbouring robots.

The use of robot swarms aims to exploit three properties of natural swarms: robustness, flexibility and scalability [1]. Robustness refers to the immunity of the behaviour of the swarm to the loss of some of its individuals, flexibility is the ability of the swarm to successfully adapt to changes in the environment and scalability is the property of the swarm to be able to operate with different group sizes. While these are in fact objectives of robotic swarm designers, the common tendency to assume them as an inherent property of a system has been challenged [4].

### 2.1.1 Emergence

Natural swarms, and therefore also robotic swarms, rely on the emergence of complex behaviours through local interactions to successfully complete various tasks. This synergy is observed in a wide variety of species, particularly in social insects, who use swarm intelligence to their advantage [5]. Honeybees, for instance, collectively forage the environment to find food sources. When they find a source, foraging agents return to the hive and transmit its profitability by means of waggle dances [6]. This allows the colony to filter foraging sites, depending on the level of food scarcity.

Social insects are also known to build structures. Army ants of the genus *Eciton*, for example, use their own bodies to assemble highly dynamic bridges and thus cross gaps in their foraging trails [7]. The shape of the bridges depends on the traffic rate and the environment, and colonies consider the trade-off between the benefit of building larger bridges and losing foraging agents. Another example are termites in the genus *Macrotermes*, which build impressive soil mounds above their nests with efficient ventilation ducts [8].

There are many other complex behaviours exhibited by social insects, including aggregation, dispersion and pattern formation [9]. Additionally, other animals also exhibit notable swarm behaviours. Some species of birds, fish and land animals are commonly seen forming flocks, schools and herds, respectively. In fact, these scenarios were accurately simulated by Reynolds by defining three basic actions that every member of the group performs: alignment with their neighbours, cohesion or attraction to the group and separation from the group [10]. The work of Reynolds is therefore often mentioned as one of the earlier examples of the viability of modelling the synergic behaviour of a swarm.

Brambilla *et al.* [11] and, more recently, Schranz *et al.* [12], developed a taxonomic classification of robot swarms according to the behaviour they exhibit. These can be separated into four groups: spatial organisation, navigation, decision making and miscellaneous. In this work, we focus on the task of collective transport within the navigation group, in which swarm agents need to cooperate to transport an object that is too heavy or too large for a single one of them to carry. Agents therefore need to cooperate to agree on a common direction of motion.

Collective transport has been observed in several species of ants [13]. During a foraging task, ants leave a nest region to find a source of food, also commonly referred to as the prey. Upon encountering an item, an ant tests its resistance to motion and, if it cannot carry it nor drag it, it will recruit other ants [14]. Once a sufficient number of ants are recruited, the group transports the item back to the nest.

### 2.1.2 Design challenges

The design of individual robot controllers to achieve an emergent behaviour is arguably one of the greatest challenges in swarm robotics. This task is often tackled manually, where designers use their intuition to construct the controllers. The use of evolutionary robotics for automatic controller generation is a popular alternative. In evolutionary robotics, an initial population of controllers is evolved following the rules of natural selection and survival of the fittest until a controller that satisfies certain performance criteria emerges [15].

The controller architecture commonly selected for evolutionary robotics is an artificial neural network (ANN), which maps sensor inputs to actuator outputs. This approach, however, results in controllers with poor generalisation properties, which Francesca *et al.* [16] argue to be due to the unconstrained representational power of ANNs. This leads to an inability of the controlled swarms to cross the *reality gap*, that is, to perform satisfactorily both in simulation and in the real world.

Several approaches have been taken to attempt to reduce the effects of the reality gap. Some rely on sampling the environment through the sensors and actuators of the robot to build an accurate simulation [17, 18], while others propose to increase the robustness of the controller by introducing an appropriate amount of noise at all simulation levels [19].

Francesca *et al.* [16] compare this sensitivity to the transition to reality to the problem of overfitting in the field of machine learning. They therefore propose the introduction of bias in the automatic design of robot controllers as a way to reduce their variance. They do so by designing modular controllers with an optimisation algorithm that may only use a predefined set of behaviours as building blocks. The resulting controllers are shown to successfully overcome the reality gap.

## 2.2 Objectives

Despite modern infrastructures, workers at warehouses and distribution plants remain at risk of injury [20]. The use of robotic platforms that may assist in the transport of heavy items, such as pallets, totes and packages, not only serves to prevent harm to workers; it also leads to reduced labour costs and improved reliability of operation [21, 22]. While the acquisition of such platforms is often expensive, modular, autonomous mobile robots provide a more flexible alternative, since the number of robots needed is adaptable to the level of demand [23]. These, however, often rely on centralised control, which makes the integrity of the entire system dependent on a central unit.

Using robot swarms, we may provide the industry with cheaper alternatives with increased fault-tolerance [24]. Since the goal behaviour is achieved through synergy, the individual robots are relatively simple and, therefore, cheaper to build. Additionally, the performance of the swarm should not depend on any individual robot, hence the fault-tolerance.

In this work, we propose a method for collective transport of platforms of different weights and sizes with a swarm of industrial robots in a simulated environment. As an additional requirement, the robots should be able to move multiple items simultaneously. In order to reduce the sensitivity of the solution to the reality gap, we use behaviour trees (BTs) as a robot controller architecture, which are modular, reactive and easy to interpret, and we evolve them using genetic programming (GP) techniques. The robots also use a decentralised, direct communication and a negotiation strategy to coordinate. The platforms and the arena are designed in a way that could easily be implemented in the real world. To evaluate the generated

controllers, we build a simple 2D physics-based simulator.

We face four main challenges in the search for our solution. First, the negotiation strategy must guarantee the integrity of the swarm, the environment and the payload. Second, we must design compatible BT and GP implementations, taking advantage of existing, powerful open-source libraries. Third, our simulator must be lightweight and efficient in order to reduce the computational resources required to compute a solution. And fourth, we must be able to assess positively the performance of the swarm for the task of collective transport, according to a pre-defined measure of performance.

The following report is structured as follows: in Sections 4 and 5, we give an overview of the theory behind BTs and GP that is relevant to this work. In Section 6, we define the collective transport task and the environment. We describe the robot control architecture in Section 7, including the BT implementation and the negotiation strategy. In Section 8, we describe our 2D simulator and, in Section 9, our implementation of GP. We analyse the obtained results in Section 10 and finish with some conclusions and ideas for further development.

# 3 Related work

In this section, we review the existing literature on collective transport, focusing on the two main aspects of our project: the coordination strategy and the automatic generation of controllers.

In robotic applications, agents may communicate to achieve collective transport. Matarić *et al.* [25] devised a turn-taking strategy with direct messaging to make two six-legged robots transport an item more efficiently than a single robot. Campo *et al.* [26] introduced a negotiation strategy that allowed the robots carrying an item to collectively estimate the goal direction. Each robot used light-emitting diodes (LEDs) to indicate their direction of motion, which the rest were able to perceive. The robots were then able to carry an object towards a goal even if some of them had noisy or no perception of the goal. Ferrante *et al.* [27] used local, direct communication between robots to transport a physically attached object to a goal location. They exploited the lack of perception of some aspects of the environment by some of the robots to develop a social mediation behaviour, which the robots used to compute the goal direction.

Other authors have demonstrated the viability of collective transport using artificial evolution. Groß and Dorigo [28] used artificial evolution to achieve group transport

with robots that were unaware of each other. Baldasarre *et al.* [29] evolved neural network controllers to make a group of robots physically attached to an item move towards a goal. The robots were able to perceive the forces exerted by the item on their chassis to determine whether to push or pull. Alkilabi *et al.* [30] also evolved neural networks to develop controllers that were robust to variability in the size of the group and the items to be carried.

More recently, Hamouda [31] proposed a modification of the Robotic Darwinian Particle Swarm Optimization (RDPSO) algorithm to transport multiple different objects simultaneously [32].

# 4 Behaviour trees

Behaviour trees are structures that allow an agent to switch between executions of different self-contained tasks [33]. In the context of robotics, an agent may be an individual robot and the tasks are reusable actions through which the agent interacts with its environment. These tasks can be combined to generate more complex behaviours, which in turn may be used to build higher-level structures [34].

BTs were originally designed to define the behaviours of non-player characters (NPCs) in video games [35, 36]. Their modular and hierarchical structures allowed game developers to encode complex behaviours with high variability that satisfied runtime constraints. While most research on BTs is focused on entertainment systems, their use in academia is becoming increasingly popular [37, 38, 39].

BTs present some interesting advantages as a controller architecture. They rely on two-way control transfers where, after a function is executed, control is returned to the function that called it. BTs are reactive, which means that a robot may quickly react to changes in the environment. They are modular, so any subtree of a BT constitutes its own valid BT. This allows for testing and reusing parts of the BT separately and also results in BTs having a hierarchical structure and, thus, improved readability.

Finite state machines (FSMs) are a common modular alternative to BTs. FSMs, however, rely on one-way transfer controls, and must therefore sacrifice modularity in order to achieve reactiveness [33].

## 4.1 Behaviour tree theory

In their most basic form, BTs are directed acyclic graphs of nodes and edges. Every pair of interconnected nodes consists of an outgoing *parent* node and an incoming *child* node. A BT has a unique parent-less node, called the *root* node, and a set of child-less nodes, called *leaf* nodes. Additional *non-leaf* nodes connect the root node to the leaf nodes. To execute a BT, the root node sends a *tick* signal down the tree periodically. The tick is propagated as specified by the non-leaf nodes, typically in a depth-first, left-to-right manner, until the leaf nodes are reached. The leaf nodes then interact with the environment through a set of variables called the *blackboard*. Once executed, the leaf nodes return one of three possible states: *success*, *failure* or *running*. This result is then propagated back up the tree, possibly triggering the execution of additional leaf nodes, until the root node is reached again, which will return the state of the tree.

In this work, we use the unified BT framework developed by Marzinotto *et al.* [40], which provides an accurate and compact method to represent BTs for robot control problems. In this framework, every child node must have a unique parent, which promotes readability of the BTs. Furthermore, the BT update frequency, $f_{\text{tick}}$ is unrelated to the controller's frequency, $f_{\text{control}}$, although, as we shall see in Section 7, we set these to parameters to be equal.

### 4.1.1 Node types

There are four types of non-leaf, or *control-flow*, nodes: *selector*, *sequence*, *parallel* and *decorator* nodes. We also include the extensions *selector** and *sequence**, which represent selector and sequence nodes with memory, as detailed below. The *action* and *condition* nodes are the two types of leaf, or *execution*, nodes that complete the set of available nodes. Each node behaves as follows:

**Selector**: a selector node with $n > 1$ children ticks each child sequentially and returns *failure* if all children return *failure*. If one child returns *success* or *running*, the selector does not tick any more of its children and also returns *success* or *running*, respectively.

**Sequence**: a sequence node with $n > 1$ children ticks each child sequentially and returns *success* if all children return *success*. If one child returns *failure* or *running*, the selector does not tick any more of its children and also returns *failure* or *running*, respectively.

**Selector* (sequence*)**: the difference between a selector* (sequence*) node and a regular selector (sequence) node is that a selector* (sequence*) node holds a variable that points to the child that has most recently returned *running*. In each iteration, the selector* (sequence*) node first ticks the child pointed to by its internal variable, and then the following children sequentially. The variable is reset every time the selector* (sequence*) node returns *success* or *failure*.

As shown by Jones [41], nodes with memory are in fact syntactic sugar, and therefore any BT containing nodes with memory may also be represented in memoryless form .

**Parallel**: A parallel node takes two parameters: $S$ and $F$, where $S, F \in \mathbb{N}$. A parallel node ticks all of its children sequentially and returns *success* if the number of succeeding children is greater than or equal to $S$. Similarly, a parallel node returns *failure* if the number of failing children is greater than or equal to $F$. Otherwise, it returns *running*. We do not use parallel nodes in this work.

**Decorator**: a decorator node has a unique child. Based on its internal variables, it decides whether to tick its child or not and can modify the state it returns. In all cases, a decorator returns *running* if its child node returns *running*. In this work, we define four distinct decorators: an *inverter* decorator, which returns *success* if its child returns *failure* and *vice versa*; a *success* decorator, which always returns *success*; a *failure* node, which always returns *failure* and a *repeat* node, which returns *running* until its child node returns *success* $n$ times, or *failure* if its child returns *failure*.

**Action**: an action node may read from the blackboard and write to some of its variables, thus modifying their contents. Upon execution, an action node may return *success*, *failure* or *running*.

**Condition**: a condition node may read from the blackboard, but it may not modify any of its contents. Upon execution, a condition node may return *success* or *failure*, but it may never return *running*.

Figures 4.1-4.5 show the graphical representation of the control-flow nodes used in this work, which we borrow from [41]. In Algorithms 1-8, the pseudocode describing the behaviour of each control-flow node is given.

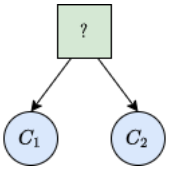



Figure 4.1: Selector node with two children

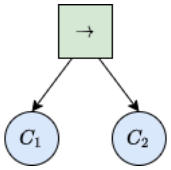



Figure 4.2: Sequence node with two children

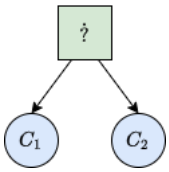



Figure 4.3: Selector* node with two children

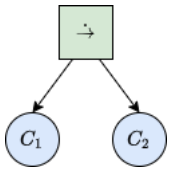



Figure 4.4: Sequence* node with two children

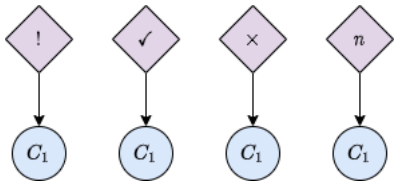



Figure 4.5: Decorator nodes: inverter (!), success (✓), failure (×), repeat ($n$)

**Algorithm 1:** Selector

```
for i ← 1 to N do
    state ← Tick(child(i))
    if state = running then
        return running
    else if state = success then
        return success
    end
end
return failure
```

**Algorithm 2:** Sequence

```
for i ← 1 to N do
    state ← Tick(child(i))
    if state = running then
        return running
    else if state = failure then
        return failure
    end
end
return success
```

**Algorithm 3:** Selector*

```
for i ← run-index to N do
    state ← Tick(child(i))
    if state = running then
        run-index ← i
        return running
    else if state = success then
        run-index ← 1
        return success
    end
end
run-index ← 1
return failure
```

**Algorithm 4:** Sequence*

```
for i ← run-index to N do
    state ← Tick(child(i))
    if state = running then
        run-index ← i
        return running
    else if state = failure then
        run-index ← 1
        return failure
    end
end
run-index ← 1
return success
```

**Algorithm 5:** Success

```
state ← Tick(child)
if state = running then
    return running
end
return success
```

**Algorithm 6:** Failure

```
state ← Tick(child)
if state = running then
    return running
end
return failure
```

**Algorithm 7:** Inverter

```
state ← Tick(child)
if state = running then
    return running
else if state = success then
    return failure
end
return success
```

**Algorithm 8:** Repeat

```
state ← Tick(child)
if state = success then
    try-index ← try-index + 1
    if try-index = cycles then
        try-index ← 0
        return success
    end
else if state = failure then
    try-index ← 0
    return failure
end
return running
```

## 4.2 Conclusions

In this section, we introduce behaviour trees and analysed the properties that make them suitable for robotics applications. We then reviewe their structure and composition according to a generalised framework and described the evaluation mechanism.

# 5 Genetic programming

Genetic programming is a type of evolutionary algorithm (EA) that aims to find a solution to a problem by evolving an initial population of computer programs, given a set of high-level objectives [42].

EAs are a set of metaheuristic optimisation techniques that draw inspiration from biological evolution [43]. In nature, the genetic material of a living organism (its *genome*) is contained in its *chromosomes*. The information contained in this genome, known as the *genotype*, will determine its observable traits, including its physical form and behaviour. These traits are known as the *phenotype* of the organism. During evolution by natural selection, individuals with favourable traits have a higher chance of survival and may reproduce at a higher rate. As a result, the presence of favourable *genes* becomes more common in successive generations of the population.

In order to apply these same principles to computer programs, we must first specify how their genome is represented. This representation must ensure that we will be able to assess the performance of an individual in a given environment (its *fitness*) and that a population of individuals will be able to reproduce to form successive generations.

## 5.1 Representation

In GP, individuals are constructed as *syntax trees* [44]. These trees may be represented in a linear form or explicitly as trees, where each edge of the tree connects two nodes from a predefined *primitive set*. For example, the program `MAX 2 TIMES 3 4` can be represented in tree-form as shown in Figure 5.1, where the tree is evaluated in a depth-first, left-to-right manner. The leaf nodes of the tree are called *terminals* and the non-leaf nodes are *functions*. A unique *root* node is the source of all the

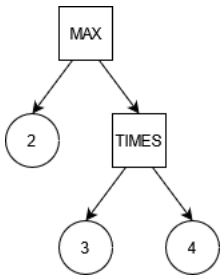



Figure 5.1: Tree representation of a program (evaluation = 12).

*branches*, or *subtrees*, of the tree. If the number of arguments of each function is known, which is usually the case, any program represented in linear form may be unequivocally translated into a tree, and *vice versa*. The representation to be used is therefore a matter of convenience.

Terminals are selected from a predefined *terminal set*, which may contain external inputs, functions with no arguments or constants. In order to introduce a degree of randomness, the values of terminal nodes may be generated at random during the construction of a tree. These values are then constants during the evaluation of the program and are known as *ephemeral random constants*.

Similarly, functions are drawn from a *function set*. Functions take a fixed number of arguments and must have *type consistency* and *evaluation safety* [44]. That is, all functions must return values of the same (or equivalent) type and must not throw runtime errors.

Finally, a core concept in this work is the definition of fixed subtrees. They are defined by the user as potentially useful structures combining terminals and functions in order to reduce the search space of the EA and thus speed up the evolution. During evolution, subtrees are treated as terminals.

## 5.2 Evolution

A run of GP can be summarised in the following steps: after initialising the population, the algorithm evaluates the fitness of its individuals, based on which it selects a subset of the population to produce a second generation by means of genetic operations. This process of selection and reproduction continues until the fitness converges to a fixed value or a predefined number of generations have been evaluated.

### 5.2.1 Initialisation

The initial population is commonly generated at random. To do so, nodes are sequentially selected from the primitive set and added to the tree until they reach a certain *depth*. The depth of a node is defined as the number of edges that must be traversed from the root node in order to reach it. The depth of a tree is therefore the maximum number of edges that must be traversed from the root node, which has a depth of zero, in order to reach any of its leaf nodes. The criterion with which nodes are selected from the primitive set in every iteration allows us to define different initialisation methods. In this work, we use Koza's *ramped half-and-half* method [45], which combines the two methods we will now introduce: *full* and *grow*.

In the full method, all the branches in a tree are generated with the same depth. To do so, nodes are selected from the function set until the maximum depth is reached. Then, only nodes from the terminal set may be selected. In the grow method, nodes are selected from the entire primitive set, so long as the maximum depth is not reached. As in the full method, once the maximum depth is reached, only terminal nodes may be selected. This results in a population where the individuals have more varied shapes.

Koza's ramped half-and-half method consists in generating trees with a range of depths. For example, given the five depths in the range $[0, 4]$, 20% of the initial population is generated with a depth parameter of zero, another 20% is generated with a depth parameter of one, and so on. For each value of the depth parameter, half of the individuals are generated using the full method, while the other half is generated using the grow method. The greater variety in sizes and shapes resulting from this method usually leads to better results [45].

### 5.2.2 Fitness

In order to compare any pair of individuals in the population, the GP algorithm assigns a fitness value to each individual [46]. The fitness is a measure of the degree to which an individual is able to satisfy the high-level objectives of the problem, and it is usually given as a single numeric value. Such high-level objectives may include the correctness of the program, its parsimony or its efficiency [45]. Thus, fitness serves to guide the algorithm towards a solution.

Often in GP, a problem will have multiple objectives, usually of a conflicting nature. In such cases, the fitness must reflect the priority of each objective and the corresponding tradeoffs. One common approach to this multi-objective GP is to combine

all individual objectives into a single value by means of a weighted sum.

### 5.2.3 Selection

Individuals for the next generation are selected based on their fitness. Commonly used methods include *fitness-proportionate selection* [47], where the probability of selecting an individual is proportional to its fitness, and *tournament selection* [48], where a group of individuals (typically two or three) are selected at random and the one with the better fitness is selected.

In this work, we use an alternative method, known as *rank selection* [49]. In rank selection, individuals are given a numerical value (rank) based on their fitness. The probability of selecting an individual is then computed based on its rank. This method reduces the difference between individuals with very high fitness and the rest of the population, thus reducing the discrimination of the method in favour of those individuals, also known as *selection pressure*. This is desirable because flooding the next generation with the genes of the fittest individuals would reduce the search space of the algorithm.

The use of any of the tactics described above may lead to an uncontrolled growth of the programs during evolution, which is known as *bloat*. The use of *elitism*, where the best individuals in the population are copied directly to the next generation so that they may be improved, has been shown to reduce bloat [50].

A more effective approach towards bloat control is the *parsimony pressure* method, where the fitness of the program is modified by subtracting from it a value that is proportional to the size of the program, that is

$$f'(x) = f(x) - cl(x), \tag{5.1}$$

where $c$ is a constant, $l(x)$ refers to the size of program $x$ and $f(x)$ to its fitness [45]. To determine the value of $c$, we use the *covariant parsimony pressure* method, which has been empirically shown to provide optimal bloat control [51]. The covariant parsimony pressure constant is calculated for each generation as

$$c = \frac{\mathrm{Cov}(l, f)}{\mathrm{Var}(l)}. \tag{5.2}$$

Note, however, that Equation 5.1 is only used to guide the selection process. To assess the evolutionary process and determine when the fitness of the program has converged to a certain value, the original fitness function, $f(x)$, is used [44].

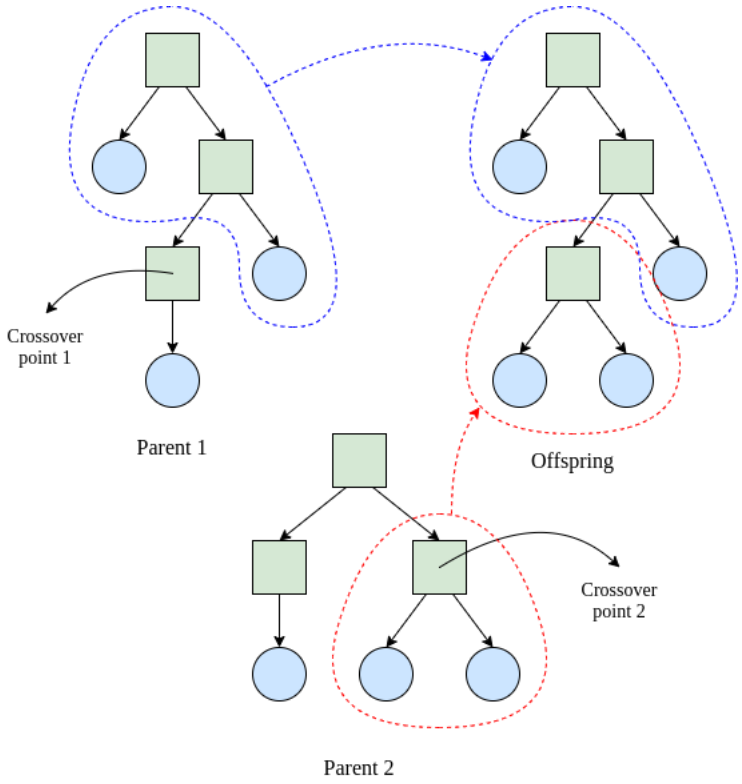



Figure 5.2: Crossover operation

### 5.2.4 Genetic operations

GP implements three operators to generate the individuals of the next generation. The *reproduction* operator simply selects an individual by means of rank selection and produces a copy for the next generation. The goal of reproduction is to increase the average fitness of the population. However, this occurs at the expense of genetic diversity [45].

The *crossover* operator is analogous to sexual reproduction. It begins by selecting two individuals using rank selection. Then, a node called the *crossover point* is selected from each tree. The offspring is formed by copying the first parent up to the crossover point and inserting at that point a copy of the subtree rooted at the crossover point of the second parent. An example of this operation is shown in Figure 5.2. As shown by Poli *et al.* [44], the majority of nodes in a tree are leaf nodes. Therefore, in order to ensure that sufficient genetic material is exchanged during crossover, inner nodes are selected as crossover points with a probability of 0.9.

*Mutation* is analogous to asexual reproduction. When the mutation operator is applied to an individual, its genetic material is altered to produce a new individual. In this work, we use three mutation operators: *point mutation*, *subtree mutation* and *parameter mutation*. In point mutation, a node is selected at random and replaced with another node with the same number of arguments. In subtree mutation, a node is selected at random and the subtree rooted there is replaced with a randomly generated subtree.

To understand parameter mutation, we get ahead of Section 5.3 to mention that, unlike in the traditional formulation of GP described in this section, many of the nodes used to construct BTs contain some kind of parameter. These parameters may be blackboard entries or ephemeral random constants, for example. In parameter mutation, A node is selected at random and, if it takes any parameters, a new set of parameters is randomly generated.

In contrast to reproduction, mutation aims restore genetic diversity to the population. However, as argued by Holland [47], the main driving forces of evolution are reproduction and crossover. Mutation is therefore used sparingly.

## 5.3 Application to behaviour trees

BTs are naturally well-suited for GP implementations. All nodes return *success*, *failure* or *running*, and all subtrees are valid BTs. Therefore, they are intrinsically type-consistent, and all genetic operators can be implemented in the BT framework.

The terminal set of a GP formulation applied to BTs consists of execution nodes (actions and conditions) and fixed subtrees. Unlike in the classical formulation of GP, BT terminals may take a fixed number of parameters, which can be constants or blackboard entries. The function set is the composed of control-flow nodes, which have at least one child and may also take parameters.

## 5.4 Conclusions

In this section, we review the classical formulation of genetic programming. We emphasise the parallelisms between GP and natural selection and survival of the fittest, and introduce the representation of computer programs as tree-like structures.

We then analyse the steps of the evolutionary process, beginning with the different initialisation methods and their effects on the size and shape of the resulting

population. The production of subsequent generations of the population through fitness-based individual selection and genetic operations that modify genetic diversity is then summarised.

Finally, we justify the intrinsic suitability of behaviour trees for genetic programming applications.

# 6 Collective transport

In this section, we define the task of collective transport and describe the experimental setup designed to evolve and test our controllers.

## 6.1 Task description

For our robot swarm to successfully achieve collective transport, the agents must be able to locate and retrieve items that are too large or too heavy for a single agent to carry. The agents must therefore be able to safely attach themselves to the item and coordinate with the rest of the *porters* in order to move it. Similarly, once they reach the goal location, they should be able to safely deposit the load.

Porters should also be able to transport an item independently of the actions of the rest of the robots in the swarm, thus allowing for simultaneous collective transport of multiple loads. Furthermore, the coordination mechanism must ensure that neither the porters nor the payload will be damaged in the process.

## 6.2 Environment

In order to evaluate the performance of our controllers, we construct a simulated environment with the following components: an arena, to delimit the area where the task may take place; loads, as the items to be carried by the agents, and a robotic platform to carry out the task. Figure 6.1 shows a possible instance of the environment.

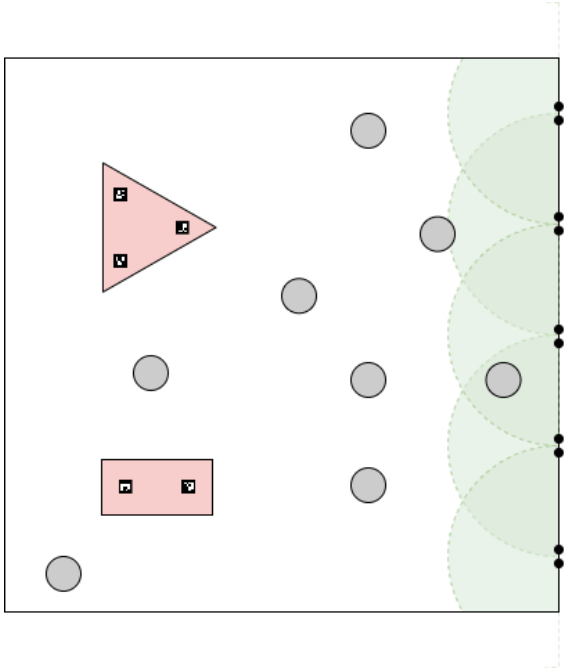


Figure 6.1: Task environment. A 5m × 5m arena with five 100mm-wide ArUco markers on the right wall indicating the location of the nest. Two loads, one to be carried by two porters and another one to be carried by three, both with lifting points indicated also by 100mm-wide ArUco markers. Additional guiding markers are omitted for the sake of clarity. Eight holonomic robots with local sensing capabilities.

### 6.2.1 Agents

The robotic platform we use as a swarm agent is the successor to the Xpuck robot used by Jones *et al.* to build a Teraflop swarm [52]. While it is still under development, it may be seen as a representative of the next generation of swarm robots for industrial purposes, as it provides additional capabilities for sensing and manipulation of the environment [24]. We base our model of the platform on the original robot, but some of the parameters may differ.

Each robot is a cylindrical device weighing 2kg that is 115mm tall and has a 250mm diameter. A minimum of three ArUco markers on the body allow for a camera to determine the relative position of the robot, regardless of its orientation [53]. Additionally, each marker also holds the unique identifier (ID) of the robot. The three omnidirectional wheels on its base, located at a distance of 120mm from its centre, make the robot holonomic and allow it to move in any direction on a plane with a maximum velocity of $0.2\text{ms}^{-1}$. Additionally, a robot can raise a circular lifting platform from 30mm to 80mm above it, which can carry a payload of up to

2kg.

In order to sense its environment, each robot relies on various sensors. Four body cameras with a 120º horizontal field of view (FOV) and a 90º vertical FOV, with the front two overlapping for stereo vision, provide full coverage of the surroundings. A fifth camera in the lifting platform is used to detect the items to be lifted. An array of 16 laser time-of-flight (TOF) distance sensors with a 150mm range surround the robot for obstacle avoidance purposes. A nine degree-of-freedom (DOF) inertial measurement unit (IMU) allows a robot to determine its acceleration and the local magnetic field and, therefore, its orientation with respect to any cardinal point. Additionally, robots can communicate between themselves, but we make no assumptions about the communication protocol they would use in the real world. A schematic representation of the robot is shown in Figure 6.2.

The dimensions $l_{pcam}$ and $w_{pcam}$ shown in Figure 6.2 represent the projection of the platform camera plane on a flat surface. Considering a flat surface at a distance $h$ above the camera, they are obtained from the following expression:

$$D_i = 2h \tan\left(\frac{FOV_i}{2}\right), \tag{6.1}$$

where $D_i = \{l_{pcam}, w_{pcam}\}$ and $FOV_i = \{FOV_l, FOV_w\}$, the vertical and horizontal FOV of the camera (see Figure 6.3).

The velocity of the robot and the linear velocities of the wheels obey the following relationship:

$$\begin{bmatrix} v_1(t) \\ v_2(t) \\ v_3(t) \end{bmatrix} = \begin{bmatrix} -\sin(\pi/3) & \cos(\pi/3) & d \\ 0 & -1 & d \\ \sin(\pi/3) & \cos(\pi/3) & d \end{bmatrix} \cdot \begin{bmatrix} v(t) \\ v_n(t) \\ \omega(t) \end{bmatrix}, \tag{6.2}$$

where $v_{i\in\{1,2,3\}}(t)$ are the linear wheel velocities at time $t$, $d$ is the distance from the centre of the base to each omnidirectional wheel and $v(t)$, $v_n(t)$ and $\omega(t)$ are the linear velocities in the $X$ and $Y$ directions and the angular velocity of the robot in the moving frame at time $t$, respectively.

In order to provide the robots with *local* sensing capabilities, we limit the communication and body camera ranges to $r_{comms} = r_{bcam} = 1\text{m}$.

### 6.2.2 Arena

We consider a 5m × 5m rectangular arena surrounded by unmovable walls. To indicate the location of the nest where the robots must deposit the items, ArUco

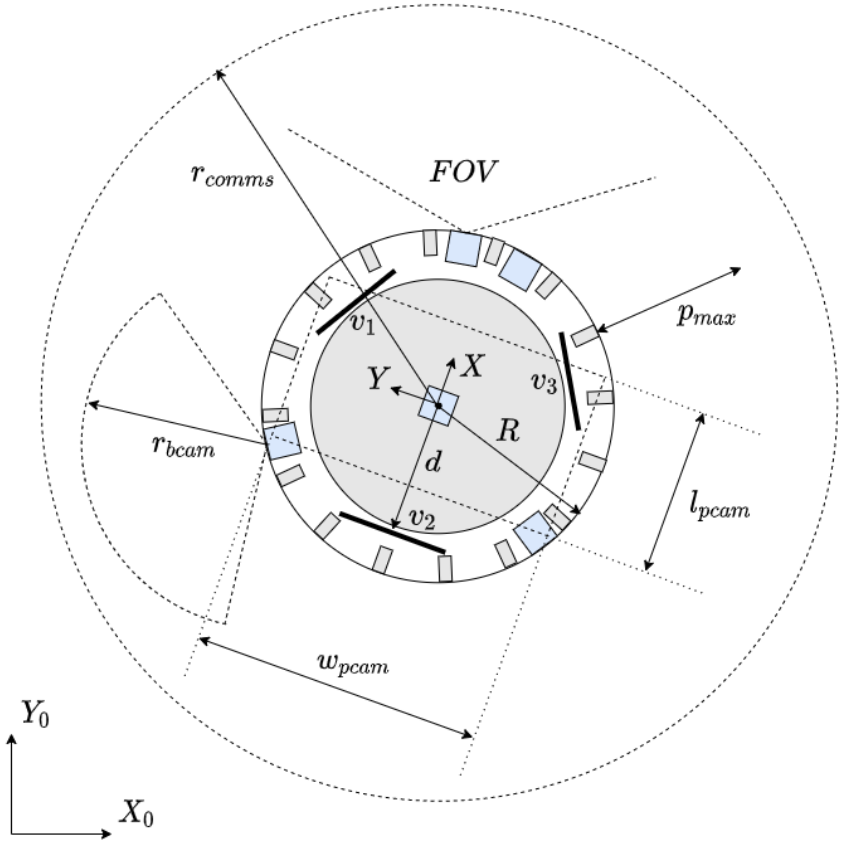



Figure 6.2: Schematic view of robot used in this work. The central grey circle represents the lifting platform. Laser TOF distance sensors are shown in grey and cameras, in blue. $R$ is the robot radius and $d$ is the distance from the centre of the robot to each omnidirectional wheel. The communication range is $r_{comms}$, $r_{bcam}$ is the body camera range and $p_{max}$ is the range of the distance sensors. The projection of the platform camera plane on a flat surface has length $l_{pcam}$ and width $w_{pcam}$. The linear velocities of each wheel are $v_1$, $v_2$ and $v_3$.

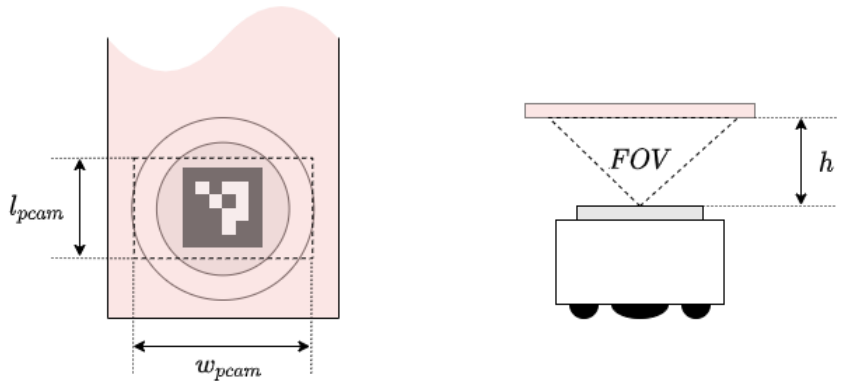



Figure 6.3: Top view (left) and side view (right) of the position of a robot that is ready to lift. $l_{pcam}$ and $w_{pcam}$ are the projections of the platform camera plane on the underside of the load, $h$ is the separation between the platform camera and the underside of the load and $FOV$ is the field of view of the platform camera.

markers are placed on the walls. The detection of a single ArUco marker provides enough information to estimate the pose of the detecting camera. By defining the nest as the area within a given radius around a marker, a robot may determine when it is at the nest and when it is not. We set this distance to $r_{nest} = 1$m.

As shown in Figure 6.1, five 100mm-wide markers are placed along the right wall of the arena to indicate the location of the nest. That way, no robot may reach the right wall without detecting a nest marker.

### 6.2.3 Loads

The items that the swarm agents need to carry back to the nest are 185mm-tall (40mm above the robot platforms) thin platforms shaped as regular polygons. We place 100mm-wide ArUco markers on the underside facing the floor to indicate the position where the agents should be located in order to lift the loads. Figure 6.3 shows this configuration and the relevant parameters of the platform camera. The markers also contain information about the unique ID of the load and the number of porters required to lift it. Additionally, given the small size of the lifting point markers and the projection of the platform camera plane, we assume the remaining surface of the load to be covered with more ArUco markers that hold a vector pointing towards the nearest lifting point in order to guide the robot.

To determine the size of the platforms, we consider the following range for the

separation between any pair of lifting point markers, $d_{lp}$:

$$2(R + p_{max}) < d_{lp} < r_{comms}, \tag{6.3}$$

where all the variables are defined in Section 6.2.1. The lower bound prevents porters under the same load from detecting each other as obstacles, while the upper bound ensures that these porters will form a fully-connected network, thus allowing them to share all information required for coordination in a single iteration of broadcasting and processing. We therefore set $d_{lp} = 0.65$m and limit the size of the loads the agents may carry to 4-porter loads[1].

Furthermore, since each robot may carry up to 2kg on its platform, the density of the items is set to $\sigma = 9\text{kg m}^{-2}$. This value ensures that we may generate up to 8-porter loads, such that, for any $n$-load, a minimum of $n$ robots are required to lift it[2]. While loads requiring more than 4 porters are not considered in this work due to the full-connectivity constraint, we select this value of $\sigma$ to leave the door open to future development.

Figure 6.1 shows a 2-porter load and a 3-porter load. Loads with a higher porter count are generated similarly, with one vertex and one lifting-point marker per porter. In order to clearly show the location of these markers, we exclude the additional guiding markers from the figure.

## 6.3 Conclusions

This section presents the three components required to construct an environment where we can generate and test the performance of robot controllers for the task of collective transport: an arena, a robotic platform and the loads. Even though we only ever treat them as simulated entities in this work, we design them with a real-world implementation in mind.

We are now ready to describe the core of our work, beginning with the control of the robots.

[1]Throughout this work, we will often refer to loads as $n$-porter loads, where $n$ is the number of porters required to lift them.

[2]The upper limit of 8 porters per load is based on the maximum number of vertices a polygon may have in the Box2D simulator (see Section 8)

# 7 Robot control

We devise a two-stage control process of our robots. For the first stage, we use BTs as the controller architecture. At every iteration of the controller, a BT is ticked and generates the desired commands for each robot. In the second stage, the desired commands are fed to a negotiation strategy to compute the outputs that will be executed. In this section, we first describe the *reference model* of our robot, followed by the BT and negotiation implementations.

## 7.1 Reference model

The reference model of a robot is a mathematical abstraction of its sensor data and actuators [16]. It represents the available inputs to the controller and the corresponding outputs. By specifying the task to be carried out by the robot, we can choose a fitting level of abstraction. Table 7.1 shows the reference model of the robots described in Section 6.2.1.

The readings from the distance sensors, $P_{j\in\{1,\dots,16\}}$ are mapped to a $[0, 1]$ range, such that an object adjacent to the sensor gives $P_j = 1$ and an object at a distance greater than or equal to $p_{max}$ results in $P_j = 0$. A linear map is used for this purpose.

While orientation is commonly calculated with respect to north, we use east as the reference cardinal direction for convenience, as it is aligned with the $X_0$-axis and the nest region of our environment is located in the $+X_0$ wall of the arena. Since our robot is holonomic, its orientation, $\theta$, is not indicative of its direction of motion. Therefore, we use $\theta$ and the local velocity of the robot, which we obtain from Equation 6.2, to compute the *heading* of the robot, $\phi$, such that $\phi = 0$ always represents the direction of motion. Using $\phi$ instead of $\theta$ as the direction of reference in the definition of all local vectors makes the control stage less geometrically involved.

The remaining input variables are obtained from the body and platform camera feeds. We treat all body cameras as a unique vision sensor. Since all robots, lifting points and nest locations are identified with ArUco markers, each robot can obtain their locations in range and bearing form. In the case of the robots, a neighbour count is also kept.

The detection of nest markers is reduced to the detection of any marker and its location in polar form, since all markers indicate the location of the same nest region. Therefore, the robot only stores the location of the nearest marker.

Table 7.1: Robot reference model.

| Input variables | Value | Description |
|---|---|---|
| $P_{j\in\{1,...,16\}}$ | $[0,1]$ | Reading of distance sensor $j$ |
| $\theta$ | $[-\pi,\pi)$ | Compass orientation |
| $n$ | $\{0,...,15\}$ | Number of neighbouring robots |
| $(r_n,\angle b_n)_{j\in\{1,...,n\}}$ | $([0,r_{bcam}],[-\pi,\pi))$ | Range and bearing of neighbour $j$ |
| $H$ | $\{0,1\}$ | Nest detection |
| $(r_h,\angle b_h)$ | $([0,r_{bcam}],[-\pi,\pi))$ | Range and bearing of nearest nest marker |
| $C$ | $\{0,1\}$ | Load detection |
| $(r_c,\angle b_c)$ | $([0,d_{lp}/2],[-\pi,\pi))$ | Range and bearing of lifting point |
| $(l,m)$ | $(\{0,...,3\},\{0,...,4\})$ | Load ID and porter multiplicity |
| Output variables | | |
| $v_{i\in\{1,2,3\}}$ | $[-v_{max},v_{max}]$ | Linear wheel velocities |
| $p$ | $\{-1,0,1\}$ | Platform commands |
| Constants | | |
| $t_{control}$ | 0.1 s | Controller update period |
| $R$ | 0.125m | Robot radius |
| $\angle q_{j\in\{1,...,16\}}$ | $\pi(j-1)/8$ | Angular position of distance sensor $j$ |
| $r_{bcam}$ | 1m | Maximum body camera range |
| $d_{lp}$ | 0.65m | Maximum inter-maker separation |
| $r_{nest}$ | 1m | Radius of the nest region centred on a marker |
| $w$ | 0.1m | Width of ArUco markers |
| $v_{max}$ | $0.17\text{ms}^{-1}$ | Maximum linear wheel velocity |

A robot can obtain the location of the lifting point markers in one of two ways. The first one is direct observation, where the robot detects the lifting point with the platform camera. This also allows the robot to extract the data of the load: its ID, $l$, and the number of porters it requires, $m$. In this case, the location of the marker is calculated as the projection of the relative location on the horizontal plane. The second way is indirectly, through the guiding markers on the platform that point towards the nearest lifting point marker.

The output variables of the model are the wheel velocities and the platform commands. The value of $v_{max}$ is then obtained from Equation 6.2 and the maximum linear velocity of the robots. As for the platform commands, we consider a finite set of possible values, which will lower $(-1)$, lift $(1)$ or not move it $(0)$.

### 7.1.1 Communication

To collectively transport an item, robots must coordinate their actions in a way that guarantees the safety of the payload and of all the porters. The coordination strategy must therefore ensure that no collisions occur between the robots or the payload and any other item in the environment. Furthermore, all porters must exert the same force on the load simultaneously in order to avoid reactive forces that may lead to damages.

We now present a coordination method for robots based on local messaging, which allows them to convey their intentions to other porters prior to acting. By means of the negotiation strategy presented in Section 7.3, the swarm may then safely lift and transport multiple loads at the same time.

At every control iteration, the BT controller of each robot generates a target velocity and a platform command, in accordance with the reference model of the robot. We refer to these outputs as the *vote* cast by the robot. Each robot then broadcasts a message, $M$, containing the following entries:

$$M = \{i, l, g, r_{vote}, \angle b_{vote}, p_{vote}\} \tag{7.1}$$

where $i$ is its unique ID, $l$ is the unique ID of the load whose lifting-point marker is directly above the robot, or zero if there are none, $g \in \{0, l\}$ is its *group ID*, $r_{vote}$ and $\angle b_{vote}$ are the magnitude and direction of the voted velocity in the *global* frame and $p_{vote}$ is the voted platform command. Note, however, that the outputs generated by the BT are not guaranteed to be in the ranges defined in the reference model. This is actually a desirable feature since, as we shall see in Section 7.2.3, magnitudes reflect the importance of a desired action.

The message is received by all the neighbours in the range $r_{comms}$. With this message structure in mind, we propose an extension of the reference model, as defined in Table 7.2.

The group ID serves two different purposes: coordination and unlocking the motion of the platform. Coordination is achieved by having each robot negotiate every action with those neighbours in the same, non-zero group. The goal is that each robot will only coordinate with the other porters under its same load, thus allowing other agents to carry a different load simultaneously, without interference. Furthermore, a robot without a group (i.e. $g = 0$) should be able to act independently of the actions of its neighbours.

To understand why the group ID is needed to unlock the motion of the platform, we first identify two critical safety concerns that arise from robots moving their

Table 7.2: Messaging extension of the reference model.

| Input variable | Value | Description |
|---|---|---|
| $s$ | $\{0, ..., 15\}$ | Number of messages received |
| $i_{j\in\{1,...,s\}}$ | $\{1, ..., 16\}$ | Messenger $j$ ID |
| $l_{j\in\{1,...,s\}}$ | $\{0, ..., 3\}$ | Load ID messenger $j$ |
| $g_{j\in\{1,...,s\}}$ | $\{0, ..., 3\}$ | Group ID of messenger $j$ |
| $(r_{vote}, \angle b_{vote})_{j\in\{1,...,s\}}$ | $([0, \infty), [-\pi, \pi))$ | Global velocity magnitude and direction proposed by messenger $j$ |
| $p_{vote,j\in\{1,...,s\}}$ | $\mathbb{R}$ | Platform velocity proposed by messenger $j$ |

platforms at will in our environment:

- A robot has no means of preventing a collision between its lifted platform and a load at rest.
- A robot has no means of knowing when a load is too heavy.

Therefore, a robot must not lift its platform unless it is located under a load, and the action of lifting and lowering a load must be executed simultaneously by all required porters located at the corresponding lifting points, thus preventing damages to the platforms. Both of these situations can be accounted for by setting the group ID only when all the porters required to carry a load are located under the lifting points, and only allowing robots to move their platforms when their group ID is non-zero. Additionally, the group ID may only be nullified when the group is broken, which can only occur when the platforms are down and either a robot leaves its lifting point or the load is removed.

To modify its group ID, each robot counts the number of different messengers with the same, non-zero value of $l$. It then compares the count to the porter multiplicity of the load, $m$, and sets $g$ according to the rules we just defined. If we define the *porter state* of a robot at a given time, $x_p$, as the 3-tuple

$$x_p = \{p, l, g\}, \tag{7.2}$$

where $p$ is the state of the platform, Table 7.3 summarises the states available and Figure 7.1 shows the possible transitions as a FSM.

Table 7.3: Possible porter states. $p$ is the lifting platform state, $l$ is the load ID and $g$ is the group ID

| $p$ | $l$ | $g$ | Description |
|---|---|---|---|
| 0 | 0 | 0 | Default |
| 0 | 1 | 0 | Waiting for other porters |
| 0 | 1 | 1 | Ready to carry a load |
| 1 | 0 | 1 | Carrying a load |

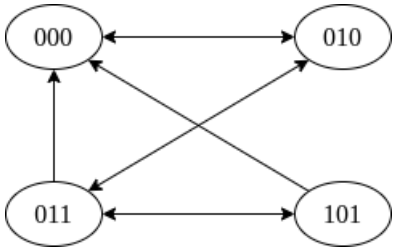



Figure 7.1: Porter state transitions available. Each 3-tuple is $x_p = \{p, l, g\}$, where $p$ is the platform state, $l$ is the ID of the load detected and $g$ is the group ID.

## 7.2 Behaviour tree controller

Due to the successful implementation of modular controllers for swarm tasks by Jones [41] and Francesca *et al.* [16], we use their work as a reference for our own implementation of BT controllers. First, we use the reference model of our robots determine the blackboard entries the BTs can access. We then define the execution nodes available to our controller, which we use to define the constituent behaviours and conditions. Finally, we describe the implementation of our BTs in C++.

### 7.2.1 Blackboard

The set of scalars and vectors shown in Table 7.4 constitute the blackboard of the BT controllers. Each robot in the swarm holds its own copy of the blackboard. In each copy, all the vectors are given in polar form with respect to the heading of the robot. Every entry of the blackboard has an access level, which indicates how the BT may interact with it. Thus, execution nodes may obtain the value of an entry with read (R) access and my also edit entries with write (W) access.

The first eight entries contain information about the scene perceived by the robot. All vectors are defined initially in the moving frame of the robot. The necessary

Table 7.4: Blackboard entries.

| Name | Access | Description |
|---|---|---|
| $\boldsymbol{v_{prox}}$ | R | Location of nearby obstacles |
| $\boldsymbol{v_{attr}}$ | R | Location of nearby neighbouring robots |
| $\boldsymbol{v_{recr}}$ | R | Location of recruiting neighbours |
| $\boldsymbol{v_{home}}$ | R | Location of the nearest nest marker |
| $\boldsymbol{v_{lift}}$ | R | Location of the nearest lifting point |
| $s_n$ | R | Number of neighbouring robots |
| $s_r$ | R | Number of neighbouring recruiters |
| $s_p$ | R | Number of porters of the same load |
| $p_{vote}$ | RW | Voted platform command |
| $\boldsymbol{v_{vote}}$ | RW | Voted velocity |
| $s_{zero}$ | R | Constant zero scalar |
| $\boldsymbol{v_{zero}}$ | R | Constant zero vector |
| $s_{scr}$ | RW | Scalar scratchpad |
| $\boldsymbol{v_{scr}}$ | RW | Vector scratchpad |

rotations to bring them to the heading-aligned frame are performed afterwards. Thus, the proximity vector, $\boldsymbol{v_{prox}}$, is calculated as

$$\boldsymbol{v_{prox}} = \sum_{j=1}^{16} (P_j, \angle q_j). \tag{7.3}$$

It therefore points in the direction of the nearest obstacles, using their proximity as a weighing factor.

The attraction vector, $\boldsymbol{v_{attr}}$, has a similar function. Instead of pointing towards the nearest obstacles, it points towards (or away from) the nearest neighbouring robots. It is therefore obtained from the following expression:

$$\boldsymbol{v_{attr}} = \begin{cases} \sum_{j=1}^{n} \left( \dfrac{1}{1 + r_{n,j}}, \angle b_{n,j} \right), & \text{if } n > 0 \\ (1, 0), & \text{otherwise.} \end{cases} \tag{7.4}$$

Similarly, the recruitment vector, $\boldsymbol{v_{recr}}$, points towards (or away from) the nearest neighbours who are under a lifting point, but need more robots to lift the corresponding load. Since neighbours are identified by ArUco markers and messages contain the ID of the sender, we can define the set $\mathcal{S}$ as the range and bearing data

of the nearest neighbours with $l \neq 0$. The vector $\boldsymbol{v_{recr}}$ is then calculated as

$$\boldsymbol{v_{recr}} = \begin{cases} \sum_{j \in \mathcal{S}} \left( \frac{1}{1 + r_{n,j}}, \angle b_{n,j} \right), & \text{if } |\mathcal{S}| > 0 \\ (1, 0), & \text{otherwise,} \end{cases} \tag{7.5}$$

where $|\mathcal{S}|$ denotes the cardinality of $\mathcal{S}$. Note that, to compute $\mathcal{S}$, we are combining current sensor data with messaging data that is obsolete by one iteration. Given the high controller update frequency, however, we expect the controller to still be able to exploit the data profitably.

Robots use the $\boldsymbol{v_{home}}$ and $\boldsymbol{v_{lift}}$ vectors to determine the location of the nest and lifting point markers. Additionally, their lengths should convey when a robot is *at* the nest (within a certain distance from the marker) and *at* the lifting point (directly under it). We therefore define the following distance measures,

$$d_{nest} = \max\{r_h - r_{nest} + 2R + 0.1, 0\}, \tag{7.6}$$

$$d_{lift} = \max\left\{r_c - \frac{w}{2} + 0.1, 0\right\}, \tag{7.7}$$

where the constants are introduced as a guarantee that, as $d_{i \in \{nest,lift\}} \to 0$, the robot is *at* the nest/lifting point. We then use them to calculate $\boldsymbol{v_{home}}$ and $\boldsymbol{v_{lift}}$ as

$$\boldsymbol{v_{home}} = \begin{cases} (d_{nest}, \angle b_h), & \text{if } H = 1 \\ (1, -\theta), & \text{otherwise} \end{cases} \tag{7.8}$$

$$\boldsymbol{v_{lift}} = \begin{cases} (d_{lift}, \angle b_c), & \text{if } C = 1 \\ (1, 0), & \text{otherwise.} \end{cases} \tag{7.9}$$

Thus, when no nest marker is detected, a robot moving in the $\boldsymbol{v_{home}}$ direction will advance towards the $+X_0$ end of the arena, where we have placed the nest markers.

Since all vectors are given in polar form, we then rotate them to the heading-aligned framed by modifying their orientation. We express this as $\boldsymbol{v'} = R(\boldsymbol{v}, \theta - \phi)$, where we have introduced a rotation operator for vectors in polar form

$$R((r, \angle A), \angle B) = (r, \angle(A + B)). \tag{7.10}$$

The scalar entry $s_n$ is set directly from the input as $s_n = n$. As for $s_r$, it is set to the number of neighbouring recruiters, that is, $s_r = |\mathcal{S}|$. The $s_p$ entry is defined as

$$s_p = \begin{cases} \sum_{j=1}^{s} \delta_{g_j g}, & \text{if } g > 0 \\ 0, & \text{otherwise,} \end{cases} \tag{7.11}$$

where $\delta_{g_j g}$ is the Kronecker delta, which in turn is defined as

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise.} \end{cases} \tag{7.12}$$

The next two entries, $\boldsymbol{v_{vote}}$ and $p_{vote}$, are the output variables of the controller, which will be fed to the negotiation strategy. We impose a the restriction on all BT controllers that either $\boldsymbol{v_{vote}}$ or $p_{vote}$ may be written no more than once per tick of the tree. This is motivated by two facts. First, that applying these outputs usually requires some finite time and overwriting them in a single iteration would render the overwritten actions of the controller useless. And second, that actuating the platform while the robot is moving is potentially dangerous because it could lead to a violation of the safety rules outlined in Section 7.1.1.

The last four entries are defined for convenience. The $s_{zero}$ and $\boldsymbol{v_{zero}}$ entries represent a constant null scalar and a constant null vector, respectively. They can only be read by the BT. The $s_{scr}$ and $\boldsymbol{v_{scr}}$ entries represent the scratchpad of the BT. They do not hold any particular value and may be read and written by the BT. Besides serving as intermediaries for operations more complex than those defined in the action set, they also provide the EA with space to evolve new behaviours.

### 7.2.2 Execution nodes

Table 7.5 shows the action and condition nodes our BTs use to interact with the blackboard. With the exception of `movpv`, they are all taken from the work by Jones [41]. While a detailed description of their functionality can be found there, we summarise its content here for self-containment of the work.

One crucial difference between our implementation of BTs and that of Jones, however, is that we treat vectors as non-separable data types (i.e. we cannot access the length or the orientation as separate entries). Therefore, nodes operating on scalar entries may not access the components of a vector.

The `movcs` node takes a signed 8-bit value, $i$, and sets a scalar entry to that value. The use of 8-bit values allows us to reduce the search space of the EA without compromising its behaviour. Similarly, the `movcv` node sets a vector entry to a unit vector with orientation given by a signed 8-bit value as $\angle a = \pi \frac{i}{128}$. Thus, $\angle a \in \{-\pi, ..., \pi \frac{127}{128}\}$.

We introduce the `movpv` node as a consequence of the holonomic nature of our robots.

Table 7.5: Execution nodes. Vectors are boldface, scalars are plain. Variables $\{d, \boldsymbol{d}\}$ indicate output blackboard entries, while $\{s1, s2, \boldsymbol{s1}, \boldsymbol{s2}\}$ represent input blackboard entries. 32-bit floating-point values are denoted by $f$, 8-bit signed integers, by $\{i, j\}$. Variables $k$ and $l$ are signed 5.3 fixed-point values [41].

| Name | Parameters | Behaviour | Description |
|---|---|---|---|
| `movcs` | $d, i$ | $d \leftarrow i$ | Set $d$ to scalar constant |
| `movcv` | $\boldsymbol{d}, i$ | $\boldsymbol{d} \leftarrow (1, \pi\frac{i}{128})$ | Set $\boldsymbol{d}$ to unit vector constant |
| `movpv` | $\boldsymbol{d}, \boldsymbol{s1}, i$ | $\boldsymbol{d} \leftarrow (1, \text{rand}(\angle\boldsymbol{s1} \pm \lvert\pi\frac{i}{128}\rvert))$ | Set $d$ to unit vector constant with random orientation within a sector |
| `mulas` | $d, s1, f, s2$ | $d \leftarrow s1 + f \times s2$ | Scalar scale and add |
| `mulav` | $\boldsymbol{d}, \boldsymbol{s1}, f, \boldsymbol{s2}$ | $\boldsymbol{d} \leftarrow \boldsymbol{s1} + f \times \boldsymbol{s2}$ | Vector scale and add |
| `rotav` | $\boldsymbol{d}, \boldsymbol{s1}, i, \boldsymbol{s2}$ | $\boldsymbol{d} \leftarrow \boldsymbol{s1} + R(\boldsymbol{s2}, \pi\frac{i}{128})$ | Vector rotate and add |
| `successl` | | *success* | Always return success |
| `failurel` | | *failure* | Always return failure |
| `ifprob` | $s1, k, l$ | $P_{success} = \frac{1}{1 + e^{k(l-s1)}}$ | Probabilistic success |
| `ifsect` | $\boldsymbol{s1}, i, j$ | $c = \pi\frac{i}{128}$ | Check for vector in a sector |
| | | $w_{1/2} = \lvert\pi\frac{j}{256}\rvert$ | |
| | | *Success* if $\lvert\angle\boldsymbol{s1} - c\rvert < w_{1/2}$ | |

By providing it with an input vector entry of the blackboard, $\boldsymbol{s1}$, and an 8-bit scalar, $i$, `movpv` sets a blackboard vector to a unit vector with random orientation. The range of possible orientations is determined by the orientation of $\boldsymbol{s1}$, which defines the centre of an angular sector, and $i$, through which we define the half-width of the sector as $w_{1/2} = \lvert\pi\frac{i}{128}\rvert$. This node provides the BT with a straightforward method to apply random velocities to the robots.

Through the `mulas` and `mulav` nodes, the BT can combine, modify and copy blackboard entries in various ways. The multiplying 32-bit floating-point value, together with the $s_{zero}$ and $\boldsymbol{v_{zero}}$ entries, allow the BT to add and subtract entries, scale them and copy their value to a different entry. Combining these capabilities with the scratchpad entries, $s_{scr}$ and $\boldsymbol{v_{scr}}$, provides us (and the EA) with flexibility to generate new, more complex behaviours.

Similarly, the `rotav` node rotates a vector by means of the operator defined in Equation 7.10, using a signed 8-bit scalar to define the angle of rotation $\angle a = \pi\frac{i}{128}$, and adds it to another vector. It also provides the possibilities of combining and copying vectors of the `mulav` node.

The remaining nodes are condition nodes. The first two, `successl` and `failurel`, simply return *success* or *failure* upon evaluation.

The `ifprob` node, on the other hand, is more complex, in that it introduces probabilistic behaviour. The node returns *success* with a probability determined by a logistic function. The 5.3 signed fixed-point values, $k$ and $l$, define the steepness and the location of the centre of the function, respectively, where we refer to the centre as the value of the scalar entry, $s1$, that makes $P_{success} = 0.5$. While a low value of $k$ results in a smoothly varying function, a large value of $k$ will give a step-like function.

Finally, the `ifsect` node checks for the existence of a vector in a given sector. Given two signed 8-bit scalars, $i$ and $j$, it defines a sector centred on $\angle c = \pi\frac{i}{128}$ with half-width $w_{1/2} = \left|\pi\frac{j}{256}\right|$. The node then returns *success* if the vector lies within that sector, $|\angle \boldsymbol{s1} - c| < w_{1/2}$, and is sufficiently large, $|\boldsymbol{s1}| >= 0.1$. Otherwise, it returns *failure*. If $j = 0$, the node returns *success* if $|\boldsymbol{s1}| < 0.1$.

### 7.2.3 Constituent behaviours and conditions

We now use the blackboard entries and execution nodes defined in the previous sections to construct a set of behaviours and conditions that are potentially useful for the task of collective transport. We express these behaviours and conditions as BTs, which will be used by the GP algorithm as subtrees to generate controllers.

We take the behaviours and conditions used by Francesca *et al.* as our main source of inspiration [16]. In their work, they construct probabilistic FSM (PFSM) controllers for e-puck robots using six atomic behaviours and six conditional state transitions. The behaviours they define are *exploration*, *stop*, *phototaxis*, *anti-phototaxis*, *attraction* and *repulsion*. The conditional transitions are *black-floor*, *gray-floor*, *white-floor*, *neighbour-count*, *inverted-neighbour-count* and *fixed-probability*.

We therefore define the following behaviours:

**Exploration:** the robot executes a random walk around the arena. To do so, it moves in a straight line until it encounters an obstacle, which occurs when $\boldsymbol{v_{prox}}$ points towards the heading of the robot and $|\boldsymbol{v_{prox}}| > 0.1$. It then sets a new random direction of motion away from $\boldsymbol{v_{prox}}$.

**Stop:** the robot does not move its wheels or its platform.

**Attraction/Repulsion:** the robot moves towards or away from its nearest neighbours, with embedded obstacle avoidance. It follows the vector $\boldsymbol{v} = \alpha\boldsymbol{v_{attr}} - k\boldsymbol{v_{prox}}$, where, $\alpha$ is a real-valued parameter in the range $[-5, 5]$ and the value of $k$ is a fixed parameter set to 5 to reflect the importance

of obstacle avoidance, as suggested by Francesca *et al.*. The robot will be attracted towards its neighbours when $\alpha$ is positive, and repelled from its neighbours when $\alpha$ is negative. The magnitude of $\alpha$ determines the degree of attraction/repulsion.

**Recruitment/Anti-recruitment:** the robot moves towards or away from its nearest neighbours who are waiting under a lifting point. It follows the vector $\boldsymbol{v} = \alpha \boldsymbol{v_{recr}} - k\boldsymbol{v_{prox}}$, where $\alpha \in [-5, 5]$ and $k = 5$. The robot will be recruited when $\alpha$ is positive, and anti-recruited when $\alpha$ is negative. The magnitude of $\alpha$ determines the degree of attraction/repulsion.

**Position:** the robot moves towards the nearest lifting point by following the vector $\boldsymbol{v} = \boldsymbol{v_{lift}} - k\boldsymbol{v_{prox}}$.

**Home:** the robot moves towards the nest by following the vector $\boldsymbol{v} = \boldsymbol{v_{home}} - k\boldsymbol{v_{prox}}$.

We also introduce an additional behaviour, *avoidance*, which is included as a fixed element in randomly generated controllers to prevent damages to robots that do not execute any of the actions defined above, but does not take part in the evolutionary process. The robot moves in a straight line and, if it encounters and obstacle ahead such that $|\boldsymbol{v_{prox}}| > 0.2$, it moves in the opposite direction. The larger magnitude required to trigger the reaction prevents the avoidance behaviour from interfering with the other predefined behaviours.

We then define the following probabilistic state transitions:

**Neighbour-count:** returns *success* with probability

$$z(s_n) = \frac{1}{1 + e^{k(l - s_n)}}, \tag{7.13}$$

where $k$ and $l$ are the logistic function steepness and centre, respectively.

**Inverted-neighbour-count:** returns *success* with probability

$$z(s_n) = 1 - \frac{1}{1 + e^{k(l - s_n)}}, \tag{7.14}$$

where all parameters are as defined in Equation 7.13.

**Recruiter-count:** return *success* with probability

$$z(s_r) = \frac{1}{1 + e^{k(l - s_r)}}, \tag{7.15}$$

where $k$ and $l$ are the logistic function steepness and centre, respectively.

**Inverted-recruiter-count:** returns *success* with probability

$$z(s_r) = 1 - \frac{1}{1 + e^{k(l - s_r)}}, \tag{7.16}$$

where all parameters are as defined in Equation 7.15.

**Porter:** if $s_p > 0$, returns *success* with probability $\beta$.

**Nest:** if $|\boldsymbol{v_{home}}| < 0.1$, returns *success* with probability $\beta$.

**Lifting-point:** if $|\boldsymbol{v_{lift}}| < 0.1$, returns *success* with probability $\beta$.

**Item:** if $|\boldsymbol{v_{lift}}| < 1.0$, returns *success* with probability $\beta$.

**Fixed-probability:** returns *success* with probability $\beta$.

Figures 7.2-7.13 show the constituent behaviours and conditions expressed as BTs, using the control-flow nodes introduced in Section 4.1.1 and the execution nodes defined in Section 7.2.2. Note that the conditions *Neighbour-count* and *Inverted-neighbour-count* can be represented with the same node, since a flip of the sign of $k$ will result in the two conditions. The same argument is used to define a unique node for *Recruiter-count* and *Inverted-recruiter-count*.

### 7.2.4 Implementation

Our C++ implementation of BTs uses the BehaviorTree.CPP library developed by Faconti and Colledanchise [54]. It is written in C++14 and has several attractive qualities for the field of robotics research. We emphasise three of these qualities. First, it is compatible with both ROS1 and ROS2, so controllers developed with this library can be readily implemented on the corresponding physical platform [55]. Second, it is compliant with Groot, a graphical user interface (GUI) also developed by Faconti to create BTs [56]. And third, BTs are generated using the XML language, which promotes readability of the trees.

With the exception of the *repeat* decorator, the rest of the control-flow nodes described in Section 4.1.1 are already defined in BehaviorTree.CPP. The equivalence between their naming convention and the one we use in Section 4.1.1 is given in Table 7.6. We also include in the table the BehaviorTree.CPP names for the `successl` and `failurel` nodes defined in Section 7.2.2. To define the *repeat* decorator and the execution nodes defined in Section 7.2.3, we follow the established inheritance tree.

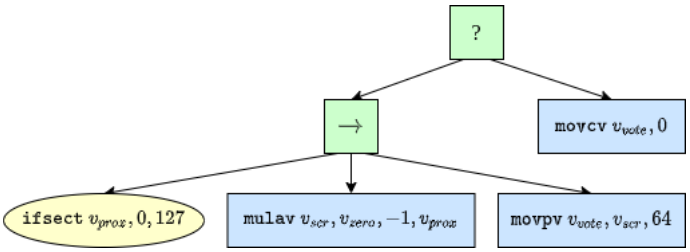



Figure 7.2: Exploration behaviour.

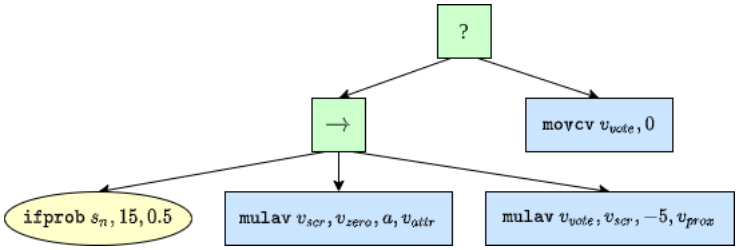



Figure 7.3: Attraction/Repulsion behaviour. $a$ is a real-valued parameter.

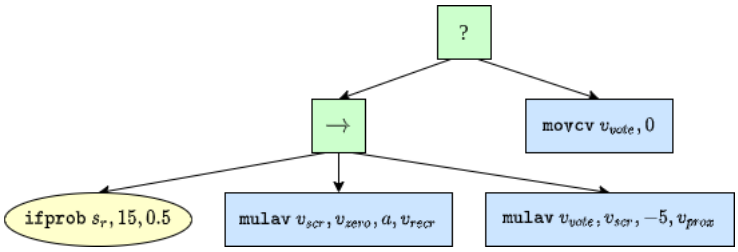



Figure 7.4: (Anti-) Recruitment behaviour. $a$ is a real-valued parameter.

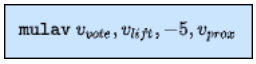



Figure 7.5: Position behaviour


Figure 7.6: Home behaviour

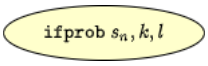



Figure 7.7: Neighbour-count/Inverted-neighbour-count conditions. $k$ and $l$ are signed 5.3 fixed point values.

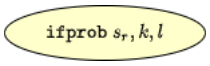



Figure 7.8: Recruiter-count/Inverted-recruiter-count conditions. $k$ and $l$ are signed 5.3 fixed point values.

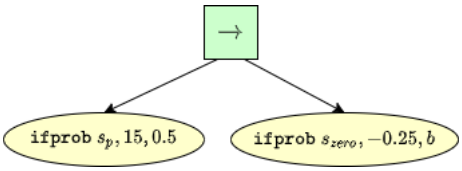



Figure 7.9: Porter condition. $b$ is a signed 5.3 fixed point value.

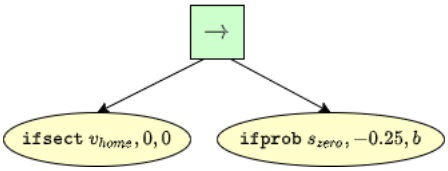



Figure 7.10: Nest condition. $b$ is a signed 5.3 fixed point value.

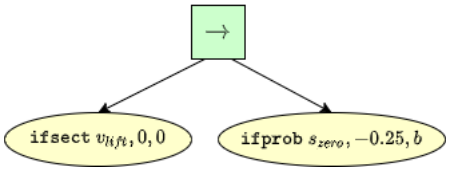



Figure 7.11: Lifting-point condition. $b$ is a signed 5.3 fixed point value.

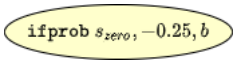



Figure 7.12: Fixed-probability condition. $b$ is a signed 5.3 fixed point value.

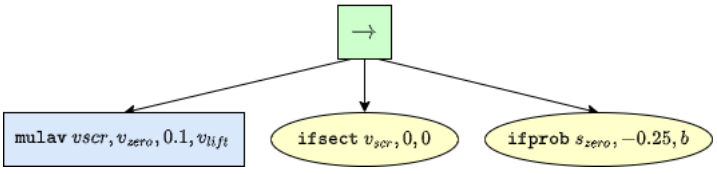



Figure 7.13: Item condition. $b$ is a signed 5.3 fixed point value.

Table 7.6: Equivalence between our naming convention and the BehaviorTree.CPP naming convention.

| This work | BehaviorTree.CPP |
|---|---|
| Sequence | ReactiveSequence |
| Selector | ReactiveFallback |
| Sequence* | Sequence |
| Selector* | Fallback |
| Inverter | Inverter |
| Success | ForceSuccess |
| Failure | ForceFailure |
| `successl` | AlwaysSuccess |
| `failurel` | AlwaysFailure |

All the nodes we define may take three different types of arguments: an integer, a floating-point value, or a vector in polar coordinates. These may be provided as explicit values like `5`, `-3.14` or `[1.2;3.4]`, respectively, or the names of blackboard entries, like `{vlift}` or `{pvote}`. A generic execution node is then written using the following XML tag template:

```
<${NAME} arg0="${X0}" ...  argn="${XN}"/>,
```

where `NAME` is the node name as defined in Table 7.5 with the first letter capitalised and {`X0`, ..., `XN`} are the parameters of the node. Thus, for example, to have the robot move towards the nest with embedded obstacle avoidance, we would write the corresponding `mulav` operation as

```
<Mulav arg0="{vvote}" arg1="{vhome}" arg2="-5.000" arg3="{vprox}"/>.
```

To write control-flow nodes in XML, we must open and close the corresponding XML tag, which encloses all the children of the node. If they contain any arguments, like the *repeat* decorator, they are included as in the execution node case. Thus, a control-flow without parameters is implemented using the following XML template:

```
<${NAME}>
   ${CHILD0}
   ...
   ${CHILDN}
</${NAME}>,
```

where {CHILD0, ..., CHILDN} are the children of the node. A control-flow node with parameters combines the two templates and is therefore implemented as follows:

```
<${NAME} arg0="${X0}" ...  argn="${XN}">
   ${CHILD0}
   ...
   ${CHILDN}
</${NAME}>.
```

As an example, a decorator that repeats the `mulav` action defined above 10 times would be written in XML as

```
<Repeat arg0="10">
   <Mulav arg0="{vvote}" arg1="{vhome}" arg2="-5.000" arg3="{vprox}"/>
</Repeat>.
```

Note that the format we use to implement all nodes is motivated by the need to generate them automatically in the GP stage. Thus, the vague argument naming allows us to sequentially include as many parameters as the node requires. Additionally, BehaviorTree.CPP provides two different syntaxes to write nodes in XML. We have demonstrated above the use of the *compact* syntax, which comes at the expense of not being able to display the resulting BTs in Groot. However, one could also use the *explicit* syntax and display the trees in Groot directly. Using explicit syntax, we would write our execution nodes as

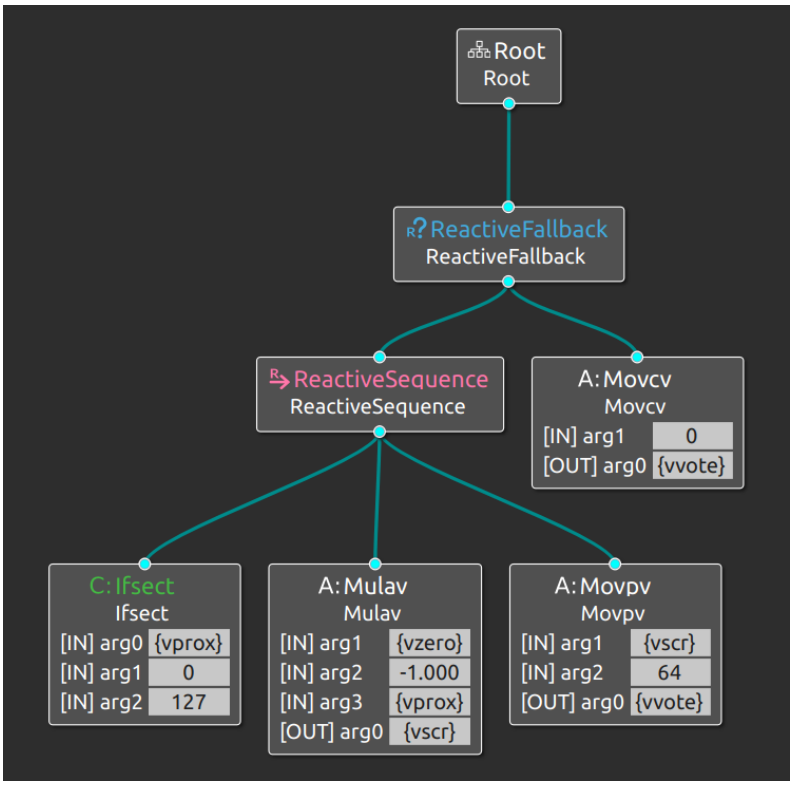
```
<${TYPE} name="${NAME} arg0="${X0} ...  argn="${XN}/>,
```

where the added field, `TYPE`, indicates the node type (Action or Condition). Introducing this distinction at the node level is undesirable because it would force us to also make it at the GP level. Therefore, for the sake of simplicity, we use the compact syntax and create a Groot template file, where we can paste the trees in compact form to obtain a Groot-ready file.

As a complete example, Figure 7.14 shows the exploration behaviour defined in Section 7.2.3 in XML and the corresponding representation in Groot.

```
<BehaviorTree ID="MainTree">
   <ReactiveFallback>
      <ReactiveSequence>
         <Ifsect arg0="{vprox}" arg1="0" arg2="127"/>
         <Mulav arg0="{vscr}" arg1="{vzero}" arg2="-1.000" arg3="{vprox}"/>
         <Movpv arg0="{vvote}" arg1="{vscr}" arg2="64"/>
      </ReactiveSequence>
      <Movcv arg0="{vvote}" arg1="0"/>
   </ReactiveFallback>
</BehaviorTree>
```

(a) XML tree.




(b) Groot display.

Figure 7.14: Exploration behaviour.

## 7.3 Negotiation

Through the negotiation strategy, each robot gathers the votes from the rest of the porter group and computes the target output. The goal is to have all porters of the same item behave as an single swarm agent that is be able to execute the behaviours of an individual robot. Therefore, we want our negotiation strategy to comply with three ideas. First, all robots must compute the same output. If each porter attempted to move in one direction, it would generate shear forces on the lifting platforms, which could lead to damages. Second, since swarm behaviour is decentralised, we want our negotiation strategy to also be decentralised, so that no robot may lead the rest of the porters. An third, each porter must be able to convey the urgency or importance of their vote. This allows the group to react to changes in the environment like a single agent. For example, if two robots are carrying an item and one encounters an obstacle, the goal velocity should result in avoidance of the obstacle.

Since the individual BT controllers already generate outputs with magnitudes proportional to the importance of the motion, each robot calculates the target output as the sum of the individual votes. Thus, for $s$ messages received, each robot obtains the the goal velocity, $\boldsymbol{v_{goal}}$, and the goal platform command, $p_{goal}$, from

$$p_{goal} = p_{vote} + (1 - \delta_{0g}) \sum_{j=1}^{s} p_{vote,j} \delta_{g_j g} \tag{7.17}$$

$$\boldsymbol{v_{goal}} = (\boldsymbol{v_{vote}} + (1 - \delta_{0g}) \sum_{j=1}^{s} \boldsymbol{v_{vote,j}} \delta_{g_j g}) \delta_{0 p_{vote}} \tag{7.18}$$

where $\delta_{0g}$ and $\delta_{g_j g}$ are the Kronecker deltas that allow the robots to consider only the neighbours in their same, non-zero group and $\delta_{0 p_{vote}}$ prevents actuating the wheels and the platform simultaneously. We choose to prioritise the actuation of the lifting platform because a successful controller should use it sparingly and rely on the wheels for most purposes.

We can therefore conclude that having a fully-connected porter network topology is necessary for the correct functioning of the negotiation strategy. At every controller iteration, each robot must receive the votes from all the other porters in order to compute a safe velocity or platform command.

Initially, we attempted to include the negotiation strategy as a fixed element of the BT controller which may potentially be evolved. Since robots need time to communicate, we designed a controller where a negotiation node, which implemented Equations 7.17 and 7.18, was the first child of a two-child selector* node. The second child of the selector* node was the original BT controller. At each BT update, the

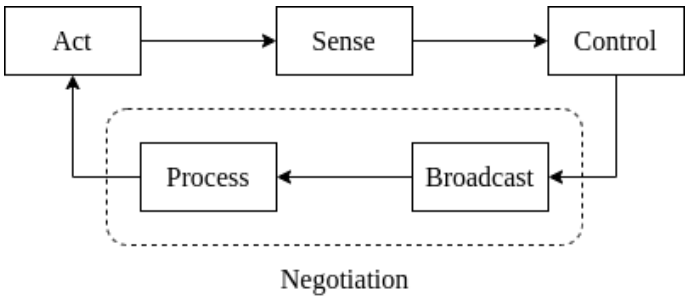



Figure 7.15: Robot cycle. After updating sensor readings, it updates its controller to generate votes. The votes are fed to the negotiation system, which produces the target output of the group. This is then applied to the actuators.

selector* node would first tick the negotiation node, which wrote to $\boldsymbol{v_{goal}}$ and $p_{goal}$, and then execute the rest of the BT and write the $\boldsymbol{v_{vote}}$ and $p_{vote}$ for the next iteration.

This method, however, led to a delay in the application of the actions calculated by the controller of exactly $t_{control}$, the controller update period. To see why this is problematic, consider the following controller iterations in a situation where the robot should apply an obstacle avoidance behaviour:

1. An obstacle is detected ahead. The negotiation node sets $\boldsymbol{v_{goal}} \leftarrow \boldsymbol{v_{vote}}$ and the robot keeps moving forward. Then, $\boldsymbol{v_{vote}} \leftarrow -\boldsymbol{v_{prox}}$.

2. First, sensor data is updated and blackboard entries are set. Since no action has been applied yet, an obstacle is still detected ahead. Then, $\boldsymbol{v_{goal}} \leftarrow \boldsymbol{v_{vote}}$, away from the obstacle. Finally, with the execution of the rest of the BT, $\boldsymbol{v_{vote}} \leftarrow -\boldsymbol{v_{prox}}$.

3. The robot is now heading away from the obstacle, so no obstacles are detected ahead. Then, $\boldsymbol{v_{goal}} \leftarrow \boldsymbol{v_{vote}}$, which brings the robot back towards the obstacle.

This cycle is repeated forever, and the robot never executes the avoidance behaviour. This led us to remove the negotiation strategy from the BT in order to execute the control and actuation steps within the same sensor update cycle. Each robot then executes the cycle shown in Figure 7.15.

## 7.4 Conclusions

In this section, we introduce a two-stage controller for our robots. We first build a reference model by abstracting their sensors and actuators, considering the task of collective transport and the environment of operation. A communication strategy for safe coordination is then proposed and the model extended accordingly.

We then define the behaviour trees that constitute the first of the stages. A suitable list of blackboard entries provide the BTs with an interface to interact with the sensors and the second stage of the controller. We construct the actions and conditions that may query and modify the balckboard by means of mathematical operators, which we then use to define a set of constituent behaviours and conditions as fixed subtrees. We also summarise our implementation of behaviour trees using C++.

Finally, we introduce a safe negotiation strategy as the second stage of the controller, which allows our robots to decide on a common goal prior to acting.

# 8 Simulation

Every generation of the evolutionary process needs to evaluate the performance of all the controllers generated multiple times. In a hypothetical process of 500 generations with an initial population of 20 individuals and 8 evaluations per individual, $80,000$ simulations of the task would be needed (without elitism). Assuming evaluations cannot be carried out simultaneously, an increment of 1s in simulation time per evaluation would result in over 22h of additional run time. A fast simulator is therefore essential to economise computational resources, so we choose to build our simulator in C++, a compiled language.

To simplify the interactions between the elements of the environment, we use Box2D, a physics simulator designed for games, to model each component as simple geometric shapes with dynamic properties [57].

## 8.1 Environment initialisation

The simulator is centred on an instance of the environment. We may add simulated objects to it, compute their interactions and obtain the resulting fitness. Additionally, we have the possibility of rendering a run using OpenGL [58].

As described in Section 6.2, there are three types of simulated objects we can add to the environment:

**Robots:** We model the agents using six *fixtures*. The body is a dynamic flat disk, which may collide with the walls of the arena and with other robots. It may not, however, collide with loads, thus reflecting that they exist on a lower plane of the environment. Four wedge-shaped fixtures with no mass act as body cameras. These wedges are configured as *sensors* and therefore do not generate a collision reaction. They do, however, generate contacts with ArUco markers of the nest and other robot bodies, and thus we can check for overlap of these fixtures. A rectangular sensor centred on the body of the robot acts as the platform camera, which generates contacts with the ArUco markers of the loads.

The distance sensors are modelled using ray casts at run time and are therefore not included in the construction of the robot.

Each robot also contains an instance of a controller. The controller consists of a BT and various functions to update it and access the blackboard entries.

**Arena:** A closed, static chain constitutes the walls of the 5m × 5m arena. The arena may collide with robots and with loads, and thus no element of the environment can leave it. We implement the ArUco markers of the nest as finite edges on the right wall of the arena, which may generate contacts with the body cameras of the robots.

**Loads:** The loads are modelled as dynamic closed polygons that may only collide with the arena. Box2D sets a maximum of 8 vertices per polygon, so we limit the size of our loads accordingly. To emulate the fact that partially occluded ArUco markers may still be detected, we only model the centre of lifting point markers, as circles with a 10mm radius. Any contact of the circles with the platform cameras of the robots is interpreted as a successful detection of the lifting point marker. We do not model the remaining guiding markers and assume that any contact between the body of a load and a platform camera results in their detection. Finally, we simplify the model of the loads by excluding any supporting structure, such as legs.

Following Jones [41], we set a coefficient of friction of $\mu_{bodies} = 0.15$ and a coefficient of restitution of $e = 0.1$ to simulate the collisions between robots and between robots and the arena walls. However, since all robot controllers include a fixed collision avoidance method, these parameters should not come into play.

## 8.2 Run stages

On every update loop of Box2D, the dynamics of all the bodies in the environment is calculated. This includes the forces resulting from the actuation of the wheels of our robots, as well as the collisions between items in the environment. The update loop of Box2D is set to run at a frequency of 30Hz, using 8 iterations for the velocity constraint solver and 3 iterations or the position constraint solver. These settings are recommended by the documentation and result in a valid performance of the simulator for our purposes.

To further simplify the simulation process, we do not consider any kind of noise effects on the sensors or the actuators. This increases the sensitivity of the controllers to the reality gap, since the evolved controllers may exploit inaccuracies of our simulated environment [59]. The addition of noise should therefore be considered in future work.

Every $t_{update}$, we compute one iteration of the cycle shown in Figure 7.15, where each stage is computed for every robot before moving on to the next stage. We now describe each of the stages of the loop in the order in which they are executed, starting with the update of the sensor readings and ending with the actuation of the wheels and the platform.

### 8.2.1 Sense

In the sensing phase, each robot updates its sensor readings to obtain the input variables of the reference model (without the communications extension).

The readings of the proximity sensors are obtained using ray casts emitted from the edge of the body of the robot. When the lifting platform of the robot is down, the ray casts can only detect the arena and the bodies of other robots. However, in order to avoid collisions between loads being carried and other loads, we allow the robots to use ray-casting to detect other loads when its platform is up.

While this added capability of porters does not correspond with the reality of the environment originally defined, we consider it an acceptable compromise that should not affect the validity of the results of this work. We argue this by claiming that, by allowing the robot to sense other loads, we are essentially introducing temporary obstacles in the arena, and thus we are indirectly testing the adaptability of the swarm in a dynamic environment.

In a brute force approach, we would perform 16 ray casts per robot (one per sensor) on every iteration. This method, however, is computationally expensive. We therefore implement a preliminary step, where we generate an axis-aligned bounding-box (AABB) centred on the body of the robot, with a size equal to $2(R + p_{max})$, and check if any of the AABBs with which it overlaps correspond to other bodies that may be detected by the ray casts. The 16 ray casts are then performed only for those robots where such a body is detected.

After obtaining the orientation and heading of each robot, the camera feeds are processed to obtain the values of the remaining entries of the reference model. In order to save on computational resources, we make here another simplification by not considering occlusion in the FOV of the body cameras, which would require additional ray casting. To detect another robot or a nest marker, we simply check for overlap of the corresponding fixtures. If an overlap is found, the corresponding distance between the fixtures is queried.

Consequently, our implementation of the body cameras corresponds more closely to the simulation of an overhead vision system, which keeps track of the positions of all the robots. Since this kind of system is commonly used in swarm robotics implementations, we also deem this an acceptable compromise.

Finally, the lifting point markers are also detected by checking for overlap between the load fixture and the projection of the platform camera plane.

### 8.2.2 Control

At the control stage, the blackboard of each robot is updated and their BT controllers are ticked to generate the $\boldsymbol{v_{vote}}$ and $p_{vote}$ outputs.

### 8.2.3 Broadcast

The broadcast stage is the first stage of the negotiation strategy. During the broadcast stage, every robot clears its message buffer and broadcasts a message with the structure specified in Equation 7.1. Equivalently, every robot gathers the messages of those neighbours within the $r_{comms}$ range.

### 8.2.4 Process

Processing the received messages is done in two stages. In the first stage, the group ID of each robot is set according to the rules defined in Section 7.1.1. In the second stage, Equations 7.17 and 7.18 are used to compute $\boldsymbol{v_{goal}}$ and $p_{goal}$, respectively.

### 8.2.5 Act

The final stage consists on applying $\boldsymbol{v_{goal}}$ and $p_{goal}$ to each robot and move the loads, if necessary.

**Wheels** Since robots are modelled as dynamic objects, we simulate their motion by applying forces to them. For every value of $\boldsymbol{v_{goal}}$, we first saturate its magnitude to 1.0 to avoid setting infinite velocities and then scale it using $v_{max}$ to obtain the target local velocity. We then use an overdamped proportional controller to determine the forces required to move the robot with a velocity equal to $\boldsymbol{v_{goal}}$ while preventing unrealistic accelerations.

**Platform** We simplify the motion of the lifting platform by simulating it as a discrete event. If the platform is down, a positive value of $p_{vote}$ lifts it in a single iteration. Likewise, a negative value of $p_{vote}$ lowers a lifted platform. In any other case, the platform does not move.

Due to the lifting rules enforced, when a robot lifts its platform, it does it to lift a load. Therefore, we also attach the body of the corresponding load to each porter and add the corresponding fraction of the mass of the load to the mass of the porter. The load is held in position due to friction between the surface of the platform and the body of the load. Additionally, since a lifted platform is pressed against the body of the load, the platform camera is no longer able to detect the lifting point marker, so we remove the corresponding fixture.

Similarly, a platform is only lowered to deposit a load. Therefore, the load is detached, the mass of the porters is reset and the platform camera fixture is recovered. Moreover, if the platform is lowered when one of the robots is at the nest, the load is removed from the arena.

## 8.3 Fitness

When a run finishes, the fitness of the controller is evaluated. We consider a homogeneous swarm, where all the robots use the same controller, so we define the fitness in terms of global quantities of the environment. For the task of collective transport, we want to reward the proximity of each load to the nest region at the end of the run. Since the nest is located in the $+X_0$ wall of the arena, this is equivalent to rewarding motion of the loads in the $+X_0$ direction:

$$f_{1,i} = \frac{X_{0,i,final} - X_{0,i,initial}}{t_{life,i} V_{max}}, \tag{8.1}$$

where $X_{0,i,initial}$ and $X_{0,i,final}$ are the start and end positions of load $i$, $t_{life,i}$ is its lifetime, which is less than the simulation time only if the load is deposited at the nest at some point, and $V_{max}$ is the maximum velocity of the robots. The value of $f_{1,i}$ is therefore in the range $[-1, 1]$.

To aid the evolutionary process, we introduce two additional terms in our fitness function. With the first term, we penalise loads that are never lifted during a run:

$$f_{2,i} = \begin{cases} -1, & \text{if load } i \text{ is never lifted} \\ 0, & \text{otherwise.} \end{cases} \tag{8.2}$$

The second term rewards finding the lifting points of a load. This term is necessary for the robots to actively group under a load in order to transport it. We define it as

$$f_{3,i} = \frac{t_{cover,i}}{t_{life,i}}, \tag{8.3}$$

where $t_{cover,i}$ is the mean time any of the lifting points of load $i$ spend with an agent directly under them. Thus, $f_{3,i}$ is in the range $[0, 1]$.

The total fitness of the run is then

$$F = \sum_i (f_{1,i} + f_{2,i} + f_{3,i}), \tag{8.4}$$

where, considering an environment with $n$ loads, $F \in [-n, 2n]$.

## 8.4 Conclusions

In order to quickly evaluate the performance of our generated controllers, we build a 2D physics-based simulator in C++, through which the user may construct an environment that is apt for the task of collective transport. In every step, the simulator

updates the sensory inputs of each robot, implements the controller described in Section 7 and applies forces to the environment according to the resulting outputs.

We also define the fitness function that the evolutionary algorithm described in the following section will use to assess the performance of the controllers.

# 9 Evolution

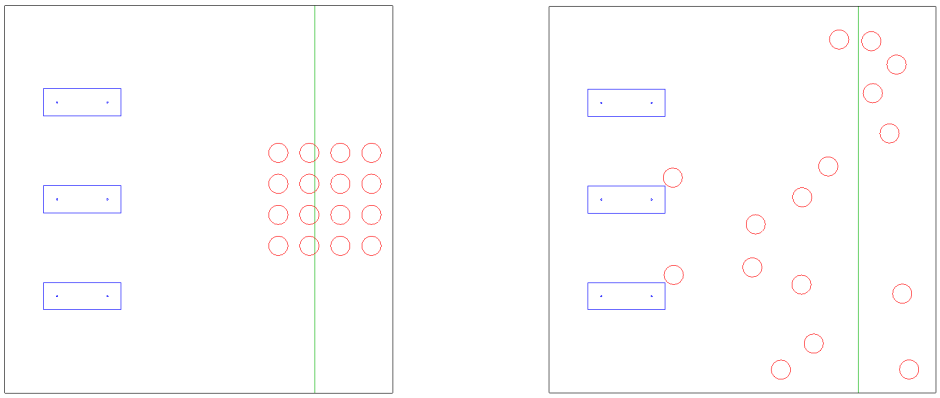
For our implementation of GP, we use openGA, a C++ library developed by Mohammadi [60]. While originally developed for genetic algorithms (GA), the library can also tackle GP problems.

It offers two main features of interest for our task. First, it allows the user to define the structure of the population individuals (chromosomes) and construct the mutation and crossover operators accordingly. And second, it can evaluate each chromosome in a separate thread and thus significantly reduce the computation time required per generation. We also need to consider that, instead of maximising the fitness of the population every generation, openGA minimises a cost function. We account for this difference by setting the cost of each individual to its negative fitness.

Each chromosome is defined as a structure containing genetic material and its cost. We define the genetic material as our BT, structured as a vector of nodes. The use of vectors to define a BT makes the implementation of the genetic operators straightforward. Each node is itself a structure containing the name, depth, arity and parameters of a BT node. It also contains a macro with the corresponding XML form of the node. This structure allows us to treat control-flow nodes, execution nodes and subtrees equally.

All the control-flow nodes introduced in Sections 4.1.1 are made available to the GP algorithm. However, in order to limit the search space, we limit the maximum number of children that *sequence*, *selector*, *sequence** and *selector** nodes may have to four. Besides the nodes defined in Section 7.2.2, the constituent behaviours and conditions described in Section 7.2.3 are also included in the set of execution nodes available.

The parameters we use as a starting point are shown in Table 9.1. Our choices are based on the works of Jones [41], Poli *et al.* [44] and Francesca *et al.* [16]. The initial population is generated using Koza's ramped-half-and-half method, as described in Section 5, with depth parameters ranging between $d_{min}$ and $d_{max}$. For each new



(a) $t = 0$ (b) $t = t_{explore}$

Figure 9.1: Environment instance for controller evolution. Three 2-porter loads (blue), 16 agents (red) and a nest region (green).

generation, a fraction of the individuals, determined by $f_{crossover}$, are generated by crossover of two parent individuals selected by rank selection. These individuals then undergo parameter, point and subtree mutation with probabilities $p_{param}$, $p_{point}$ and $p_{subtree}$, respectively. The fittest $n_{elite}$ individuals of the previous generation and the new individuals combined are transferred directly to the next generation, and the rest of the population is obtained by reproduction via rank selection from the same pool.

To evaluate the fitness of each controller, we choose an instance of the environment with three 2-porter loads placed on the left-hand side of the arena and 16 robots on the right-hand side of the arena, as shown in Figure 9.1a. The robots are initially allowed to run the *exploration* behaviour during $t_{explore}$ in order to obtain a random initial configuration for every run, such as the one shown in Figure 9.1b. Each controller is then run on all the robots during $t_{sim}$, and the resulting fitness is stored. This process is repeated $n_{evals}$ times, and the total fitness is computed as the mean of all evaluations. While $n_{evals}$ is usually in the order of 8-10, we reduce this value to economise computational resources.

Since it is common to observe jumps in fitness as the evolution progresses, we decide to let the GP algorithm run all $n_{gen}$ generations, even if the fitness converges to a fixed value.

Table 9.1: Initial GP parameters.

| Parameter | Value | Description |
|---|---|---|
| $n_{gen}$ | 200 | Number of generations |
| $t_{sim}$ | 120s | Simulation time |
| $t_{explore}$ | 10s | Initial exploration time |
| N | 20 | Population size |
| $d_{min}$ | 0 | Minimum tree depth (for initialisation) |
| $d_{max}$ | 4 | Maximum tree depth (for initialisation) |
| $p_{inner}$ | 0.9 | Probability of inner node selection |
| $p_{param}$ | 0.05 | Probability of parameter mutation |
| $p_{point}$ | 0.05 | Probability of point mutation |
| $p_{subtree}$ | 0.1 | Probability of subtree mutation |
| $f_{crossover}$ | 0.7 | Crossover rate |
| $n_{evals}$ | 4 | Number of evaluations |
| $n_{elite}$ | 3 | Number of elite individuals |

# 10 Results

Initial trials with the primitive set and the parameters defined in Section 9 revealed that trees with a depth of zero never achieved the goal of collective transport. This is a consequence of the absence of a behaviour that may modify both $\boldsymbol{v_{vote}}$ and $p_{vote}$, which results in controllers requiring at least two behaviours, and therefore a minimum depth of one, to achieve collective transport. Additionally, the controllers generated with the $[0, 4]$ initial depth range tended to be too shallow, which we believe to be caused by the introduction of covariant parsimony pressure. We therefore increase the initial tree depth range to $[1, 5]$.

We also observed that all recruitment behaviours and entries were always filtered out by the evolution. Consequently, we removed the *recruitment* and *anti-recruitment* behaviours and the *recruiter-count* and *inverted-recruiter-count* conditions from the primitive set, and we deleted the $\boldsymbol{v_{recr}}$ and $s_r$ entries from the blackboard in order to speed up future runs.

Having implemented these modifications, we ran the evolutionary algorithm five times. For each generation, we recorded the original fitness of the fittest individual, that is, $f(x)$ in Equation 5.1, where $f(x)$ is given by Equation 8.4, and the average fitness of all the individuals. We then normalised the results to obtain a more intuitive interpretation of the data. The results are shown in Table 10.1.

Figure 10.1 then shows the normalised original fitness of the fittest individuals

Table 10.1: Normalised original fitness of the fittest individual and average fitness of all individuals in the last generation during five independent evolutionary runs.

| Run | Best | Average |
|---|---|---|
| 1 | 0.587 | 0.567 |
| 2 | 0.556 | 0.525 |
| 3 | 0.600 | 0.584 |
| 4 | 0.609 | 0.568 |
| 5 | 0.570 | 0.570 |

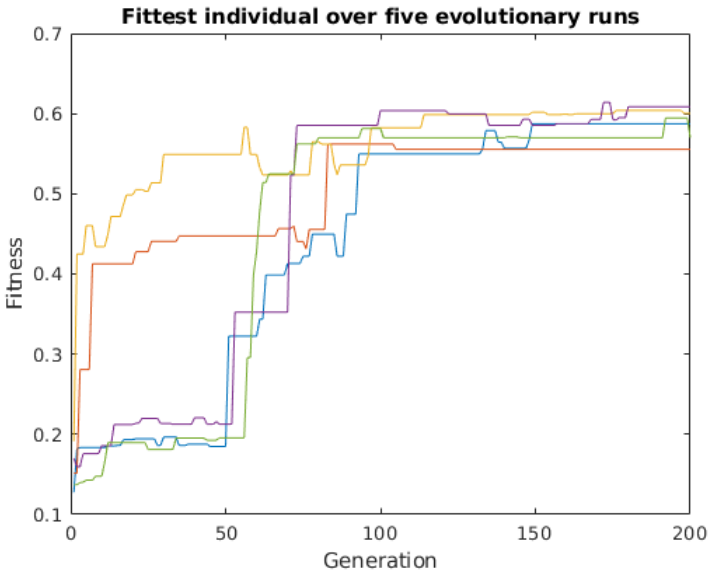



Figure 10.1: Five independent evolutionary runs with modified parameters. The normalised original fitness of the fittest individuals is shown.

of each generation. Note that, since the covariant parsimony pressure coefficient changes every generation, the fittest individual may not be the one with the highest original fitness. Furthermore, the fittest individual of a generation may no longer be as fit as the fittest individual from previous generations. Consequently, the obtained fitness curves are no longer monotonically increasing.

Nevertheless, all five runs show that the GP algorithm is able to generate controllers that partially complete the task of collective transport. In two of those runs, the robots are able to start transporting loads almost from the start, while the other three require 50 generations to reach that stage. By the 100th generation, the fittest individuals of all the populations are able to transport some of the items towards the nest. Figure 10.2 shows two examples in which the robots successfully manage to lift and transport multiple items simultaneously. For links to some videos of the

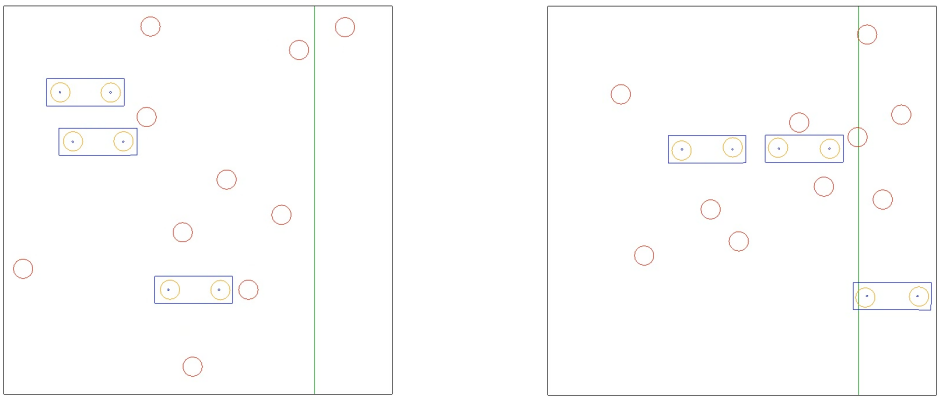


Figure 10.2: Successful collective transport of multiple loads simultaneously towards the nest region.

performance of the controllers, see Appendix 13.

However, none of the controllers generated were able to deposit any loads at the nest. As a consequence, we observed that robots often reach the nest region, but then return to the left-hand side of the arena, thus obtaining a lower reward. Since the fitness function in Equation 8.4 rewards loads with a shorter lifetime, we believe that further tuning of the evolutionary parameters would achieve the complete desired behaviour.

# 11 Planning

Figure 11.1 shows a Gantt chart with the plan to develop the work presented. The columns indicate the weeks of the corresponding months. Tasks are shown in blue and milestones, in gold.

The design and implementation of the controllers, the simulator and the evolutionary algorithm, together with the writing of this report, constitute the four main tasks of the project. We therefore allocate most of the time to their execution. Since testing our solution requires waiting several hours for the simulations to run, the report should be completed during this time.

We also identify two milestones. The first one is a progress report near the end of March in order to assess the evolution of the project. The second one is the submission of the project.

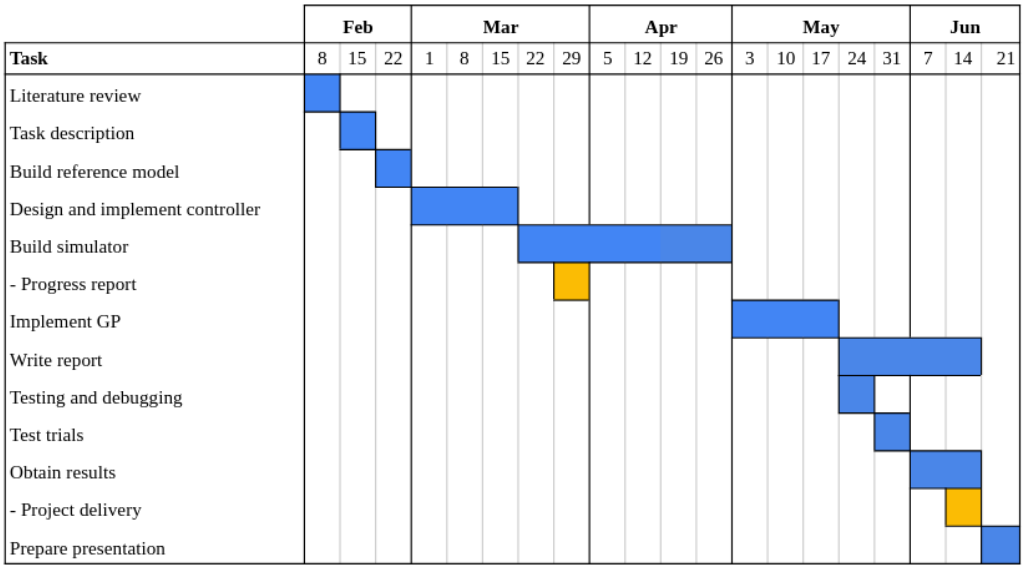



Figure 11.1: Gantt chart of project plan.

# 12 Budget analysis

This project was fully developed in a simulated environment. The only kind of hardware required is therefore a computer. While there are no minimum requirements for the amount of processing power required, an architecture with multiple cores is recommended, since openGA separates the evaluation of the population individuals in as many threads as possible. Thus, for a population with 20 individuals, like the one considered in this work, the GP algorithm would ideally have access to 11 cores (22 threads, with 20 for the individual evaluations and 1 for the main program).

Processors with more than 11 cores are not uncommon. The Intel® Core™ X-Series Processor Family[3] offers three alternatives, all of which cost between $689 and $1,000. The AMD Ryzen™ Threadripper™ Processors[4] also offer multiple options with up to 64 cores and prices ranging between $285[5] and $5,121[6]. Without any further assumptions about platform requirements, a $2,000 desktop computer should be able to comfortably run the GP algorithm in a minimum time. A more expensive solution would allow for mutliple concurrent runs of the GP algorithm.

However, in order to run a large number of evolutionary processes simultaneously,

[3]https://www.intel.com/content/www/us/en/products/details/processors/core/x.html
[4]https://www.amd.com/en/products/ryzen-threadripper
[5]https://www.amazon.com/AMD-Threadripper-24-thread-Processor-YD192XA8AEWOF/dp/B074CBJHCT
[6]https://www.amazon.com/AMD-Ryzen-Threadripper-3990X-128-Thread/dp/B0815SBQ9W

a computer cluster would be a highly valuable asset. Depending on the size of the cluster, it may allow for dozens of simultaneous runs, which would speed up the parameter tuning process. Unfortunately, computer clusters are typically privately owned, and thus access to them is restricted.

All the external libraries used are free and open source and are distributed under permissive licenses. Both Box2D and BehaviorTree.CPP use the MIT License, while openGA uses the Mozilla Public License 2.0. Consequently, no software expenses need to be considered.

# 13 Environmental impact

Given that the project was developed exclusively in a simulated environment, we have no environmental concerns to report. Nevertheless, we may consider the effect that the implementation of the solution presented in a real-world industrial environment could have. To do this, we refer to Section 2.2, were we point out that an industrial robot swarm capable of collective transport could help prevent harm to plant workers, reduce costs of labour and increase the reliability of the plant.

# Conclusions

In this work, we have presented a method for collective transport of heavy items using an industrial swarm in a simulated environment. The robots use controllers with a BT architecture, which are evolved with GP techniques, to generate actuator commands. They then coordinate using a decentralised negotiation strategy that relies on direct communication.

The results show that the GP algorithm is able to generate controllers that are able to safely lift and transport multiple loads simultaneously towards a nest region. No controllers were able to deposit the loads at the nest, however, so we suggest further tuning of the evolutionary parameters as a potential solution.

A clear next step in the development of the method proposed would be to implement it in a real-world environment in order to assess its sensitivity to the reality gap. Prior to this, however, it would be worth considering the limitations of the simulator we have used. As we mention in Section 8, the effect of noise on sensor readings and actuator outputs is not considered. As a consequence, the evolved controllers

may be using inaccuracies of the simulated environment to their advantage. A more accurate model of the sensors and the actuators, where noise effects are considered, would thus result in more realistic simulations.

We also do not consider the possibility of failure during message delivery and reception. While this greatly simplifies the implementation of the negotiation strategy, it also reduces its fault-tolerance. The inclusion of a mechanism to deal with these failures would therefore be crucial before attempting to transfer the controllers to the real world.

Finally, we identify the requirement of a fully-connected porter network as an important limitation of our method. Future work could introduce a method relying on communication hops to determine when each robot has received all the information required to compute the actuator outputs. Such a method would also need to consider the required messaging rate and the amount of data contained in each message.

Despite these limitations, we believe that the proposed method has potential for practical applications of swarm robotics in real-world industrial environments. The use of direct communication for decentralised negotiation results in safe handling of the transported items, since every porter executes the exact same motion. Additionally, its implementation in an already existing environment would not require any additional infrastructure, since the robots use ArUco markers to navigate.

# Acknowledgements

I would like to thank my supervisor, Dr. Simon Jones, for his support and guidance throughout this project.

I would also like to thank Dr. Sabine Hauert and the rest of the swarm robotics group at Bristol Robotics Laboratory for letting me be a part of their team during this time.

Finally, I would like to thank Dr. Ramon Costa Castelló for helping me make this project possible in the first place.

# Bibliography


[1] SAHIN, E. "Swarm Robotics: From Sources of Inspiration to Domains of Application". In: vol. 3342. Jan. 2005, pp. 10–20.

[2] DORIGO, M. et al. "Swarmanoid: A Novel Concept for the Study of Heterogeneous Robotic Swarms". In: *IEEE Robotics Automation Magazine* 20.4 (2013), pp. 60–71.

[3] BENI, G. "From Swarm Intelligence to Swarm Robotics". In: *Swarm Robotics*. Ed. by Şahin, E. and Spears, W. M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–9.

[4] BJERKNES, J. D. and WINFIELD, A. F. T. "On Fault Tolerance and Scalability of Swarm Robotic Systems". In: *Distributed Autonomous Robotic Systems: The 10th International Symposium*. Ed. by Martinoli, A. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 431–444.

[5] BONABEAU, E., DORIGO, M., and THERAULAZ, G. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[6] Seeley, T. "Honey bee foragers as sensory units of their colonies". In: *Behavioral Ecology and Sociobiology* 34 (2004), pp. 51–62.

[7] REID, C. R. et al. "Army ants dynamically adjust living bridges in response to a cost–benefit trade-off". In: *Proceedings of the National Academy of Sciences* (2015).

[8] BARDUNIAS, P. et al. "The extension of internal humidity levels beyond the soil surface facilitates mound expansion in Macrotermes". In: *Proceedings of the Royal Society B: Biological Science* 287.1930 (2020).

[9] HAMANN, H. *Swarm Robotics: A Formal Approach*. Jan. 2018.

[10] REYNOLDS, C. W. "Flocks, Herds and Schools: A Distributed Behavioral Model". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 25–34.

[11] BRAMBILLA, M. et al. "Swarm robotics: a review from the swarm engineering perspective". In: *Swarm Intelligence* 7 (2012), pp. 1–41.

[12] SCHRANZ, M. et al. "Swarm Robotic Behaviors and Current Applications". In: *Frontiers in Robotics and AI* 7 (2020), p. 36.

[13] KUBE, C. R. and BONABEAU, E. "Cooperative transport by ants and robots". In: *Robotics and Autonomous Systems* 30.1 (2000), pp. 85–101.

[14] KOLAY, S., BOULAY, R., and D'ETTORRE, P. "Regulation of Ant Foraging: A Review of the Role of Information Use and Personality". In: *Frontiers in Psychology* 11 (2020), p. 734.

[15] FLOREANO, D., HUSBANDS, P., and NOLFI, S. "Evolutionary Robotics". In: *Springer Handbook of Robotics*. Ed. by Siciliano, B. and Khatib, O. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1423–1451.

[16] FRANCESCA, G. et al. "AutoMoDe: A novel approach to the automatic design of control software for robot swarms". In: *Swarm Intelligence* 8 (2 2014), pp. 89–112.

[17] MIGLINO, O. "Evolving mobile robots in simulated and real environments". In: *Artificial Life* 2 (1995), pp. 417–434.

[18] JAKOBI, N., HUSBANDS, P., and HARVEY, I. "Noise and the reality gap: The use of simulation in evolutionary robotics". In: *Advances in Artificial Life*. Ed. by Morán, F. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 704–720.

[19] JAKOBI, N. "Evolutionary Robotics and the Radical Envelope-of-Noise Hypothesis". In: *Adaptive Behavior* 6 (1997), pp. 325–368.

[20] STATISTICS, U. B. O. L. *Industries at a Glance: Warehousing and Storage: NAICS 493*. `https://www.bls.gov/iag/tgs/iag493.htm#`. 2021.

[21] ROODBERGEN, K. J. and VIS, I. F. "A survey of literature on automated storage and retrieval systems". In: *European Journal of Operational Research* 194.2 (2009), pp. 343–362.

[22] AZADEH, K., DE KOSTER, R., and ROY, D. *Robotized Warehouse Systems: Developments and Research Opportunities*. Tech. rep. ERS-2017-009-LIS. May 2017.

[23] YUAN, Z. and GONG, Y. "Improving the Speed Delivery for Robotic Warehouses**This research is supported by China Scholarship Council and Chutian Scholarship." In: *IFAC-PapersOnLine* 49.12 (2016). 8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016, pp. 1164–1168.

[24] JONES, S. et al. "Distributed Situational Awareness in Robot Swarms". In: *Advanced Intelligent Systems* 2.11 (2020), p. 2000110.

[25] MATARIC, M., NILSSON, M., and SIMSARIN, K. "Cooperative multi-robot box-pushing". In: *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*. Vol. 3. 1995, 556–561 vol.3.

[26] CAMPO, A. et al. "Negotiation of goal direction for cooperative transport". In: *ANTS 2006: Ant Colony Optimization and Swarm Intelligence*. Lecture Notes in Computer Science. 4150 (2006), pp. 191–202.

[27] FERRANTE, E. et al. "Socially-Mediated Negotiation for Obstacle Avoidance in Collective Transport". In: *Distributed Autonomous Robotic Systems: The 10th International Symposium*. Ed. by Martinoli, A. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 571–583.

[28] GROSS, R. and DORIGO, M. "Evolution of Solitary and Group Transport Behaviors for Autonomous Robots Capable of Self-Assembling". In: *Adaptive Behavior* 16.5 (2008), pp. 285–305.

[29] BALDASSARRE, G., NOLFI, S., and PARISI, D. "Evolution of Collective Behavior in a Team of Physically Linked Robots". In: vol. 2611. Feb. 2003.

[30] *Cooperative Object Transport Using Evolutionary Swarm Robotics Methods*. Vol. ECAL 2015: the 13th European Conference on Artificial Life. ALIFE 2020: The 2020 Conference on Artificial Life. July 2015, pp. 464–471. eprint: `https://direct.mit.edu/isal/proceedings-pdf/ecal2015/27/464/1903925/978-0-262-33027-5-ch083.pdf`.

[31] HAMOUDA, A. I. "Cooperative Transport in Swarm Robotics". MA thesis. The American University in Cairo, 2018.

[32] COUCEIRO, M. S. et al. "A fuzzified systematic adjustment of the robotic Darwinian PSO". In: *Robotics and Autonomous Systems* 60.12 (2012), pp. 1625–1639.

[33] COLLEDANCHISE, M. and ÖGREN, P. "Behavior Trees in Robotics and AI". In: (July 2018).

[34] IAN MILLINGTON, J. F. *Artificial Intelligence for Games*. CRC Press, 2009.

[35] ISLA, D. "Handling Complexity in the Halo 2 AI". In: *Game Developers Conference*. 2005.

[36] ISLA, D. "Building a Better Battle: Halo 3 AI Objectives". In: *Game Developers Conference*. 2008.

[37] JONES, S. et al. "Evolving Behaviour Trees for Swarm Robotics". English. In: *Distributed Autonomous Robotic Systems*. Springer Tracts in Advanced Robotics. Springer, Mar. 2018, pp. 487–501.

[38] LIGOT, A. et al. "Automatic modular design of robot swarms using behavior trees as a control architecture". In: *PeerJ Computer Science* 6 (Nov. 2020), e314.

[39] KUCKLING, J., VAN PELT, V., and BIRATTARI, M. "Automatic Modular Design of Behavior Trees for Robot Swarms with Communication Capabilites". In: *Applications of Evolutionary Computation*. Ed. by Castillo, P. A. and Jiménez Laredo, J. L. Cham: Springer International Publishing, 2021, pp. 130–145.

[40] MARZINOTTO, A. et al. "Towards a unified behavior trees framework for robot control". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 5420–5427.

[41] JONES, S. "Onboard Evolution of Human-Understandable Behaviour Trees for Robot Swarms". PhD thesis. University of Bristol, 2020.

[42] VIKHAR, P. A. "Evolutionary algorithms: A critical review and its future prospects". In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. 2016, pp. 261–265.

[43] SIMON, D. *Evolutionary Optimization Algorithms*. Wiley, 2013.

[44] POLI, R., LANGDON, W. B., and MCPHEE, N. F. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[45] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[46] KOZA, J. R. et al. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Springer US, 2003.

[47] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. second edition, 1992. Ann Arbor, MI: University of Michigan Press, 1975.

[48] MILLER, B. L. and GOLDBERG, D. E. "Genetic Algorithms, Tournament Selection, and the Effects of Noise". In: *Complex Systems* 9 (1995), pp. 193–212.

[49] BAKER, J. E. "Reducing Bias and Inefficiency in the Selection Algorithm". In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Cambridge, Massachusetts, USA: L. Erlbaum Associates Inc., 1987, pp. 14–21.

[50] POLI, R., MCPHEE, N., and VANNESCHI, L. "Elitism reduces bloat in genetic programming". In: Jan. 2008, pp. 1343–1344.

[51] POLI, R. and MCPHEE, N. "Covariant Parsimony Pressure for Genetic Programming". In: (Feb. 2008).

[52] JONES, S. et al. "Onboard Evolution of Understandable Swarm Behaviors". In: *Advanced Intelligent Systems* 1.6 (2019), p. 1900031.

[53] GARRIDO-JURADO, S. et al. "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6 (2014), pp. 2280–2292.

[54] FACONTI, D. and COLLEDANCHISE, M. *BehaviorTree.CPP*. `https://github.com/BehaviorTree/BehaviorTree.CPP/`. 2018.

[55] QUIGLEY, M. et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009.

[56] FACONTI, D. *Groot*. `https://github.com/BehaviorTree/Groot`. 2018.

[57] CATTO, E. *Box2D, A 2D Physics Engine for Games*. `https://github.com/erincatto/box2d`. 2006.

[58] SEGAL, M. and AKELEY, K. *The OpenGLR Graphics System: A Specification (Version 3.3 (Core Profile))*. The Khronos Group Inc. 2010.

[59] LIGOT, A. and BIRATTARI, M. "Simulation-only experiments to mimic the effects of the reality gap in the automatic design of robot swarms". In: *Swarm Intelligence* 14.1 (Oct. 2019), pp. 1–24.

[60] Mohammadi, A. et al. "OpenGA, a C++ Genetic Algorithm Library". In: *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*. IEEE. 2017, pp. 2051–2056.