ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

# Automatic modular design of control software in robot swarms
## Towards exploitation of behaviour trees features

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en informatique à finalité spécialisée

**Vincent van Pelt**

DEMIURGE
Automatic Design of Robot Swarms
An ERC Consolidator Project

Directeur
Professeur Mauro Birattari

Superviseur
Jonas Kuckling

Service
IRIDIA

Année académique
2019 - 2020

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme: DEMIURGE Project, grant agreement No 681872

## Abstract

In the past years, AutoMoDe has been introduced as a novel approach to the design of robot swarms. In AutoMoDe, an optimisation algorithm assembles and tunes predefined modules into control software, maximizing the score of the swarm against a particular mission-specific objective function assessed in a simulation environment. AutoMoDe achieves a relatively smooth transition between simulation and real world by introducing bias using only high-level modules.

A version of AutoMoDe, called `Maple`, introduced behaviour trees as a possible control software structure for assembling modules, reusing the modules, optimisation algorithm and missions of previous AutoMoDe versions. This work pursues the exploration of behaviour trees by defining a new version of AutoMoDe, called `Cedrata`. `Cedrata` introduces a new set modules, specifically created to be used in behaviour trees, and assesses the performance of behaviour trees in new missions. Experiments include comparison between `Maple` and `Cedrata`, but also against manual designs.

Results give interesting insights. Behaviour trees and the set of modules allow human designers to reach good performances in a small amount of time. One some missions, `Cedrata` reaches the best performances; but on some other missions the optimisation algorithm seems to fail in finding the optimal solution. Results also demonstrate that the freedom given on the tree structure have an impact on the quality of the generated control software.

**Résumé**

Ces dernière années, AutoMoDe a été présenté comme une nouvelle approche pour la conception d'essaims de robots. Dans AutoMoDe, un algorithme d'optimisation assemble et paramètre des modules prédéfinis en un logiciel de contrôle, tout en maximisant le score obtenu par l'essaim dans une mission donnée exécutée en simulation. Les logiciels peuvent ensuite être portés sur de vrais robots avec relativement peu de différences en terme de performances grâce au biais introduit par AutoMoDe qui encourage l'utilisation de modules de haut niveau.

Une version d'AutoMoDe, appelée `Maple`, a introduit les arbres de comportement comme une structure viable pour assembler les modules. `Maple` réutilise les modules, l'algorithme d'optimisation et les missions de versions précédentes d'AutoMoDe. Dans ce document, une nouvelle version d'AutoMoDe appelée `Cedrata` poursuit l'étude des arbres de comportement. `Cedrata` introduit un nouvel ensemble de modules, construits spécifiquement pour les arbres de comportement, et évalue ces arbres dans de nouvelles missions. Les expériences présentées proposent une comparaison entre `Cedrata` et `Maple`, mais aussi avec des logiciels conçus manuellement.

Les résultats aboutissent à des observations intéressantes. Les arbres de comportement et l'ensemble de modules permettent de concevoir des logiciels de manière aisée en peu de temps. Sur certaines missions, `Cedrata` atteint les meilleures performances ; sur d'autres l'algorithme d'optimisation échoue dans la recherche de la meilleure solution. Les résultats montrent en outre que la liberté donnée à la structure des arbres de comportement influe sur la qualité des logiciels générés.

# Contents

# Chapter 1

# Introduction

Swarm intelligence, as defined by Dorigo, Birattari, et al. [15], is “the discipline that deals with natural and artificial systems composed of many individuals that coordinate using decentralized control and self-organization”.

In swarm intelligence, groups of simple behaving individuals achieve complex behaviours without central control. In nature, such kind of systems include colony of ants, termites and bees, flocks of birds, herds, etc [55].

The collective behaviour emerges from the interactions between the individuals and their environment and, more importantly, the local interactions between individuals. These interactions can be of multiple kind: direct communication (message exchange), indirect communication through the presence or absence of other individuals or through the environment. The last one is called *stigmergy* [24]. For example, ants achieve to find routes between food sources and the nest by deposit of pheromones. Through feedback mechanisms, pheromones trails get reinforced and allow ants to navigate more efficiently in their environment. Such swarms are generally relatively homogeneous, i.e. individuals are all indistinguishable from each other or belong to a few typologies.

Swarm intelligence systems present interesting properties [49]:

**Robustness:** The swarm is able to pursue its task when an individual is lost or unable to function properly, even if it is often at the cost of some efficiency. Indeed, no individual is absolutely required to achieve the target goal, and they can be replaced by others when a loss or another problem occurs. As cooperation is an emergent effect of the swarm, even parts of homogeneous swarms can be removed without provoking the discontinuation of the operations. This also holds for heterogeneous swarms, as long as there is sufficient robots of each typology. The multitude and simplicity of individuals also reduce the chance of error at the collective level. Simple individuals are less error-prone than complex ones, and the multitude reduce the risk of missing information in the environment.

**Flexibility:** The swarm is capable to adapt its collective behaviour depending on the environmental conditions. Furthermore, it allows the swarm to perform parallel tasks, by organizing itself in multiple teams where individuals, spatially separated, create different collective behaviours. That way, a single swarm can tackle different aspects of a complex mission.

**Scalability:** The swarm can handle the addition or removal of individuals without changing how the parts interacts between themselves. Individuals only interacts with other *locally*, meaning that the number of interactions of an individual will not tend to grow with the size of the swarm population. It means that in some tasks, without the need to update existing individuals, a swarm intelligence system performance can be improved simply by adding individuals.

These three properties, in addition of the simplicity of the individuals, make swarm intelligence systems appealing for applications in various engineering fields. The study of domains were swarm intelligence principles are applied to create new systems is called *swarm engineering*.

Swarm intelligence includes *swarm optimisation* [7] and *swarm robotics* fields [3]. In swarm optimisation, algorithms are designed using the swarm intelligence principles to solve optimisation problems. This include algorithms like Ant Colony Optimisation (ACO) originally developped by Dorigo, Colorni, and Maniezzo [16] or Particle Swarm Optimisation (PSO) by Kennedy and Eberhart [33].

## 1.1 Swarm Robotics

Swarm robotics is an approach were we design at set of relatively simple robotic agents following the principles of swarm intelligence in order to achieve a task [2]. In particular, robots that implement the swarm intelligence principle have the following characteristics [9]:

- They are autonomous;
- They are localized in the environment and they can modify it (stygmergy);
- They have local-only sensing and communication capabilities;
- They have no global knowledge (no central control);
- They cooperate in order to achieve a collective task.

Ideally, robots are designed in a way these design concepts make them implement the swarm intelligence properties seen before: the swarm must be robust, flexible and scalable [49].

One of the difficulties is predicting how collective behaviours can emerge from local control software. It makes the designing such of robots a difficult task. In swarm robotics, the designer work on robot hardware and software is done at the individual level but the relevance and performance of the produced design are evaluated at the collective level.

Multiple approaches to the design of robot swarms have been studied in the past years, but we can separate them in two categories: manual design and automatic design.

In manual design, the designer iteratively implement the behaviour of the robots, evaluate the collective performance of the swarm in experiments against a particular task and improve the individual implementation based on the results of the experiments. The design stops when the collective behaviour that is obtained is sufficiently close the the desired one.

In automatic design, an optimisation algorithm takes the place of the designer and generate the control software of the robot. Various researchers have studied swarm robotics from an automatic design angle, and we can classify these researches using three criteria:

1. *Off-line* or *on-line* methods [5]. In off-line methods, robots are trained in a simulated environment and the obtained software is uploaded afterwards in real robots. In on-line methods, the software is continuously improved while real robots execute their task.

2. The design method. Two of them are mainly used to design control software: Reinforcement Learning (RL) [32] and Evolutionary Robotics (ER) [45]. In RL, robots receive a reward for each action that they take. Over time, they create a policy that map the robot state to the best action to do, i.e. the action that should give them the higher reward. In ER, robots control software are simulated in populations over multiple generations. The best control software are selected and improved over generations using principles inspired by the biological theory of evolution [12].

   Others design methods includes Novelty Search [23], a derivative of ER where new individuals are created in way they differentiate from the previous generations instead of exploiting best software so far. It makes the process divergent instead of convergent and thus promotes exploration of the search space. Racing algorithms [4] are an other alternative, were a set of candidates is evaluated multiple times and weak ones are progressively eliminated.

3. The control software structure. These structure include Neural Networks, Virtual Force functions, Probabilistic Finite State Machine (PFSMs), Behaviour Trees (BTs), etc.

## 1.2 Off-line design

In this work, we are interested in *automatic off-line design* [5]. In automatic off-line design, the problem of designing control fortware for robot swarm is transformed into an optimisation problem. The software is then uploaded in the robots that operate in the real world. This method works as an opposite to on-line automatic design, were software is updated as robots already started their task in the target environment.

A method that can be used in off-line automatic design is Evolutionary Robotics. They often create control software in the form of Neural Networks [44].

Evolutionary Robotics has already been applied successfully to design control software. As examples, Sperati, Trianni, and Nolfi [53] used evolutionary robotics to design a robot swarm that discover paths between two locations. Hauert, Zufferey, and Floreano [27] designed control software for a swarm of aerial robots. Duarte et al. [17] designed neural control software for aquatic surface robots on multiple missions.

An evolutionary algorithm, in the context of automatic off-line design, works as follows:

1. An initial set of individuals is created, with randomly generated chromosomes.

2. Each individual is decoded into a robot control software, for example a neural network. In a simulated environment, robots controlled by the decoded software are evaluated against a specific collective task objective.

3. The individuals that lead to the best robots performance are selected. New individuals are generated from them using genetic rules, e.g. chromosomes exchanges and random mutations.

4. The algorithm restarts at step 2 and iterate for a specified number of generations or until a particular condition is met.

One of the concerns about automatic off-line design is called the *reality gap* [5]. The reality gap is the fact that it is generally difficult to predict the performance of a robot swarm that has been trained in a simulated environment, creating a “gap” between the performance in design and in experiences. Evolutionary Robotics methods often suffer from this gap [25].

## 1.3 AutoMoDe

Simulations are widely used to design the control software of robots swarm. They do not involve real robots, so they cancel the risk of material damage. They also are often quicker than real world experiments, as long as the experimenter have the sufficient computational power. However, from the robot point of view, the simulation environment is never strictly identical or indistinguishable from the real world environment, or even another simulation environment. A control software that makes the swarm perform a task adequately in simulation may not lead to the same result in an other environment. This is what the *reality gap* is about: the performance difference between simulation and real environment.

A swarm robotics design method should reduce the impact of the reality gap on the final task performance in real world. Some methods fail to overcome this gap, leading to less effective or even non-functional swarms [14].

Multiple techniques have been proposed to deal with this reality gap. Jakobi, Husbands, and Harvey [28] tuned the simulation sensors and actuator noise to reduce behaviour difference between simulation and real world. Zagal, Ruiz-del-Solar, and Vallejos [57] proposed a training method were simulations and real world experiments are done successively. Fitness difference between these experiments allow to adapt the parameters of the simulator.

Francesca et al. [20] proposed to look at the reality gap problem as a generalization problem as it can be done in machine learning. In machine learning, learning algorithms are subject to bias/variance trade-off [22]. When the learning algorithm present a low bias, variance on the results produced is high, meaning that the algorithm is highly sensible to the input values and present difficulties to generalize to new data. Robot control software can be seen as functions that map sensors input to actuators, and such functions are also subject to bias/variance trade-off, as illustrated by Lawrence, Tsoi, and Back [36]. A low bias in the control software may lead to a bad generalization and thus less efficient collective behaviour when the robots are deployed into a different environment. It is typically the case when the swarm is moved from simulation to real world, and this can explain why the reality gap cause problems. Francesca et al. [20] proposed AutoMoDe as a method that is inherently biased and thus mitigates the effects of the reality gap.

To achieve this mitigation, AutoMoDe (that stands for Automatic Modular Design) introduces predefined and tunable modules that act as bias and that must be combined into control software. The set of modules is designed according to the capabilities of the robots that are considered. These modules and the structure in which they are assembled depends on the particular study that have been conducted. These studies, that are called AutoMoDe *flavours*, include as a non exhaustive list: `Vanilla` [20], `Chocolate` [19], `Gianduja` [26], `Maple` [35], `Waffle` [50], `TuttiFrutti` [21], `Coconut` [52], `IcePop` [34].

These studies have shown that AutoMoDe flavours successfully manages to mitigate the reality gap, by comparison against an Evolutionary Robotics method that have been called `Evostick` [18].

## 1.4 Behaviour Trees

AutoMoDe flavours traditionally assemble their predefined modules into PFSMs. AutoMoDe-`Maple` introduced behaviour trees [39] as a possible control software structure.

Behaviour trees, originally developed for video games [10, 40] and recently used in swarm robotics [30, 51, 31, 42], are tree-like structures that implement *two-ways control transfers* [11]. When they are executed, a tick is propagated through the tree. As results of being ticked, a node execute a predefined behaviour and return a value describing the progression state of its task, allowing the parent node to act consequently.

A behaviour tree contains multiple types of nodes. Leaf nodes are either condition or actions nodes, that respectively test sensor input or execute an unitary task. Intermediate nodes are called control nodes. The way they tick their children and their return value depend of the return value of their children.

## 1.5 Scope of this work

This work sets up in the development and study of AutoMoDe by studying the potential use and exploitation of behaviour trees. This includes the creation of a new modules set that is specifically designed to be used in behaviour trees; defining new missions that highlight the possibilities of this new set; and analysing the performances of this new design method.

The Chapter 2 explains the concept of behaviour trees and gives an overview of how they are used today in swarm robotics. The AutoMoDe flavours on which this work is based are detailed in Chapter 3. The Chapter 4 lists related state of the art studies and explains differences between these studies and AutoMoDe-`Maple` and this work. The Chapter 5 lists the changes that this work introduce compared to the previous versions of AutoMoDe. The Chapter 6 gives a description of the tasks that will be used to evaluate the new AutoMoDe version developed in this work. The Chapter 7 explains the experiments conducted and the methods that have been used. The Chapter 8 describes the results obtained. Finally, the Chapter 9 concludes this work.

# Chapter 2

# Behaviour Trees

Behaviour trees are structures that offer a way to organise and execute different tasks in autonomous agents, such as robots [10].

This chapter explains how behaviour trees work, gives a short motivation of the concept and draws an overview on how they are used today in robotics.

## 2.1 Structure

The definition of behaviour trees that will be used is the one proposed by Marzinotto et al. [39], which is also used by `Maple`.

A behaviour tree is a tree-like structure that contains one root node, control nodes and execution nodes. Execution nodes are always leaf nodes, and control nodes are placed in between the root and the leaf nodes. The root node periodically generates a signal called *tick* that is propagated through the tree, in a way that depends of the control nodes. Once a node is ticked, it execute its algorithm and return one value that can be either *success*, *failure* or *running*. Execution nodes and are either *action nodes* or *condition nodes*, and control nodes are either *Sequence*, *Selector*, *Decorator* or *Parallel*. A behaviour tree example is shown in figure 2.1.

Condition nodes are tests on the outside world. They cannot alter the state of the agent that run the tree, and can return only *success* or *failure*. Action nodes are actions or behaviours that the agent can execute. They can alter the state of the agent and can return any value in *success*, *failure* or *running*. The latter means that the action is still being performed by the agent, indicating that the action will take more that a tick cycle to be executed. It is necessary that the action give control back to the tree in such cases, so the agent can react to external stimuli when executing the action.

Control nodes distribute the tick on their children and return *success*, *failure*

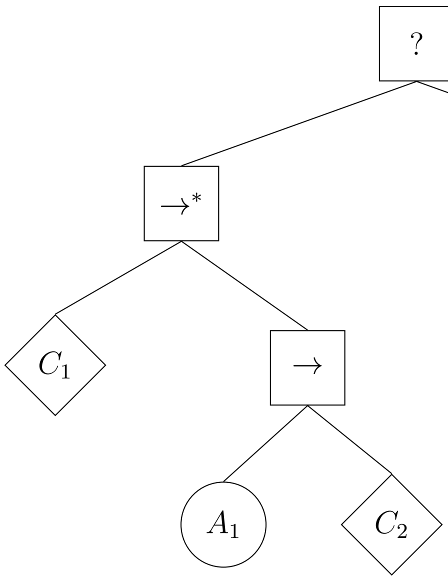


**Figure 2.1** – A behaviour tree example. Control nodes, condition nodes and action nodes are respectively represented by squares, diamonds and circles.

or *running* depending on the value returned by the ticked children.

**Selector** (?) When enabled, it ticks its children sequentially as long as they return *failure*. When a child returned *success* or *running*, it immediately stops and returns the children value. If all children returned *failure*, the node returns *failure*.

**Selector*** ($?^*$) Same as Selector, but if a child returned *running* at the previous execution, the node restarts with this child, ignoring the previous ones in the children list.

**Sequence** ($\rightarrow$) When enabled, it ticks its children sequentially as long as they return *success*. When a child returned *failure* or *running*, it immediately stops and returns the children value. If all children returned *success*, the node returns *success*.

**Sequence*** ($\rightarrow^*$) Same as Sequence, but if a child returned *running* at the previous execution, the node restarts with this child, ignoring the previous ones in the children list.

**Parallel** ($\rightrightarrows$) When enabled, it ticks all its children sequentially. If the number of succeeding children is greater than $S$, the node returns *success*. If the number of failing children is greater than $F$, the node returns *failure*. Otherwise, the node returns *running*. $S, F \in \mathbb{N}$ are parameters of the node.

**Decorator ($\delta$)** The Decorator node have only one child. When enabled, it executes a function set as parameter and return the value given by the function. Depending on the function, the child can be ticked or not.

Modularity is one of the main advantages of using behaviour trees. The reuse of sub-trees, even in a same tree instance, is a common use of this modularity. To implement such cases, some behaviour trees framework allows nodes to have multiple parents [29]. This usage is however discouraged by Marzinotto et al. [39], who rather advice to encapsulate such sub-trees in dedicated actions node. By doing this, the human tree readability is increased, without loss of generality. Consequently, in this work, nodes will be allowed to have one parent at maximum.

## 2.2 Advantages

Behaviour trees were initially developed as tools for creating artificial characters in video games [10, 40], replacing finite states machines that were previously widely used. The main advantage of behaviour trees is their inherent modularity: tasks and subtrees can be easily reused in multiple places, as they not need knowledge about the nodes higher up in the tree.

The modularity gain that comes from the behaviour trees can be explained from their *two-way control transfers* that they implement [11]. The different levels of a behaviour tree act like the function call stack of a programming language: at some point, the program can delegate some work by calling a *function* (respectively a *node* in the context of behaviour trees) which will provide some feedback about the executed task in the form of a return value. In the same way a function can be called in different places in a computer program, a node or a subtree in a behaviour tree can be reused. Functions or nodes are independent of their calling context; it is not their task to know what has to be done after their work.

This is fundamentally different of what finite state machines do. Finite state machines implements a *one-way control transfers*: when a transition is activated and the machine switch to a new state, it becomes the responsibility of this new state to decide what have to be done next. It becomes way more difficult to add, removes or reuse part of the state machine, as it will break the execution path. To continue with the programming language comparison, finite state machine transitions are comparable to Goto statements, which are known to be a bad programming habit [13].

## 2.3 Use in robotics

In consequence of their development in the industry, behaviour trees started to be used and studied by academics in robotics and AI [10]. As examples, Bagnell et al. [1] developed their robotic platform around behaviour trees. Bojic et al. [8] extended the JADE (Java Agent DEvelopment) framework with behaviour trees to overcome difficulties with large finite state machines. Ogren [46] argue that behaviour trees may be used to improve modularity and complexity in the context of Unmanned Aerial Vehicle (UAV).

More recently, behaviour trees have been studied to be used in coordination with automatic design. As example, Nicolau et al. [43] automatically assembled behaviour trees into agents that navigate in platform games using Grammatical Evolution (GE). More specifically, the use of behaviour trees and automatic design methods to create control software in the context of *robots swarms* have been studied by Jones et al. [30], Scheper et al. [51], Jones et al. [31], Neupane and Goodrich [42] and as an AutoMoDe flavour by Kuckling et al. [35].

# Chapter 3

# AutoMoDe

AutoMoDe [20] is an approach to the automatic off-line design of control software for robots swarm developed at IRIDIA. The aim of AutoMoDe is to provide software that are resistant to the reality gap (section 1.3).

This chapter details the flavours of AutoMoDe on which this work is based. `Vanilla` is the first flavour that has been published, and it describes the basis of the approach. `Chocolate` introduced a new racing algorithm, that is the one that will be used in this work. `Maple` introduced Behaviour Trees, the control software structure which will be deepened in this work.

## 3.1 Main concepts

This section introduces the basis of AutoMoDe, and how the approach can be applied to a specific robotic platform.

AutoMoDe is a design approach, that describes how bias can be injected in an automatic design process. This bias takes the form of predefined modules, in the form of *behaviours* and *conditions*. Behaviours are basic and unitary tasks that the robots can execute, and conditions are probabilistic tests on the sensory input of the robots. Both behaviours and conditions have parameters that can be fine tuned by the optimisation process, allowing AutoMoDe to adapt the modules to the particular task for which they are required.

The set of behaviours and conditions depends of the capabilities of the robots that are considered for the current process. Specializing AutoMoDe for a robotic platform begins by creating the *reference model* of the platform, that defines an interface between the hardware and the control software. Essentially, the reference model gives a list of variables that can be read and/or written by the software, along with an update period. The reference model implicitly defines the set of possible tasks than can be executed by the robots. The list of modules should then be created on the basis of this reference model. It has to be noted that the reference model and the list of modules depend on the robotic platform, but are independent of the particular tasks that has to be achieved.

AutoMoDe flavours also define a control software structure, in which behaviours and conditions modules are assembled. The commonly used structure is the Probabilistic Finite State Machine (PFSM). In this work, we will study the use of Behaviour Trees.

The last thing needed to make a flavour an automatic design method is the optimisation algorithm, that is used to assemble the module into control software designed for a particular task. In the following AutoMoDe flavours, the optimisation algorithm that are used are racing algorithms. This not imposed by the guidelines of the AutoMoDe approach, and other flavour used different algorithms. For example, AutoMoDe-`IcePop` [34] study the efficiency of a Simulated Annealing algorithm.

To achieve such software creation, the algorithms need a measure of the efficiency of the robot swarm against the particular task for which they are designed. In AutoMoDe, this is referred as the *objective function*. This objective function, coupled with the description of the environment, initial position of the robots, etc. is referred as the *mission*. Missions are not linked with a particular flavour and exist independently of them, but are often designed in a way to highlight and assess particularities. For example, AutoMoDe-`Gianduja` [26] introduced communication based behaviour and conditions, and defined new missions that promotes such communication to assess its new modules.

## 3.2 AutoMoDe-`Vanilla`

Francesca et al. [20] introduced AutoMoDe-`Vanilla` as the first flavour of AutoMoDe. In their paper, the authors defined several notions and concepts that were reused in the following versions of AutoMoDe, including this work.

`Vanilla` produces control software in the form of probabilistic finite state machines. Finite state machines are composed of states and transitions. States are chosen among the list of behaviours and transitions among a list of conditions. Behaviour are executed endlessly until a transition is triggered and that a new behaviour is selected, and conditions are used as trigger for transitions to react to particular events.

Francesca et al. designed `Vanilla` for the e-puck robots [41]. They are two wheeled robots, designed for research and educations, that are extended with ground sensors and range-and-bearing board for `Vanilla`. Their reference model is shown in Table 3.1. The control software can read the variables $prox_i$, $light_i$, $gnd_i$, $n$, $r_m$ and $\angle b_m$ and writes the variables $v_l$ and $v_r$. The range-and-bearing

**Table 3.1** – E-puck reference model used in `Vanilla` and `Chocolate`.

| Sensors/Actuators | Variables |
|---|---|
| Proximity | $prox_i \in [0,1]$, $\angle q_i$, with $i \in \{1,2,...,8\}$ |
| Light | $light_i \in [0,1]$, $\angle q_i$, with $i \in \{1,2,...,8\}$ |
| Ground | $gnd_i \in \{0, 0.5, 1\}$, with $i \in \{1,2,3\}$ |
| Range-and-bearing | $n \in \mathbb{N}$ and $r_m$, $\angle b_m$, with $m \in \{1,2,...,n\}$ |
| Wheels | $v_l, v_r \in [-v, v]$, with $v = 0.16m/s$ |
| Update period: $100ms$ | |

module can reliably exchange messages at a distance of $0.7m$.

The variables are defined in the following ways: $prox_i$ is the reading of the $i$-th proximity sensor and $\angle q_i$ is the angle of the $i$-th sensor with respect to the head of the robot; the proximity reading equals 0 when no obstacle is perceived in a range of $0.03m$ and equals 1 when an obstacle is closer than $0.01m$. $light_i$ is the reading of the $i$-th light sensor and $\angle q_i$ is the angle of the $i$-th sensor with respect to the head of the robot; the light reading equals 0 when the sensor perceives only ambient light and equals 1 when the sensor saturates. $gnd_i$ is the reading of the $i$-th ground sensor; it is equals to 0, 0.5 or 1 when the sensor perceive respectively a black, grey or white ground. $n$ is the number of robots perceived by the range-and-bearing module in the neighbourhood; $r_m$ and $\angle b_m$ are respectively the range and bearing of the $m$-th neighbour. $v_r$ and $v_l$ are the speed of the right and left wheels of the robots. The readings are updated with a period of $100ms$.

Based on this model, 6 behaviours and 6 conditions have been created. Each behaviour is associated to a state, and is executed as long as no outgoing transition is triggered. At each control cycle, all conditions of outgoing transitions are evaluated. If one or more transitions are enabled, one of them is randomly selected and the current state is updated accordingly.

## Behaviours

**Exploration** The robot moves straight, until a proximity sensor placed at the front of the robot perceives an obstacle, i.e. $prox_i > 0.1\ \forall i \in \{1,2,7,8\}$. When it happens, the robots turns on itself for a random number of control cycles in $\{0,1,...,\tau\}$ where $\tau$ is an integer in $[1,100]$. The robots turns in the opposite direction of $\angle q_i$ where $i = \max_{i \in \{1,2,7,8\}} prox_i$.

**Stop** The robot does not move.

**Phototaxis** The robot moves toward the light source, in the direction of vector $w = w' - kw_0$, which is computed as a function of the light attraction vector

$w'$ defined as:

$$w' = \begin{cases} w_l = \sum_{i=1}^{8}(light_i, \angle q_i) & \text{if light is perceived,} \\ (1, \angle 0) & \text{otherwise.} \end{cases}$$

and the obstacle avoidance component $kw_0$, where $k$ is a constant fixed to 5 and $w_0$ is defined as follows:

$$w_0 = \sum_{i=1}^{8}(prox_i, \angle q_i)$$

**Antiphototaxis** The robot moves away from the light source. The behaviour is defined in the same way as Phototaxis but using $w'$ defined as a light repulsion vector:

$$w' = \begin{cases} -w_l & \text{if light is perceived,} \\ (1, \angle 0) & \text{otherwise.} \end{cases}$$

where $w_l$ is defined in Phototaxis behaviour.

**Attraction** The robot use range-and-bearing to go in the directions of the robots in the neighbourhood. Obstacle avoidance is embedded as in Phototaxis, the robot moves in the direction $w = w' + kw_0$ where $w'$ is computed as follows:

$$w' = \begin{cases} w_{r\&b} = \sum_{m=1}^{n}(\frac{\alpha}{r_m}, \angle b_m) & \text{if robots are perceived,} \\ (1, \angle 0) & \text{otherwise.} \end{cases}$$

where $\alpha$ is real parameter in $[1, 5]$.

**Repulsion** The robot moves away from the other robots in the neighbourhood. The behaviour is defined in the same way as Attraction but using $w'$ defined as follows:

$$w' = \begin{cases} -w_{r\&b} & \text{if robots are perceived,} \\ (1, \angle 0) & \text{otherwise.} \end{cases}$$

where $w_{r\&b}$ is defined in Attraction behaviour.

## Conditions

**Black-floor** When all grounds sensors detect a black floor, the transition is enabled with probability $\beta$.

**Gray-floor** When all grounds sensors detect a gray floor, the transition is enabled with probability $\beta$.

**White-floor** When all grounds sensors detect a white floor, the transition is enabled with probability $\beta$.

**Neighbor-count** Transition is enabled with probability

$$z(n) = \frac{1}{1 + e^{\eta(\xi - n)}}$$

where $n$ is the number of robots in the neighborhood, $\eta \in [0, 20]$ and $\xi \in \{0, 1, ..., 10\}$ are parameters.

**Inverted-neighbor-count** Same as Neighbor-count but with probability $1 - z(n)$.

**Fixed probability** Transition is enabled with probability $\beta$, where $\beta$ is a parameter.

Behaviours and conditions are tuned and assembled into a finite state machine by the F-Race algorithm [4]. F-Race sequentially evaluates candidate configurations (here, finite state machines) in simulation using an objective function that is dependant of the current task for which the software is designed for. F-Race progressively discards the configurations that are statistically weaker than the others. Each run is started with a different random seed, e.g. a different position distribution of the robots on the experiment space. The candidate configurations are created by randomly selecting states, transitions and the values of the parameters of each module. To restrain the configuration space and keep up with the idea of biased software, the generated finite states machines were allowed to have at most four states and four transitions.

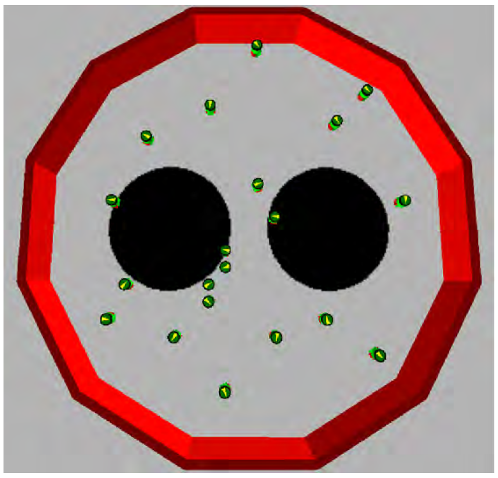
The quality of generated control software by AutoMoDe-`Vanilla` have been evaluated on two missions: Aggregation and Foraging. Both missions are carried out by a swarm of 20 robots.

In the Aggregation mission, the arena is composed of a grey floor and two black circular areas (Figure 3.1). Robots are initially randomly distributed and have to aggregate on only of the two areas. The objective function is

$$F_{aggregation} = \frac{\max(N_a, N_b)}{N}$$

where $N = 20$ is the swarm size, and $N_a$ and $N_b$ are the number of robots on the two black areas at the end of the experiment.

In the Foraging mission, the arena is composed of food sources and a nest (Figure 3.2). Robots must move a maximal number of objects from the sources to the nest. As e-pucks do not have grabbing capabilities, robots are considered


**(a)** Simulation

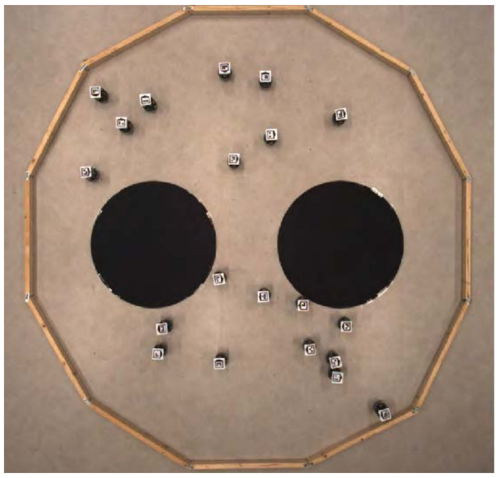
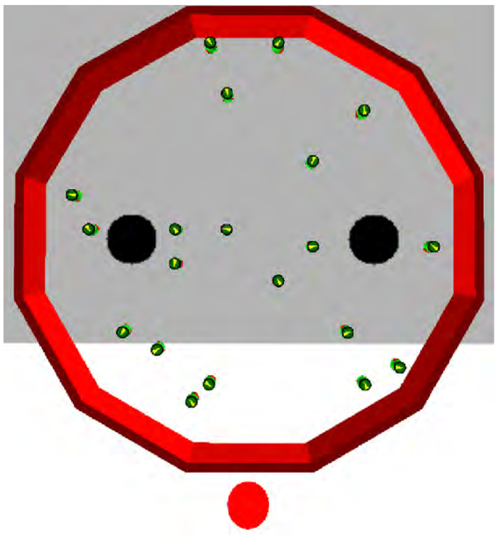

**(b)** Real arena

**Figure 3.1** – Aggregation arenas[20]. Both of these arenas are dodecagonal arenas of $4.91m^2$, centred at the coordinates (0;0). The ground is grey except for two black circular areas that have a radius of $0.35m$ and that are centred in $(0.6;0)m$ and $(-0.6;0)m$.


**(a)** Simulation

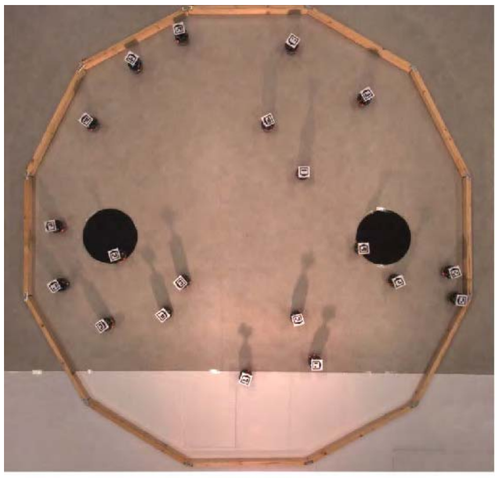

**(b)** Real arena

**Figure 3.2** – Foraging arenas[20]. Both of these arenas are dodecagonal grey arenas of $4.91m^2$, centred at the coordinates (0;0). The sources are two circular black areas with a radius of $0.15m$ and are centred in $(0.75;0)m$ and $(-0.75;0)m$. A light source is positioned behind the nest, at $0.75m$ over the coordinate $(0;1.25)m$.

having picked an object when entering in the source and having dropped their object when entering in the nest. The objective function is

$$F_{foraging} = N_o$$

where $N_o$ is the number of objects deposited in the nest.

Francesca et al. used `Vanilla` to create control software for these two missions using `irace` R package [38, 54] for the optimization algorithm and ARGoS [47] for the simulator. The results were compared to the ones generated by evolutionary robotics method that the authors called `Evostick`, a method that they already successfully used in previous experiments [18]. In `Evostick`, the control software is created in the form of a two layered feed-forward neural network with a constant structure defined based on the reference model. The optimisation algorithm then learns the weight to associate with each connection.

It has to be noted that AutoMoDe-`Vanilla` and `Evostick` are *automatic* design methods, meaning that they create control software without human intervention, apart from the initial conception of modules and the neural network structure. In particular, the modules, the neural network or the design process should not be modified because of mission particularities or previous experiments results analysis.

Both methods were assessed using the same missions definition and criteria, and using multiple optimisation budgets. Results are in the form of plots in the original paper by Francesca et al. [20]. Comparison between `Vanilla` and `Evostick` show that `Vanilla` fulfils its objective against the reality gap problem. The robots trained with `Vanilla` perform similarly in simulation and in real world experiments. In simulation, their performance may be lower than `Evostick`–it was especially the case in Foraging mission. However, `Vanilla` performs significantly better than `Evostick` in real world experiments.

## 3.3 AutoMoDe-`Chocolate`

After testing `Vanilla` against `Evostick`, Francesca et al. [19] extended the experiment by testing both methods on new missions, and against human designed control software. They kept the same robotic platform, reference model and set of modules, but added two human design methods called `U-Human` and `C-Human`. The performance of the four methods were evaluated against five new missions.

In `U-Human`, the designer is free to implement the software the way he wants, without any structural restrictions. As in `Vanilla` and `Evostick`, the designer can use a simulator to assess the performance of its control software and iteratively improve it by a trial-and-error process. In the experiments described by Francesca et al., the designer can have all resources he needs at his disposal, including previously developed code.

In `C-Human`, the designer is constrained to use the same software representation as in `Vanilla`, meaning that he must create a finite state machine with at most four states and four transitions, using the same set of modules. The designer replaces the optimisation algorithm that were used in `Vanilla`. As in `U-Human`, the designer iteratively improve its solution by a trial-and-error process.

The four design methods were tested using five new missions, designed to be achievable by robots that conforms to the e-puck reference model (Table 3.1). All these missions take place in the same dodecagonal arena used for the Aggregation and Foraging missions described before. The arena definition and the objective function of these missions can be found in the original paper written by Francesca et al. [19].

The results obtained after the experiments lead to multiple observations. Firstly, as expected from the results obtained with Aggregation and Foraging missions, `Vanilla` performs significantly better than `Evostick` in real world conditions. Secondly, `U-Human` seems to suffer from the reality gap similarly as `Evostick`, and so was outperformed in reality by `Vanilla`. Thirdly, `C-Human` showed to be resistant to the reality gap while significantly outperforming `Vanilla` and thus the two other methods.

The comparison between `U-Human` and `C-Human` confirms that the bias/variance trade-off approach taken by Francesca et al. [20] can solve the reality gap problem. On the other hand, it shows that `Vanilla`, while having a set of module appropriated for its reference model, is not able to produce as good results as an human designer. This highlight the fact that the optimisation algorithm adopted by `Vanilla` is not able to fully exploit the modules potential.

Francesca et al. [19] created a new version of AutoMoDe called `Chocolate`, which shares reference model, set of modules, finite state machine structure with `Vanilla`. The difference lies in the optimisation algorithm that is used to build the control software. `Vanilla` uses F-Race [4], `Chocolate` uses Iterated F-Race [6]. Iterated F-Race consists of sequential runs of F-Race, where the best candidate configurations of each iteration serve as seed to generate the candidates of the next iteration. In the first iteration, candidates are generated randomly and uniformly from the feasible candidates space as in classical F-Race. In the following iterations, the candidates are sampled following a probability distribution that depends on the remaining candidates of the previous generation in a way to favours candidates that are close to them. `Chocolate` adopts the Iterated F-Race algorithm that is proposed by López-Ibáñez et al. [38] in the `irace` R package.

Francesca et al. ran `Chocolate` on the five missions that were used previously and compared the scores of the generated control software against the performance of `Vanilla` and `C-Human` obtained before. As expected, the results show that `Chocolate` significantly outperforms `Vanilla`, but also `C-Human`. This show that, on these five missions, the Iterated F-Race is a sufficiently advanced optimisation algorithm to outperforms human designers. As previous results show that `C-Human` performs better than `U-Human`, we can extend the conclusion to the fact that `Chocolate` outperforms humans whether they are constrained to modules or not.

## 3.4 AutoMoDe-`Maple`

Both `Vanilla` and `Chocolate` assembled their modules into probabilistic finite state machines. With AutoMoDe-`Maple`, Kuckling et al. [35] wanted to assess behaviour trees as control software structure for swarm robotics.

`Maple` works in a similar way as `Chocolate` and `Vanilla`. Given a set of modules, a mission and a simulation environment, an optimisation algorithm tune and assemble selected modules into a control software. To assess properly behaviour trees against finite state machines, `Maple` use the exact same reference model, set of modules and optimisation algorithm as `Chocolate`.

However, behaviour trees are inherently different from finite state machine, making `Vanilla` modules not directly suitable for use in behaviour tree. In particular, states are meant to be run *indefinitely*, until a transition is triggered and cause a state switch. Translated into a behaviour tree action, it will mean that this action will always return *running*. This can be a problem, because a Sequence or a Selector control node will stop to tick its children when it sees a running child, meaning that any subtree placed after an action will never be executed.

To overcome this problem that could happen with the `Vanilla` modules, `Maple` restrict the structure of the tree to the one depicted at Figure 3.3. The root node is a Sequence* node and have Selector nodes as children. Each Selector node has exactly two children: one condition node and one action node, in that order. To match with the `Vanilla` and `Chocolate` restriction to have maximum four states, behaviour trees in `Maple` cannot have more than four Selector sub-trees.

In such trees, an action will be ticked as long as its associated condition return *failure*. When the condition return *success*, the Selector node will stop and return *success*. The Sequence* then tick the next Selector sub-tree, until its condition returns *success*.

Kuckling et al. compared `Maple` against `Chocolate` and `Evostick` on the two missions initially developed for `Vanilla`: Foraging and Aggregation. Results showed that both `Chocolate` and `Maple` outperforms `Evostick`, which suffers from the reality gap. Control software generated by `Maple` and `Chocolate` reach equivalent performances, by using similar strategies.

Although `Maple` provides good results, it obviously restricts the behaviour trees potential, and this both because of the modules and because of the imposed structure. In this work, a new set of modules will be designed to benefit as much as

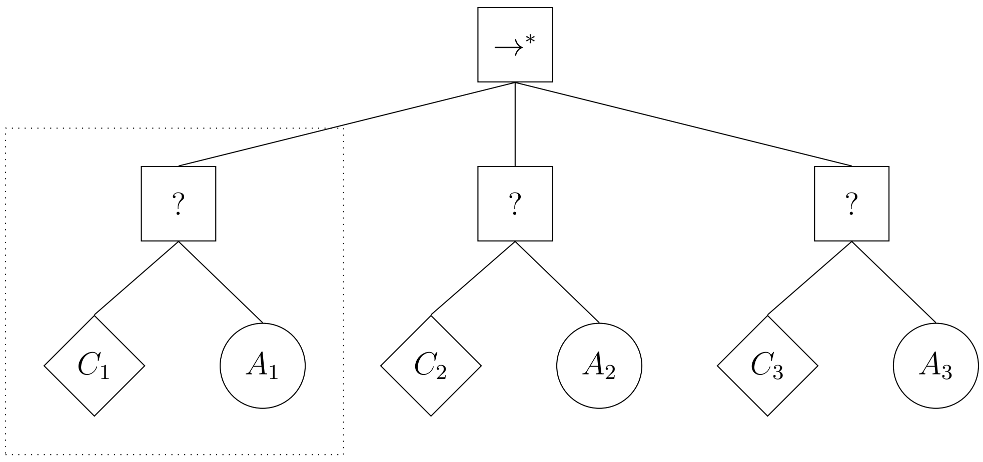


**Figure 3.3** – Behaviour tree structure used in `Maple`. Control nodes, condition nodes and action nodes are respectively represented by squares, diamonds and circles. The first Selector sub-tree is highlighted using a dotted box. In this example, the tree contains three Selector sub-trees, but `Maple` allow it to contain up to four.

possible from the behaviour trees return values. With actions that stops at some point by no longer running indefinitely, a more flexible structure for behaviour trees will be adopted.

# Chapter 4

# Related Work

Behaviour trees were only recently used in the context of automatic design of robot swarms. This chapter gives an overview of the state of the art studies in this field, compares them with AutoMoDe-`Maple` and explains the defaults they share with it.

The use of behaviour trees and automatic design methods to create control software in the context of *robots swarms* have been studied in multiple papers [30, 51, 31, 42], including AutoMoDe-`Maple` [35]. However and as specified before, behaviour trees implement *two-way control transfers*, and these studies do not make full use of it. In particular, we can see that the defined action nodes in most of these papers do not fully exploit the range of return values, that include *success*, *failure* and *running*. On another perspective, the results obtained in some of these papers also illustrate well the effects of the reality gap.

Jones et al. [30] used behaviour trees as control software structure to design, through genetic programming, a swarm of robots performing a foraging task. This paper, in many aspects, is similar to what is done with `Maple`. They are both off-line automatic design methods, evaluated on a related mission (Foraging). The robots used have similar capabilities; in particular, the only actuator they have is their locomotion system who allows them to move on the experiment arena. The set actions proposed by Jones et al. [30] include movements–straight and left/right turn–of 1 tick duration for each of them. Such actions return *running* at the first tick, and *success* at the second one. It also have to be noted that theses actions, when used in experimental setup, show an significant effect of the reality gap on the produced control software. Following the conclusions of AutoMoDe and as explained by the authors themselves, it may highlight the fact that such actions should be encapsulated in higher level behaviours.

Scheper et al. [51] designed an evolutionary robotics process to create behaviour trees for the DelFly Explorer, a flapping wing micro air vehicle, making them explore a square room and searching for an open window to escape. In their work, the behaviour tree interact with the sensors through a blackboard, that give access to the variables of the robot reference model, with read or write access depending of the variable. In such situations, actions are reduced to variable assignments, and hence can only succeed, meaning that they always return the same value. It can also be noted that they allow their control software to directly set the value of the actuators, which is comparable to the direct assignment of outputs from the neural network to the actuators in `Evostick`, the evolutionary method that were used as control method in the first versions of AutoMoDe. Like `Evostick`, Scheper et al. [51] show a significant impact of the reality gap on their results.

Jones et al. [31] used behaviour trees to implement an *on-line* automatic design. They obtained promising results by showing that a fully evolved control software can be obtained in 15 minutes. The actions in the set they used work in similar way than the actions Scheper et al. [51] used, i.e. they are actions that directly set variables of the robot reference model, which reduce the power of the *two-way control transfers* of behaviour trees. However, as they do an *on-line* automatic design, remarks about reality gap does not apply here.

Neupane and Goodrich [42] defined a grammar based on behaviour trees to represent the control software of their robots. They used it to create software for foraging, cooperative transport and nest maintenance missions. Although they do not detail the internal functioning of their actions–and thus the possible return values–, they seem to propose higher level behaviours.

Most of the studies presented use low level actions, where it is difficult to propose informative return values to the parent nodes. It seems though that such values are important to make the behaviour trees work at full power: without proper return value from actions, the control nodes cannot react and change the execution flow in the tree. A way of overcoming this issue is the addition of condition nodes before or after actions. This is however not an ideal solution, since it increase the complexity of the tree and do not fill the absence of *running* values.

In addition to this and according to the result obtained by the authors, such low level actions also comes with effects due to the reality gap. We can expect that introducing high-level behaviour could solve multiple problems at once, giving more expressiveness to the actions, reducing the size and complexity of the tree, and managing the effects of the reality gap.

# Chapter 5

# AutoMoDe-`Cedrata`

In this chapter, a new flavour of AutoMoDe, called `Cedrata`, will be defined. This new flavour essentially introduce a new set of modules for the e-puck robots [41]. This modules redesign include the introduction of a new reference model, evolution of some behaviours from `Vanilla` and the introduction of new ones. This set is meant to be assemble into a behaviour tree, which will have different constraints that the ones proposed for AutoMoDe-`Maple`.

## 5.1 The new reference model

The design of new modules is motivated by the lack of state information given by ones of previous AutoMoDe versions (see section 3.4). Based on the three return values that exist in behaviour trees–*success*, *failure* and *running*–, we easily imagine action behaviours that represent unitary tasks that are meant to end at some point or eventually fail.

Behaviours proposed in `Vanilla` are designed to be used in finite state machine, i.e. to run indefinitely until a transition is triggered. Some of them can be extended to include *success* and *failure* conditions, and it has been done (see section 5.2). However, most of them cannot be translated so easily.

The solution that was adopted was to use the range-and-bearing actuators of the e-puck robots to extend the capabilities of the robots compared to `Vanilla`, and thus allowing the creation of new behaviours. The new reference model is shown in figure 5.1. It introduces signals, exchanged between robots by the range-and-bearing with every message. A robot always send a signal value $s_m$, that can be equal to 0, which is a special value that means *no signal* and that is sent by default, or a integer in $\{1, ..., 6\}$. Signal values does not have particular meaning, it is the role of the designer to give them one depending on the mission.

This reference model allow direct communication between the robots. It is

| Sensors/Actuators | Variables |
|---|---|
| Proximity | $prox_i \in [0,1]$, $\angle q_i$, with $i \in \{1,2,...,8\}$ |
| Light | $light_i \in [0,1]$, $\angle q_i$, with $i \in \{1,2,...,8\}$ |
| Ground | $gnd_i \in \{0, 0.5, 1\}$, with $i \in \{1,2,3\}$ |
| Range-and-bearing | $n \in \mathbb{N}$, $r_m$, $\angle b_m$ and $s_m \in \{0,1,...,6\}$, with $m \in \{1,2,...,n\}$ |
| Wheels | $v_l, v_r \in [-v, v]$, with $v = 0.16 m/s$ |
| Update period: $100ms$ | |

**Table 5.1** – The E-puck reference model used by `Cedrata`. This is the same as the one used in `Vanilla` (see Table 3.1) with the addition of a signal $s_m$ exchanged using the range-and-bearing actuator.

commonly known that computer communication is error-prone: messages can be lost, answers may be unexpected, etc. On the other hand, a exchange between two communicants reach a successful end when all the required messages have been exchanged. The addition of communication in the form of signals allows the creation of a variety of behaviours exploiting the *success* and *failure* return values.

The communication system have been kept simple: robots have only 7 different messages that they can exchange (when including the 0 signal). This choice have been made for two reasons: (i) to stick with AutoMoDe philosophy of bias introduction by having only high level behaviours at the disposition of the optimisation algorithm, (ii) because a large message space will greatly increase the possible control software space and will make the task of the optimisation process harder. The second one is especially important when the optimisation process need to find concordant signals between two or more signal based behaviour that are present in the tree: all discordant signal combinations will lead to inefficient control software.

## 5.2 Modules

This section lists and defines the set of modules that will be used. In the following descriptions of the signal based conditions and behaviours, the set of signals $\{1, ..., 6\}$ will be denoted $S$. Some behaviours can use a special value *any* that is activated if any of the signals in $S$ is received. The set $S^* = S \cup \{any\}$ will denote the sets used by these behaviours.

### Conditions

The new set of conditions is shown below. Conditions are kept unchanged from `Vanilla`, with the addition of the Receiving Signal condition.

**Black-floor** When all grounds sensors detect a black floor, the transition is enabled with probability $\beta$.

**Gray-floor** When all grounds sensors detect a gray floor, the transition is enabled with probability $\beta$.

**White-floor** When all grounds sensors detect a white floor, the transition is enabled with probability $\beta$.

**Neighbour-count** Transition is enabled with probability

$$z(n) = \frac{1}{1 + e^{\eta(\xi - n)}}$$

where $n$ is the number of robots in the neighbourhood, $\eta \in [0, 20]$ and $\xi \in \{0, 1, ..., 10\}$ are parameters.

**Inverted-neighbour-count** Same as Neighbour-count but with probability $1 - z(n)$.

**Fixed probability** Transition is enabled with probability $\beta$, where $\beta$ is a parameter.

**Receiving signal** Returns *success* if the robot has perceived a neighbour sending $s \in S^*$ in the last 10 ticks. Returns *failure* otherwise.

## Behaviours

The new set of behaviours is shown below. The Exploration behaviour is kept unchanged from `Vanilla`. Exploration have been previously used in a lot of scenarios, and it has been considered too basic to be removed from the set of actions. The old Attraction and Repulsion behaviours have been upgraded into the new Grouping and Isolation behaviours, that add thresholds on the number of neighbours as conditions of *success* and *failure*. The new Meeting, Acknowledgment and Emit Signal behaviours have been added to let the robot use the new signal framework. The old Phototaxis and Antiphototaxis behaviours have been dropped, in order to keep a small set of available modules for the optimisation process.

**Exploration** is a random walk strategy. The robot moves straight until it perceives an obstacle in front of itself. Then the robot turns on the spot for a random number of tick in $\{0, ..., \tau\}$, where $\tau \in \{1, ..., 100\}$ is a parameter. This behaviour always return *running*.

**Stop** orders the robot to stay still. This behaviour always return *running*.

**Grouping** The robot tries to get closer from its neighbours by moving in the direction of the geometric centre of its neighbours. If the number of neighbours becomes greater than $N_{max}$, the behaviour *succeeds*. If the number of neighbours becomes smaller than $N_{min}$, the behaviour fails. Otherwise, it returns running. As in Attraction, the speed of convergence is controlled by the parameter $\alpha \in [1, 5]$.

The Grouping behaviour has embedded collision avoidance as in `Vanilla` behaviours: the robot moves in the direction $w = w' - kw_0$, where $w'$ is computed as follows:

$$w' = \begin{cases} w_{r\&b} = \sum_{m=1}^{n}(\frac{\alpha}{r_m}, \angle b_m) & \text{if robots are perceived,} \\ (1, \angle 0) & \text{otherwise.} \end{cases}$$

and where $kw_0$ is the obstacle avoidance component, where $k$ is a constant fixed to 5 and $w_0$ is defined as follows:

$$w_0 = \sum_{i=1}^{8}(prox_i, \angle q_i)$$

**Isolation** The robot tries to move away from its neighbours by moving in the opposite direction of the geometric centre of its neighbours. If the number of neighbours becomes smaller than $N_{min}$, the behaviour succeeds. If the number of neighbours becomes greater than $N_{max}$, the behaviour fails. Otherwise, it returns *running*. As in Repulsion, the speed of divergence is controlled by the parameter $\alpha \in [1, 5]$.

The Isolation behaviour use the same embedded collision avoidance than in Grouping, but with $w'$ defined as follows:

$$w' = \begin{cases} -w_{r\&b} & \text{if robots are perceived,} \\ (1, \angle 0) & \text{otherwise.} \end{cases}$$

where $w_{r\&b}$ is defined in Grouping behaviour.

**Meeting** The robot listens for a signal $s \in S^*$ emitted by another robot(s) and moves forward towards the geometrical centre of the emitters. The behaviour returns *success* if the distance between the robot and the geometrical centre is smaller than a distance $d_{min}$. The behaviours fails if the robot did not have perceived a robot that send the expected signal. Otherwise, the behaviour returns *running*.

The Meeting behaviour use the same embedded collision avoidance than in Grouping, but with $w'$ defined as follows:

$$w' = \begin{cases} w_{r\&b} = \sum_{m \in S_r^*}(\frac{\alpha}{r_m}, \angle b_m) & \text{if robots are perceived,} \\ (1, \angle 0) & \text{otherwise.} \end{cases}$$

where $S_r^*$ is the set of robots that have a signal that match with $s$.

**Acknowledgment** The robot send a signal $s \in S$ and waits for a answer in the form of the same signal. The behaviour returns *success* if the signal is received or *running* if not. After $t_{max}$ ticks, the behaviour returns *failure* if the signal is still not received. This behaviour also sets the wheels velocity to zero.

**Emit Signal** Sets the emitted signal to $s \in S \cup \{0\}$ for the current tick. This behaviour always returns *success.* This behaviour also sets the wheels velocity to zero.

The implementation of the reference model that is used here keeps received messages up to 10 ticks in memory, if a newer message by the same robot is not received before. It means that if the description of a behaviour say that the robot must perceive a signal $s$, this signal could have been send 10 ticks ago.

## 5.3 Behaviour tree structure

In `Maple`, with *running* being the only value that behaviours can return, the tree have been constrained to a particular structure to avoid problems with the execution flow. With the new set of behaviour, the problem is greatly reduced, although not completely removed: the Exploration and Stop behaviours are still in the set. However, it means that the behaviour tree can reasonably be assembled in a more flexible way. Allowing the design process to change the structure and choose the control nodes type moves toward a better exploitation of the two-way control transfer.

In `Cedrata`, the optimisation process can create a tree that have maximum two levels and maximum three children per node. Unlike `Maple`, all branches are not forced to have a depth of two: the root can have children that are action or condition nodes. The optimisation process can choose any control node type from Sequence, Sequence*, Selector or Selector*. To match with the `Maple` restriction of four behaviours and four conditions, the tree is allowed to have at most four action nodes and four condition nodes. The constraints on the depth and on the number of children implicitly impose that the tree contains no more than four control nodes. The structure of such trees is depicted in figure 5.1.

In opposition of `Vanilla` or `Maple` where a behaviour has to be executed at each tick of the the control software, the tree structure adopted in `Cedrata` allows the tree to stop after ticking only conditions, depending of the chosen control nodes. It means that some actuators may not be updated properly. It can be dangerous in the case of the wheels velocity, because the robot may keep moving

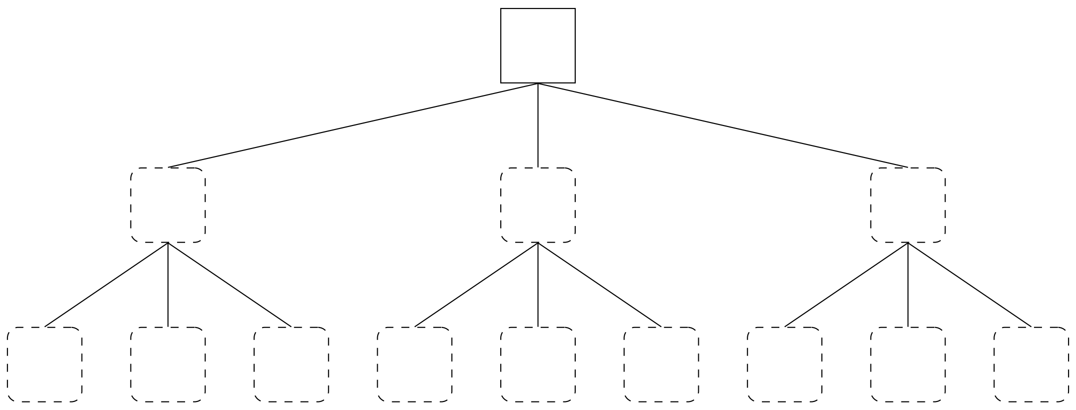


**Figure 5.1** – The behaviour tree structure used with the new modules. Control nodes, condition nodes and action nodes are respectively represented by squares, diamonds and circles. Nodes that can be control, actions or conditions are showed using dashed boxes. A node of the second level can be an action or condition node, in this case it have no children. It has to be noted that the tree is allowed to have maximum four conditions and four actions, making this example impossible in practice due to the nine leaf nodes that it has.

without the control of a behaviour and the embedded collision avoidance. If no behaviour is ticked during the tree execution, the wheels velocity is assumed to be zero. The same goes for the emitted signal, which is set to zero if no behaviour explicitly set it.

# Chapter 6

# Missions

This chapter lists and describes missions that will be used to compare `Maple` with `Cedrata`.

This list include missions that were used in previous AutoMoDe studies along with new ones that are designed to assess the use of the new behaviours that were introduced in the previous chapter. The missions listed in this chapters are either: (i) missions for which `Maple` already show successful results, and for which we are interested in the performance of `Cedrata`; (ii) missions that promotes the uses of the new features introduced with the new modules, like the signal framework. In such missions, `Cedrata` is expected to demonstrate better results than `Maple`.

## 6.1 Foraging

Francesca et al. [20] designed the Foraging missions for `Vanilla`. These missions were reused by Kuckling et al. [35] to assess the performance of `Maple`.

In the Foraging mission, the robots must take objects from sources and carry them to a nest. As the e-puck robots have no grabbing capabilities, they are considered having picked an object when they enter a source, and having dropped off the object when entering the nest.

The Foraging arena is depicted in figure 6.1. The arena contains two black spots that act as sources, and a white area that act as nest. To help robots to navigate in their environment, a light source is placed behind the nest.

Initially, all robots are placed randomly over the whole arena, including sources or the nest. The mission lasts during 250 seconds or equivalently 2500 control steps. The objective function is

$$F_{foraging} = N_o$$

where $N_o$ is the number of objects deposited in the nest.

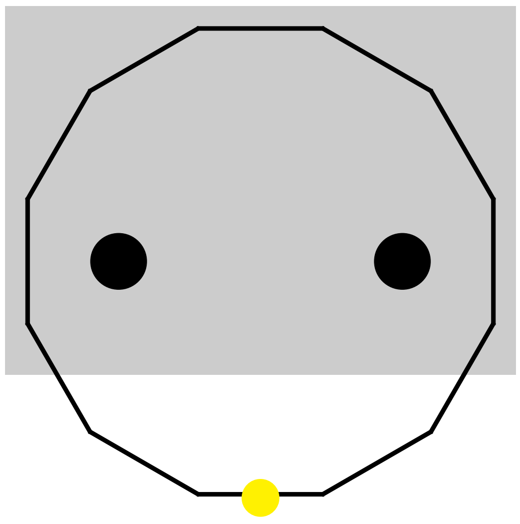


**Figure 6.1** – The Foraging mission arena. It is a dodecagonal arena with an apothem of 1.231 meters centred at the coordinates $(0; 0)$. The ground is grey except for the sources and the nest. The sources are two black circle of radius $0.15m$ centred in $(-0.75; 0)m$ and $(0.75; 0)m$. The nest is a white area that contains all the positions that are under $-0.6m$ on the vertical axis. A light is placed at $0.75m$ over the position $(0, -1.25)m$ and is represented in yellow.

It is worth to remind that the new set of behaviours used by `Cedrata` do not provide access to the light readings, and thus that the robots are unable to exploit the light placed behind the nest.

## 6.2 Marker Aggregation

In the Marker Aggregation mission, the swarm must aggregate on a predefined sub area of the arena. The difficulty comes from the fact the border of this aggregation area is not perceivable by the robots; instead a small black patch called the *marker* is placed at the centre of the aggregation area.

The Marker Aggregation arena is depicted in figure 6.2. The aggregation arena is large enough to contain the whole swarm of 20 robots. The marker size is designed to be able to contain only a small fraction of the robot swarm, but large enough to be easily found by the robots if they randomly explore the space. The mission is designed to promote communication inside the swarm. The robots that are in the aggregation area but not over the marker do not have any sensor that can tell them if they are in the aggregation area or not; they should get this information from their neighbours that have this knowledge, i.e. the robots that are over the marker.

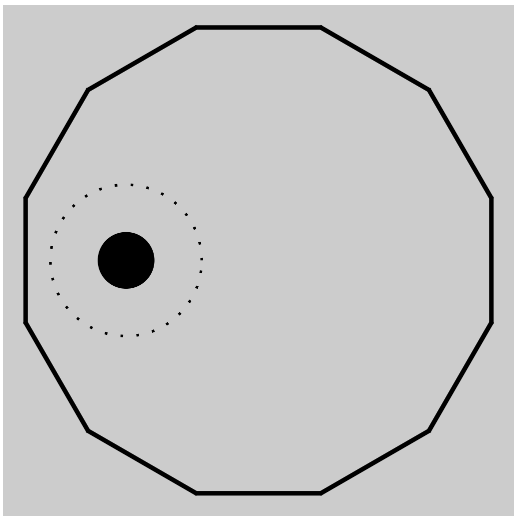


**Figure 6.2** – The Marker Aggregation mission arena. It is a dodecagonal arena with an apothem of 1.231 meters centred at the coordinates $(0; 0)$. The ground is grey except for a black circle of radius $0.15m$ centred in $(-0.7; 0)m$ called the *marker*. The aggregation area is a circle of $0.4m$ centred over the marker and that is not perceivable by the robots.

At the beginning of the mission, all robots are randomly placed in the whole arena. Some of them may be placed in the aggregation arena, or even over the marker. The mission lasts during 250 seconds or equivalently 2500 control steps. The objective function is

$$F_{MarkerAggregation} = \sum_{i=0}^{2500} N_A^i$$

where $N_A^i$ is the number of robots in the aggregation arena at tick $i$.

## 6.3 Stop

The Stop mission has originally been developed by Hasselmann, Robert, and Birattari [26] for AutoMoDe-`Gianduja`, a flavour that introduce communication based behaviour in the set of modules. This mission is designed to promote communication between the robots, making it suited for the new set of modules and the signal framework introduced in this work.

In the Stop mission, the robots must explore the arena to find a white spot and then stop moving. The white spot is considered as being discovered as soon as a robot enters it, and the whole swarm should to stop the quicker as possible after that. That is promoting communication between the robots, since only one of them have the information that stopping is now required.

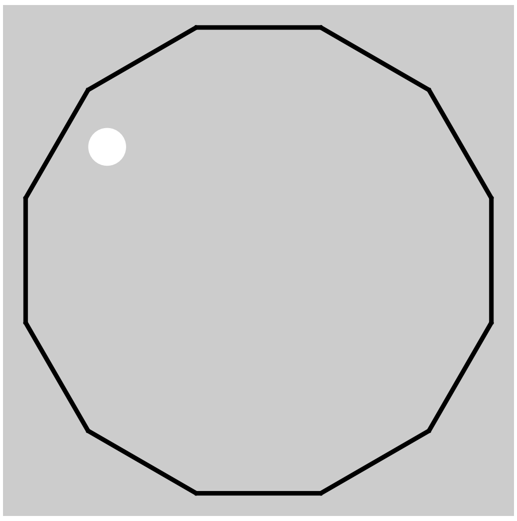


**Figure 6.3** – The Stop mission arena. It is a dodecagonal arena with an apothem of 1.231 meters centred at the coordinates $(0; 0)$. The ground is grey except for a white circle of radius $0.1m$ centred in $(-0.8; 0.6)m$.

The Stop mission arena is depicted in figure 6.3. The arena ground is grey, except for a small white spot placed on the top-left of the arena, near the walls.

Initially, all robots are placed in the right part of the arena, i.e. at positions where the first coordinate is a positive number. The mission lasts during 250 seconds or equivalently 2500 control steps. The original mission was meant to run only for 120 seconds, but have been extended here to keep all considered missions on the same time constraint. The objective function is computed as follows:

$$F_{Stop} = 100000 - \left( \bar{t}N + \sum_{t=1}^{\bar{t}} \sum_{i=1}^{N} \bar{I}_i(t) + \sum_{\bar{t}}^{2500} \sum_{i=1}^{N} I_i(t) \right)$$

where $\bar{t}$ is the tick at which a robot pass over the white spot for the first time, $N = 20$ is the swarm size and $I_i(t)$ and $\bar{I}_i(t)$ are defined as follows:

$$I_i(t) = \begin{cases} 1 & \text{if robot } i \text{ is moving;} \\ 0 & \text{otherwise.} \end{cases} \qquad \bar{I}_i(t) = 1 - I_i(t)$$

A robot is considered a moving if it has travelled more than $5mm$ in the last tick. This objective function encourages the robots to find the spot quickly with the term $\bar{t}N$ and the first summation for $t < \bar{t}$, and promotes that robots stop quickly after the discovery tick $\bar{t}$ with the second summation.

# Chapter 7

# Experimental Setup

In this chapter, a comparison procedure between three alternative design methods will be described. The design methods include `Maple`, `Cedrata` and manual design using the set of modules and the tree structure of `Cedrata`.

Control software will be compared against each other and against the reality gap. This should allow to evaluate (i) the swarm performance evolution from `Maple` to `Cedrata`; (ii) the efficiency of the optimisation algorithm on the new set of modules by comparison to manual designs. In particular, we are interested in the following questions:

1. Do the use of behaviour trees with more adapted modules and a more flexible structure than in `Maple` effectively increase the expressiveness of the control software, under the same restrictions of four conditions and four actions ?

2. Do `Cedrata` present a good resistance against the reality gap, like the others flavours of AutoMoDe ?

3. As seen previously in `Chocolate` [19], the choice of the optimisation algorithm, under identical budget constraints, influence greatly the performance of the generated control software. Is the Iterated F-Race algorithm, that was used for `Chocolate` and `Maple` and reused in this work, adapted for the new set of modules ?

To answers these questions, a series of experiments will be conducted.

## 7.1 Design methods

The experiments described in this chapter will involve comparison between multiple design methods, that include `Maple`, `Cedrata`, and manual design run by experts.

`Maple` is a flavour of AutoMoDe that assembles the modules from `Vanilla` and `Chocolate` into behaviour trees. Using modules that are not designed for behaviour trees, `Maple` restricts the structure of the generated trees. The optimisation algorithm used is Iterated F-Race.

`Cedrata` is a flavour of AutoMoDe that assembles into behaviour trees modules that were designed for them. In particular, they have a greater expressiveness to the behaviour trees return values. Unlike `Maple`, the tree structure is allowed to be more flexible and is designed by the optimisation algorithm. As `Maple`, the optimisation algorithm used is Iterated F-Race.

In the manual design method, an human designer builds the control software of the robot using the `Cedrata` set of modules and assembles them into behaviour trees, under the same constraints than `Cedrata`. The method is similar to the `C-Human` introduced by Francesca et al. [19] in the `Chocolate` study (see section 3.3). In the design process, the designer have access to a visual interface that allow them to build trees and test them in the ARGoS simulator [47]. He have access to the value of the objective function as automatic methods and to a visual representation of the arena and the behaviour of the swarm for inspection. The designers are chosen among people that have knowledge in swarm robotics, but not on this study. In particular, they do not have prior knowledge in behaviour trees.

## 7.2 Protocol

For each mission, design methods will be executed with different budgets: 100,000 and 200,000 simulation runs. For each budget size, 10 runs of the methods are run and lead to 10 control software. The manual design will be done by four human designers for each mission, with a maximum design duration of 4 hours. During the writing process of this work, additional experiments with budgets of 20,000 and 50,000 simulation runs have been performed for the missions Marker Aggregation and Foraging, and they will be included in the results.

To assess the performance of the control software against the reality gap, the best procedure is to make robots execute the mission they were designed for in a real environment. However, the experiments were done and this work was written during the 2020 lockdown due to the COVID-19 pandemic [48]. As result, the experimental setup for real world robot experiments was not accessible. Instead, the impact of reality gap is tested using pseudo-reality.

Ligot and Birattari [37] showed that the effect of the reality gap can be mimicked in simulation-only environments. In particular, they created a new simulation model that reproduce the rank reversal that have been observed with `Vanilla`, `Chocolate` and `Evostick` in real world conditions.

**Table 7.1** – Design and pseudo-reality simulation models

| Sensor/actuator | $M_A$ | $M_B$ |
|---|---|---|
| Proximity | 0.05 | 0.05 |
| Light | 0.05 | 0.90 |
| Ground | 0.05 | 0.05 |
| Range-and-bearing | 0.85 | 0.90 |
| Wheels | 0.05 | 0.15 |

Simulation models are shown in table 7.1. The model $M_A$ is the *design model*, i.e. the model that is used by the optimisation process. The model $M_B$ is the *pseudo-reality* model, i.e. the model that is used to assess the reality gap. The numbers shown in the table are the noise values that are applied to each sensor readings and actuator values. For the proximity, light and ground sensor, a uniform white noise is applied: a random value uniformly taken in $[-p; p]$, where $p$ is the number shown in the table, is added to the sensor reading. For the range-and-bearing sensor, the table number is the probability of message loss. For the wheels actuator, the number is the standard deviation of a Gaussian noise with mean 0 that is added to the velocity value.

## 7.3 Reference Designs

In addition to the behaviour trees created by the manual designs method, a *reference* tree will be added for each mission. These reference designs are not part of the experimental protocol and are designed by people with knowledge of the study. They are built using the newly defined set of behaviours and the same constraints on the tree structure, but without time constraints. These designs will only serve as examples strategies to solves the missions for comparison purposes, and are not in the knowledge of the human designers.

The manual design method already gives, by comparison with the automatic design, a measure of the efficiency of the optimisation algorithm on that new set of modules. The introduction of the reference designs serves additional objectives. Firstly, behaviour trees have initially be developed to ease the design of control software. Following this idea, coming up with efficient control software for each mission should not be a hard task for a human designer. This point should be assessable through comparison between the manual designs (under experimental conditions) and the reference designs. Secondly, the reference designs will serve as examples to show the possibilities of the new modules set.

In the following subsections, we will describe for each mission the behaviour tree that has been designed and the strategy that is behind it.

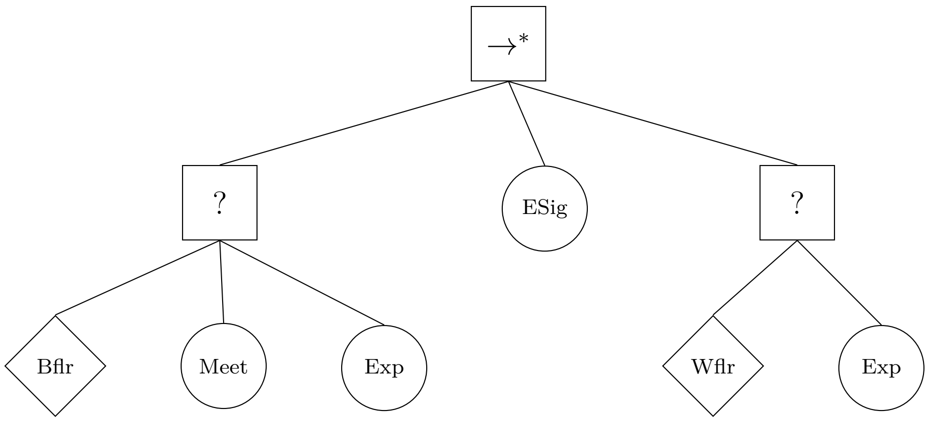


**Figure 7.1** – Hand design for the Foraging mission. The conditions and actions names have been abbreviated in the following way: Bflr: Black-floor; Meet: Meeting; Exp: Exploration; FPrb: Fixed Probability; ESig: Emit Signal; Wflr: White-floor.

## Foraging

The Foraging mission was initially designed with a light over the nest area to guide robots in their search. However, the new set introduced here do not have light detection capabilities, making the obtention of this information beyond the capabilities of the robots, regardless of the fact that a human or an optimisation algorithm is designing the software.

The design that have been implemented is shown in figure 7.1. In this design, robots exploit the signal framework to send indications about the location of the food sources to their neighbours. When a robot find a food source, it emits a one-tick signal. The signal can then be received by robots that are in search of a food source to attract them to it.

The tree is organised in three subtrees. The first one executes until the robot find a food source (black floor). During that search, the robot will be attracted to a signalling neighbour, if any, or explore randomly. The second subtree executes after the discovery of the food source and emit the signal. The third subtree is a random exploration until the nest (white floor) is found.

## Marker Aggregation

In the Marker Aggregation mission, the robot must aggregate on a spot that is set in a way in which all robots cannot have the localisation information. This mission was designed to force the robots in using the signal framework.

The design that have been implemented is shown in figure 7.2. In this design, robots explore the arena until they find the marker. Then, using the signal framework, they will attract their neighbours to the aggregation area.

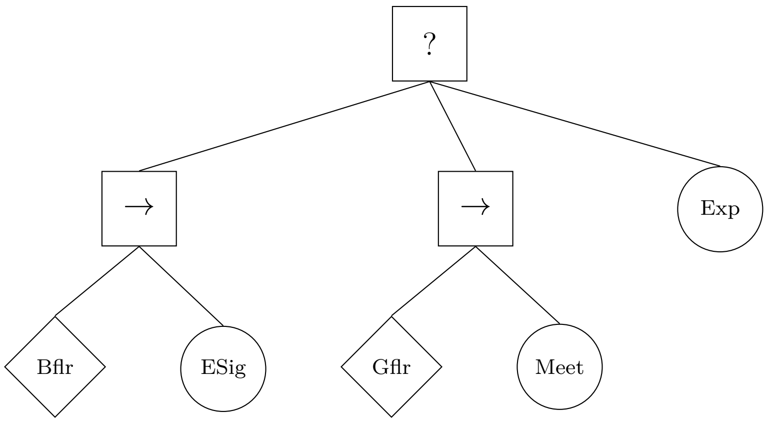


**Figure 7.2** – Hand design for the Marker Aggregation mission. The conditions and actions names have been abbreviated in the following way: Bflr: Black-floor; ESig: Emit Signal; Gflr: Gray-floor; Meet: Meeting; Exp: Exploration.

The tree is organised in three subtrees, that correspond to three situations of the robot. The first one is meant to be executed when the robot is on the marker. When the Black-floor condition returns *success*, the robot send a signal and the tree execution stop there. If the robot is not on the marker, the second subtree is executed and makes the robot go towards the marker using the Meet behaviour. If the Meet behaviour could not find matching signal, because the marker has not been found yet or because the emitting robots are too far away, the third subtree is executed and makes the robot explore the area, until it either find a signal or find the marker.

## Stop

In the Stop mission, robots must find a white spot in the arena. Once found, all robots must stop moving as quickly as possible.

The reference design is shown in figure 7.3. In this design, robots will send and forward signals to their neighbours to transmit the information of the white spot discovery. If a robot received a signal, it stops; if it do not receive any signal, it explore the arena in order to find the white spot.

The tree is divided in three subtrees. The first one tests if the robot is over the white spot. If that is the case, it sends a signal and do not move. If the robot is not on the white spot, the second subtree is executed and tests if the robot receives a signal, meaning that the white spot has been found. If that is the case, it retransmits the signal and do not move. If the third subtree is executed, it means that no signal is received and thus, with high probability, that the white spot has not been discovered yet. When it happens, the robot explores the arena until it

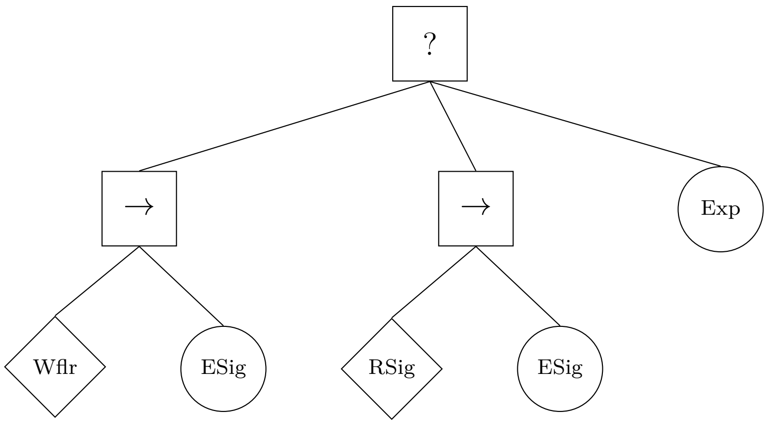


**Figure 7.3** – Hand design for the Stop mission. The conditions and actions names have been abbreviated in the following way: Bflr: Black-floor; ESig: Emit Signal; RSig: Receiving Signal; Exp: Exploration.

finds the white spot or receives a signal meaning that it has been discovered.

# Chapter 8

# Results

This chapter presents the results of the three design methods described previously, by mission. It gives a comparison between `Maple`, `Cedrata` and the manual design, evaluates the performance of the Iterated F-Race algorithm on the new set of modules by comparison with manual design, and verifies that the `Cedrata` still mitigates the effects of the reality gap as the previous flavours of AutoMoDe.

In the following sections, methods are often claimed to "perform significantly better" or "outperform" another method. It implies that a Wilcoxon rank sum test [56] has been performed with a confidence of 95%.

## 8.1 Foraging

The performance of the three design methods on the Foraging mission are shown in figure 8.1. The figure includes results with $100,000$ and $200,000$ design budgets for automatic methods, with 10 designs for each method and each budget. The results of the four manual designs and the reference design are the same in the two plots. The performance evolutions of `Maple` and `Cedrata` in function of the budget size are shown in figure 8.2. The figures include the results in both design and pseudo-reality environments.

The plots present similar results for all budget sizes, and a detailed inspection of the created behaviour trees shows that the adopted strategies, for each automatic method, are the same regardless of the budget. `Maple` presents slightly lower results on small budget sizes, but the difference is not significant and essentially comes from the module tuning. The fact that strategies does not evolve with the budget size implies that finding these strategies are not a difficult task for the optimisation process, because it manages to do it even with low budgets.

`Maple` uses the light to navigate more easily from the food sources to the nest. A typical tree is shown in figure 8.3. The first Selector subtree makes the robots

**(a)** Budget: $100,000$ simulation runs **(b)** Budget: $200,000$ simulation runs

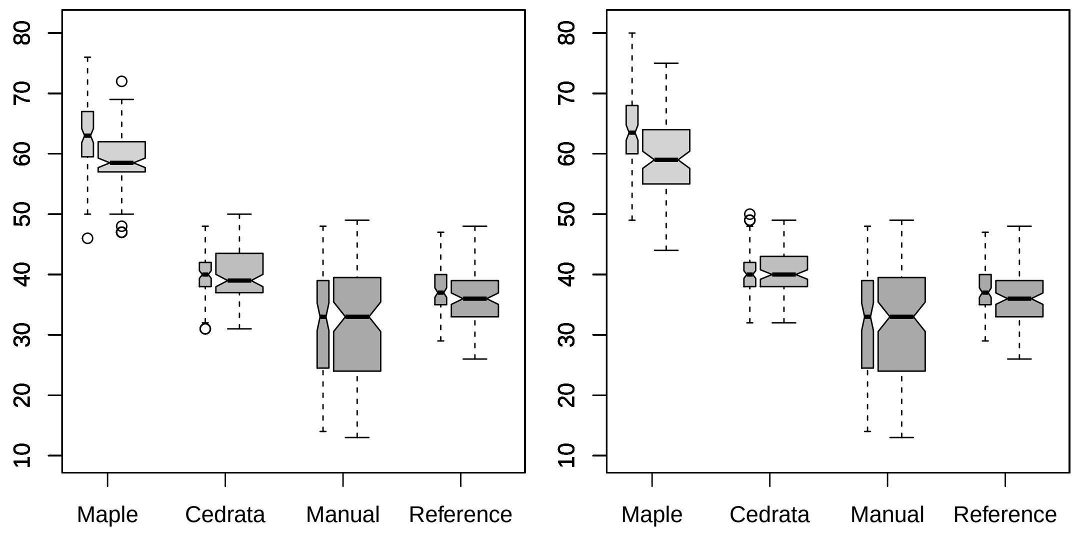


**Figure 8.1** – Foraging mission results. The performance of control software assessed in the design and pseudo-reality environments are showed using respectively narrow and wide boxes.

**(a)** `Maple` **(b)** `Cedrata`

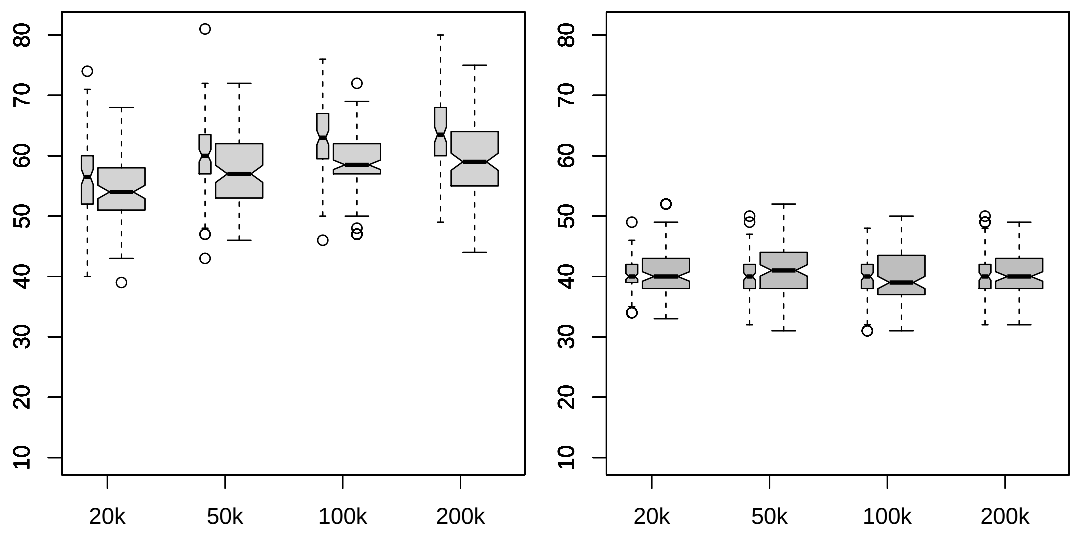


**Figure 8.2** – Results evolution per budget size on the Foraging mission. The performance of control software assessed in the design and pseudo-reality environments are showed using respectively narrow and wide boxes.

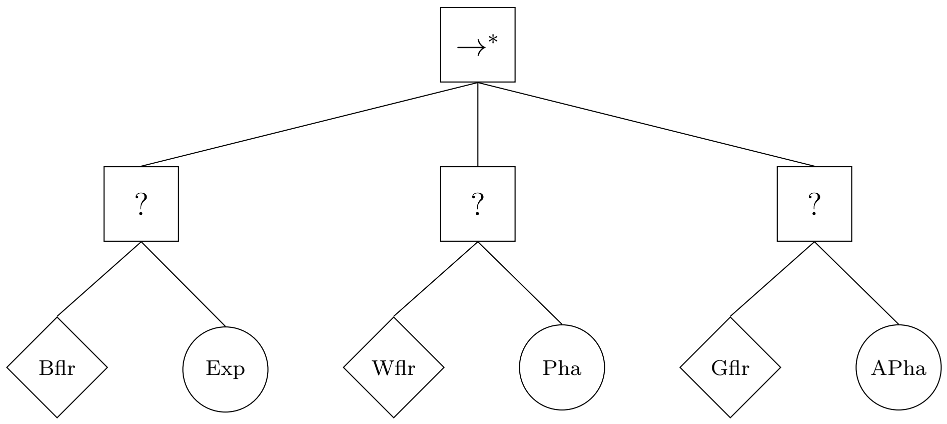


**Figure 8.3** – A typical design of `Maple` for the Foraging mission. Abbreviations: Bflr: Black-floor; Exp: Exploration; Wflr: White-floor; Pha: Phototaxis; Gflr: Gray-floor; APha: Antiphototaxis.

explore randomly the arena until they find a source. At this point, the Black-floor condition returns *success* and the robot switches to the second subtree, that makes it move in the direction of the light, which is placed behind the nest. When arrived, the third subtree is executed and makes the robot quickly leave the nest by moving away from the light. Although being efficient, this strategy does not make use of local communication.

With no light-related modules, the strategy adopted by `Cedrata` is more primitive. All the generated trees contain an Exploration behaviour which make the robot explore until the end of the mission. Eventually, the robot will pass over a food source and then over the nest. As in `Maple`, this strategy does not involve local interactions. A tree example is shown in figure 8.4. In this example, we can see an Exploration behaviour but also two conditions that could have been removed: the Fixed probability does not trigger a particular action, and the Neighbour-count is, in this case, too high to ever return *success*. As for the shown tree, a lot of generated control software contains what we can call *superfluous* modules, i.e. modules that do not play a part in the strategy. This is something that also happens sometimes with `Maple`, but in a way more limited way. This could be explained by the constraints on the tree structure: `Maple` impose subtrees that have one condition and one action node, leaving few possibilities to add extraneous node. On the contrary, a superfluous module can easily be added in `Cedrata` and, as it will not influence the performance of the swarm, be kept through the optimisation iterative process.

On all budget sizes and both design and pseudo-world environments, `Maple` outperforms `Cedrata`, but also the manual and reference desings. This can be simply explained by the presence and absence of light-based behaviours, which

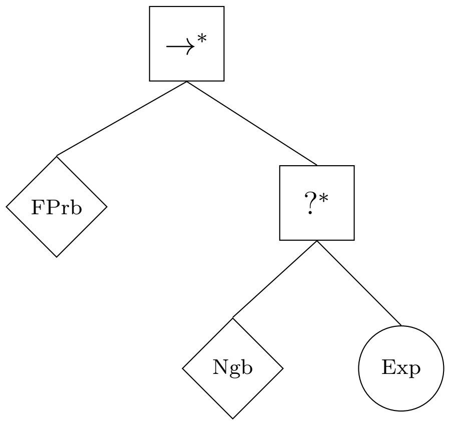


**Figure 8.4** – Example design of `Cedrata` for the Foraging mission. Abbreviations: FPrb: Fixed probability; Ngb: Neighbour-count; Exp: Exploration.

gives a huge advantage to `Maple`.

Human designers used the same strategy as `Cedrata`, which is based on the Exploration behaviour. However, it seems that the optimisation process does a better job at creating trees for this strategy, that can be explained by the budget size. Even the lower budget of $20,000$ simulation runs is a lot more than the runs that a human designer can execute in four hours, allowing the automatic method to test a greater amount of configurations and modules tuning. Although visually providing better results, the confidence of the Wilcoxon test is not high enough to conclude that `Cedrata` outperforms the manual designs.

No human designer used the strategy developed in the reference design, which is to use the signal framework to attract robots to the food sources. However, some of the manual designs reach as good performances as the reference design (and `Cedrata` do even better), which leads to two conclusions. Firstly, the use of signal that is done in the reference design does not give a substantial advantage. Secondly, it supports the motivations behind the introduction of behaviour trees in artificial intelligence, that is the convenience of the design. In this experiment, designers with no prior knowledge of behaviour trees are able to understand the concepts behind them and reach as good performances as the reference design, where we could have expected lower performances due to the four hours time window.

All three methods showed to be resistant to the reality gap, at least in the pseudo-world experiment. By extension, and using the conclusions of Ligot and Birattari [37], we can assume that they should also demonstrate to be resistant against the reality gap in the context of a real experiment.

## 8.2 Marker Aggregation

The performance of the three design methods on the Marker Aggregation mission are shown in figure 8.5. The figure includes results with $100,000$ and $200,000$ design budgets for automatic methods, with 10 designs for each method and each budget. The results of the four manual designs and the reference design are the same in the two plots. The performance evolutions of `Maple` and `Cedrata` in function of the budget size are shown in figure 8.6. The figures include the results in both design and pseudo-reality environments.

As in Foraging, `Maple` shows similar results in all budgets, meaning that the used strategy is probably easy to find for the optimisation process. A typical tree is shown in figure 8.7. The strategy works in three steps that correspond to the three subtrees. At first, the robot explore randomly the arena until it finds the black marker. Then, the robot aggregate with its neighbours. If the robot leaves the marker, the third subtree is executed and makes the robot stop. The Neighbour-count condition is parametrized in a way that it shouldn't return *success* at any point. The Stop behaviour is interesting, because it prevent the robot to leave the aggregation area by blocking it at the border of the marker; however robots in such state impede the others that are in search of the marker. This problem is solved in the reference design because robots in the marker can tell their neighbours that the marker is close using a signal, but `Maple` does not have such communication capabilities.

On the other side, `Cedrata` gives different performances for each budget sizes, with clear improvements for increasing budget sizes, as shown in figure 8.6b. It proposes two main strategies: one that is similar to what `Maple` does and one that is similar to the one adopted in the reference design. The global performance of the automatic design is influenced by how many strategies of each kind are generated. In this experiment, the designs generated with the lower budget sizes $20,000$ and $50,000$ contain only `Maple`-like designs. The experiment with a budget of $100,000$ simulation runs gave one reference-like strategy out of ten. When the budget increase to $200,000$ simulation runs, this proportion increase to four out of ten. This increasing proportion of reference-like design can directly be linked to the budget size. The implementation of the Iterated F-Race algorithm provided by `irace` [38] use the same initial number of iterations regardless of budget, using higher budgets to increase the exploration of new designs at each iteration. The reference-like design seems to be a difficult design to find for the optimisation process; and it is more easily discovered with greater exploration.

In low budget experiments, `Maple` and `Cedrata` use the same strategy, but `Maple` presents better performances. In higher budget experiments, `Cedrata` includes reference-like trees that are more efficient, but this proportion is not sufficient for the Wilcoxon test to conclude that `Cedrata` outperforms `Maple`. However,

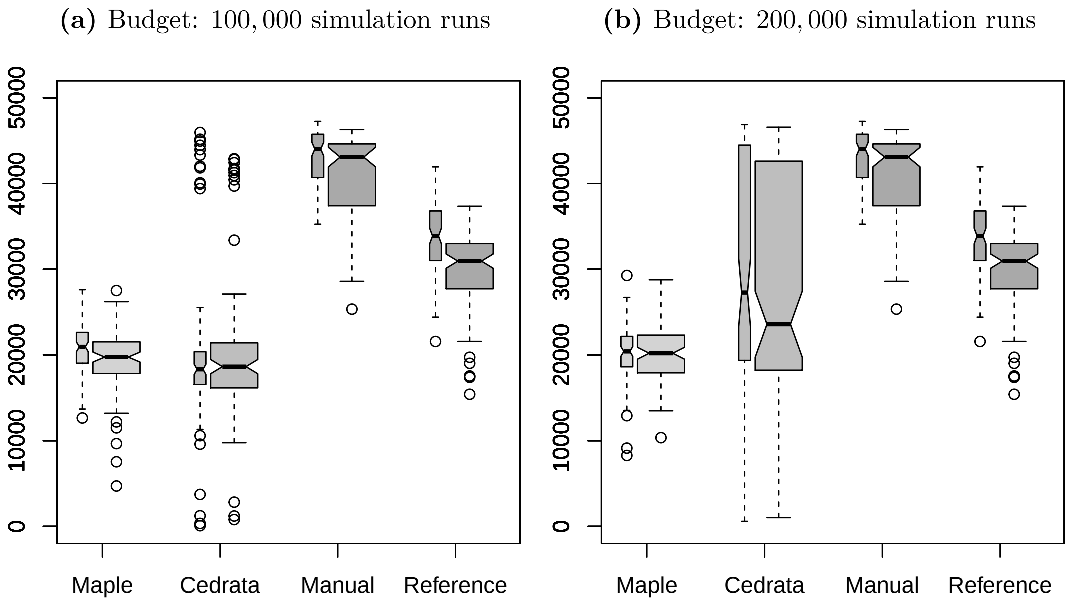


**Figure 8.5** – Marker Aggregation mission results. The performance of control software assessed in the design and pseudo-reality environments are showed using respectively narrow and wide boxes.

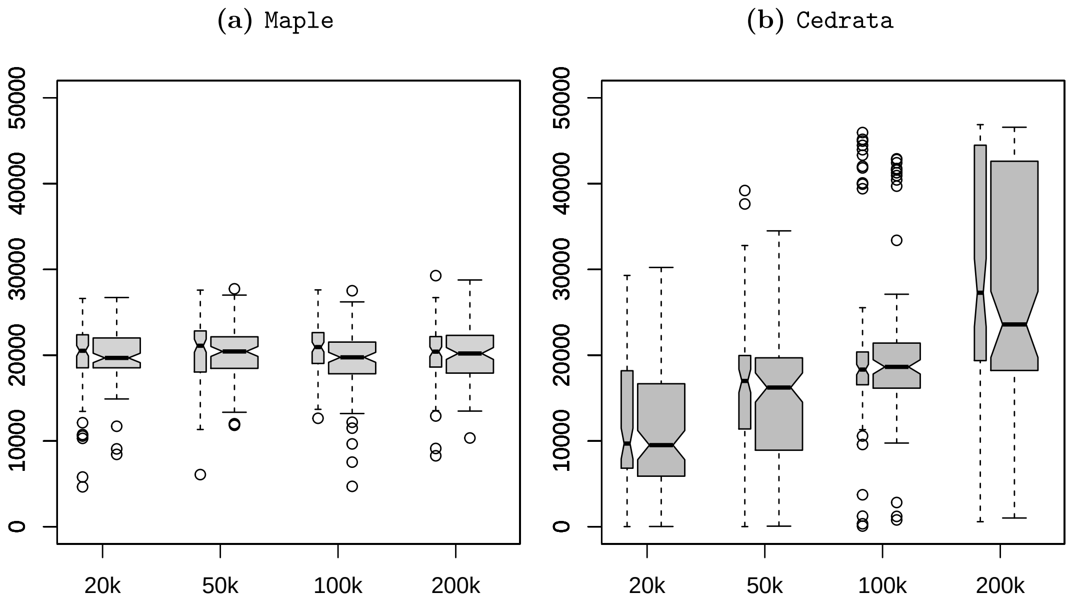


**Figure 8.6** – Results evolution per budget size on the Marker Aggregation mission. The performance of control software assessed in the design and pseudo-reality environments are showed using respectively narrow and wide boxes.

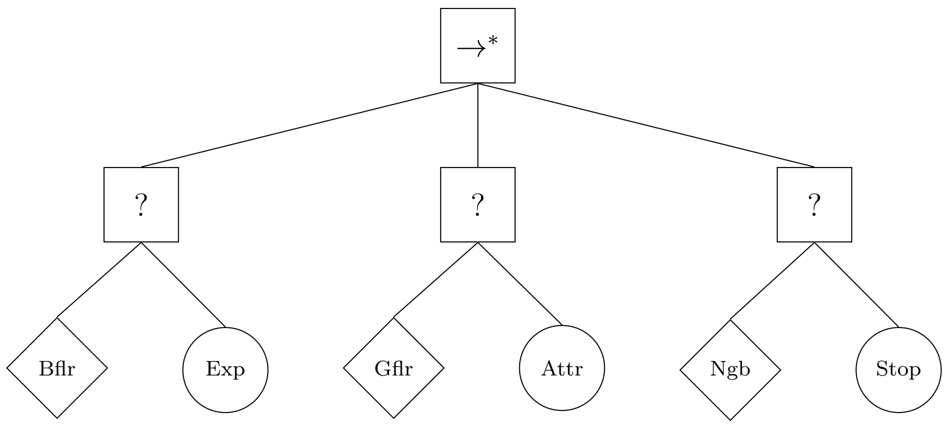


**Figure 8.7** – Typical design of `Maple` for the Marker Aggregation mission. Abbreviations: Bflr: Black-floor; Exp: Exploration; Gflr: Grey-floor; Attr: Attraction; Ngb: Neighbour-count.

we can expect the proportion to increase even more with higher budgets, until a time where `Cedrata` outperforms `Maple`.

`Cedrata` also leads to more variety in the trees: for a same strategy, trees with different topology are created. Trees also present superfluous modules, as seen in Foraging. An example of a `Maple`-like design is shown in figure 8.8. This tree contains some modules that are not useful: the two Emit Signal behaviours send signals that will never be perceived, as no other signal-based modules are present; the Sequence* subtree will always only execute its first child because the Stop behaviour will always return *running*. This can be explained using the same reasoning that has been made about the Foraging mission, which is that `Cedrata` is more flexible on the tree structure.

The manual designs use a strategy that is similar to the one used in the reference design but with better module tuning, making them outperforming the reference design. As in Foraging, it clearly highlights the convenience of behaviour trees. The manual designs also outperform `Cedrata`, although for the $200,000$ budget size the confidence of the Wilcoxon test is not sufficient enough to draw a strict dominance. However, by considering only designs trees that uses the same strategy, `Cedrata` performs similarly. This is particularly visible on the $100,000$ budget size plot: the top points above the $40,000$ score correspond to the one tree that uses the reference-like strategy, and they clearly present equivalent performances to the manual designs. As in the comparison with `Maple`, we can expect that, with a higher budget, the proportion of reference-like trees increase, and that `Cedrata` finishes by giving equivalent results to the manual designs for all generated trees.

All three design methods successfully manage to mitigate the effects of the pseudo-reality gap. The performance in pseudo-reality of the reference design

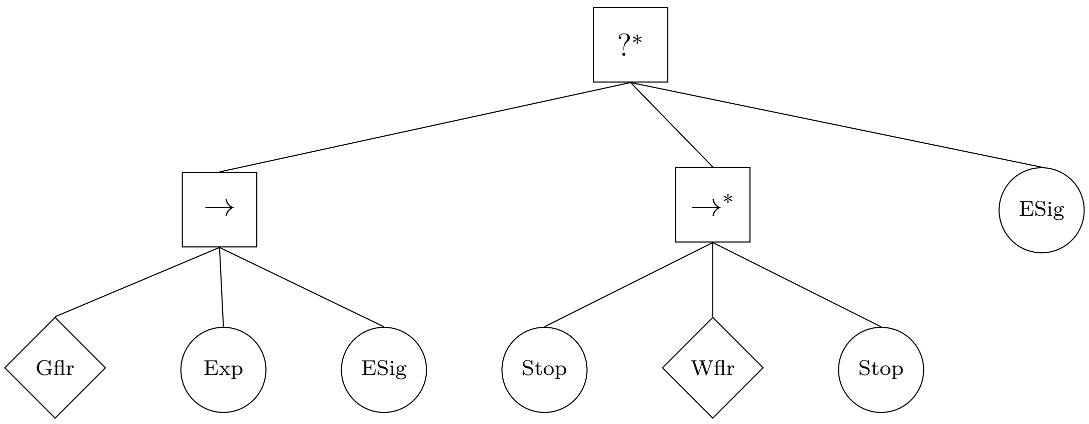


**Figure 8.8** – Example of `Maple`-like tree of `Cedrata` for the Marker Aggregation mission. Abbreviations: Gflr: Grey-floor; Exp: Exploration; ESig: Emit Signal; Wflr: White-floor.

show a bit lower performance, but the confidence on the Wilcoxon test does not allow to conclude on a statistical difference.

## 8.3 Stop

The performance of the three design methods on the Stop mission are shown in figure 8.9. The figure includes results with $100,000$ and $200,000$ design budgets for automatic methods, with 10 designs for each method and each budget. The results of the four manual designs and the reference design are the same in the two plots. The performance evolutions of `Maple` and `Cedrata` in function of the budget size are shown in figure 8.10. The figures include the results in both design and pseudo-reality environments.

As in Foraging, both automatic design methods show similar performance for the two budget sizes and use the same strategies in both experiments, meaning that strategies exposed here are easy to find for the optimisation process.

A behaviour tree example from `Maple` is shown in figure 8.11. It makes the robots explore the arena until they find sufficient amount of neighbours to stop. A robot often finds the white spot before stopping. This behaviour allows the swarm to reach an honourable score, but it fails to fill the objective of the mission which is the communication of the white spot discovery. This failure is expected, as `Maple` do not have direct communication at its disposal. The figure 8.11 also illustrate how `Maple` can suffer from superfluous modules as in `Cedrata`: here, we have two subtrees that could have been merged into one.

`Cedrata` uses a different strategy: robots simply isolate form the others. It

**(a)** Budget: 100, 000 simulation runs **(b)** Budget: 200, 000 simulation runs

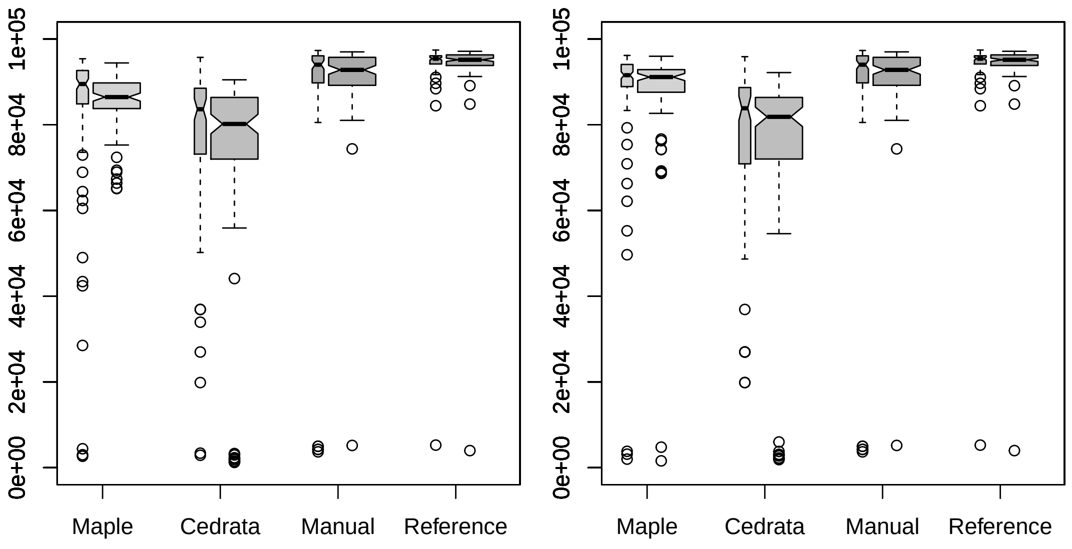


**Figure 8.9** – Stop mission results. The performance of control software assessed in the design and pseudo-reality environments are showed using respectively narrow and wide boxes.

**(a)** `Maple` **(b)** `Cedrata`

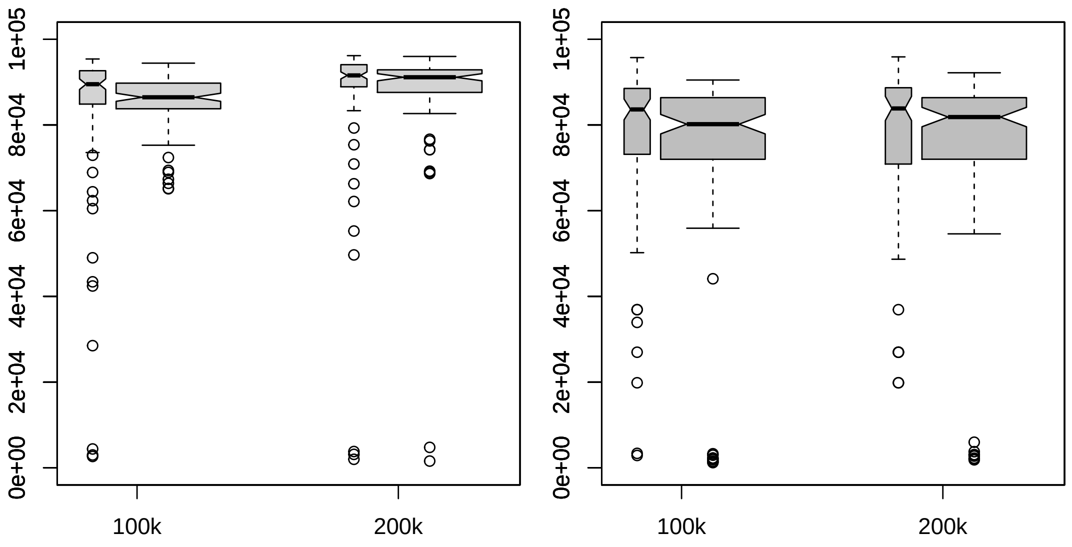


**Figure 8.10** – Results evolution per budget size on the Stop mission. The performance of control software assessed in the design and pseudo-reality environments are showed using respectively narrow and wide boxes.

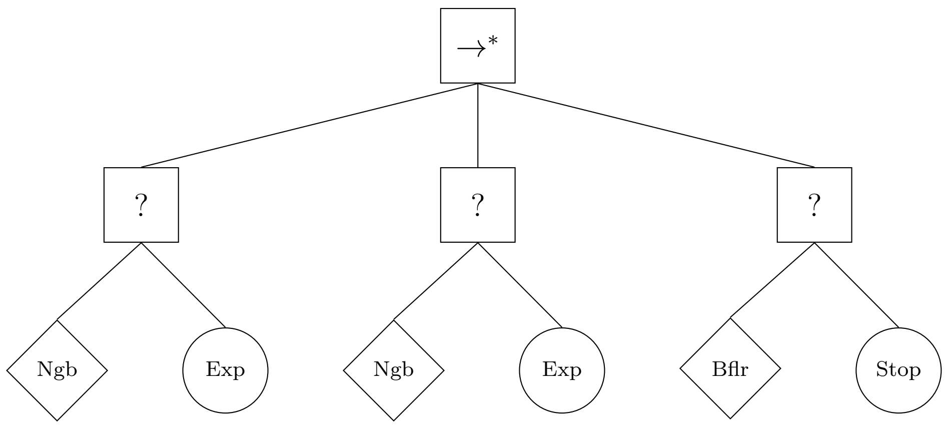


**Figure 8.11** – Example design of `Maple` for the Stop mission. Abbreviations: Ngb: Neighbour-count; Exp: Exploration; Bflr: Black-floor.

makes the swarm expand and covers all the arena, giving a high probability to find the white spot. They manage to get a relatively high score because the objective function consider that a robot that moves slower than $5mm$ per second is not moving, and robots in the Isolation behaviour often pass under this threshold. Some trees have an Exploration behaviour for when robots do not detect neighbours, as showed in figure 8.12. This tree contains again superfluous modules, especially the three Receiving signal conditions for signal that can't be sent, which is very common for control software generated with `Cedrata`.

The results of `Cedrata` on the Stop mission are surprising: they are lower than the ones of `Maple`, although without statistical significance according to the Wilcoxon test. We could have expected that `Cedrata` reaches at least the same performances as `Maple`, considering that the generated trees of `Maple` on this mission are in the search space of `Cedrata`. Besides, `Cedrata` has direct communication modules at its disposal that should give it a substantial advantage over `Maple`. In this mission, the optimisation algorithm of `Cedrata` fails to correctly exploit the modules and the behaviour trees characteristics at its disposal. The reasons of that failure are unclear, but following the discussion on the Marker Aggregation results the large control software space of `Cedrata` could make it difficult for the optimisation process to find interesting strategies because they are lost in a too big amount of less-efficient ones. This hypothesis gets more convincing given the fact that the strategy adopted by `Cedrata` is mainly based on a single Isolation behaviour, making it easy to find.

The human designers used strategies that are similar to the one used as reference, making the manual designs and the reference design the only methods that fulfil satisfactorily the objective of the mission, i.e. the communication of the white spot discovery. This leads to higher scores, making the reference design outper-

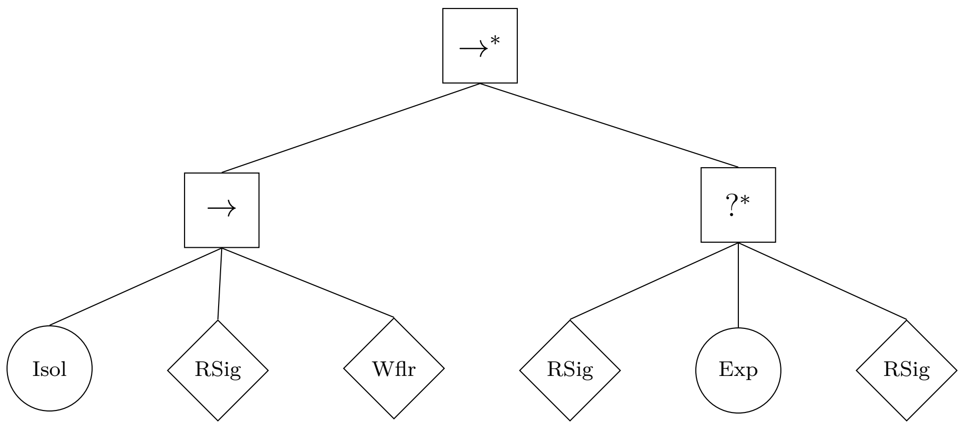


**Figure 8.12** – Example design of `Cedrata` for the Stop mission. Abbreviations: Isol: Isolation; RSig: Receiving signal; Wflr: White-floor; Exp: Exploration.

forming both `Maple` and `Cedrata`, and the manual design outperforming `Cedrata`. The Wilcoxon test does not gives enough confidence to conclude that the manual design also outperforms `Maple`, even if visually the results seems to be globally higher.

Unlike previous missions, the manual designs are less efficient than the reference one. However, the difference is minimal, and we can still support the idea that behaviour trees are convenient to use.

For all methods, some simulation runs, either in the design or in the pseudo-reality environment, show very low results (almost equal to zero) compared to the other simulation runs. These results correspond to some experiments where no robot find the white spot. Visually, it is the reference design that miss the least to find the white spot, but keep in mind that plots of different methods are not based on the same amount of designs: ten for the automatic methods, four for the manual designs and only one reference design.

As in the previous two missions, all methods showed to be resistant against the pseudo-reality gap.

# Chapter 9

# Conclusion

In this work, a new flavour of AutoMoDe called `Cedrata` has been introduced to pursue the work started with a previous one called `Maple`, which introduced behaviour trees. `Maple` uses modules from earlier flavours, which have been designed for finite state machines. This forces the behaviour tree to adopt a particular structure. `Cedrata` introduce a new set of modules that are specifically designed for behaviour trees, and allowing the tree to have a more flexible structure.

Both methods have been compared on three different missions. These experiments included manual designs, done by human designers on the module set of `Cedrata` under the constraint of four design hours, and reference designs, done as manual ones but without time constraints and with the objective of serving as examples. Multiple observations can be extracted from the results.

The manual designs are, in average, as good as the reference ones. It means that, under the experimental conditions of four hours, designers with no prior knowledge of behaviour trees are able to understand and use them to solve missions efficiently. This highlights one of the advantages of behaviour trees, that are claimed to be convenient to design agents in artificial intelligence. This also support that behaviour trees should be an interesting choice for AutoMoDe flavours control software structure.

Behaviour trees are convenient to design for human designers, but it seems to be more difficult for automatic processes. In this work, `Cedrata` was unable to reach as good performances as the manual or reference designs on some missions. This fraction seems to increase with experiment budget, but no experiment here have a sufficiently high budget to make `Cedrata` a serious alternative to manual designs. Based on the Marker Aggregation mission results, we could have stated that it is only a matter of budget, and that `Cedrata` should give performant results as soon as we provide enough budget. However, the results in the other two missions contradicts that, since we cannot observe any evolution that is function of the budget. The problem may lie in the optimisation algorithm, which may be unadapted for `Cedrata`. One of the main problem may be the size of the search space, which is larger in `Cedrata` than it is in `Maple`. Due to the flexible structure of the trees, the search space contains a lot of redundant control software due to the high quantities of superfluous modules that we find in `Cedrata` trees.

In this work, a new set of modules was introduced to make better use of the return values that is a part of the functioning of behaviour trees. However, the comparison between new modules and ancient ones, respectively used by `Cedrata` and `Maple`, do not show clear evidence that the new modules improved the performance of the swarm. In the Foraging mission, `Maple` performs better, but it uses light-based behaviours that have been removed from the new module set used by `Cedrata`. In the Marker Aggregation mission, `Cedrata` performs better, but the mission is designed to promote communication between the robots and `Cedrata` possess the new signal-based behaviours that are not accessible to `Maple`. Clearly, the performance of the automatic methods depends primarily on the mission and how well suited the design method and its reference model are to deal with the information present in the mission, and not on the specificity of modules against the structure in which they are assembled in. This does not allow to draw a conclusion about the dominance of the one design method over the other.

The Stop mission promotes communication as Marker Aggregation, but instead of performing well as we could have expected, `Cedrata` is the method that performs the worse among all. This is surprising because `Cedrata` have a advantage over `Maple` as it possess signal-based modules. At least, `Cedrata` should have reached the performances of `Maple` since the control software generated by the latter are included in the search space of `Cedrata`. This highlights even more the fact that the optimisation process fails in creating good control software in `Cedrata`.

## 9.1 Future work

`Cedrata` is a second step, after `Maple`, into making behaviour trees a usable structure to use in AutoMoDe, but multiple things still need to be brightened up. In particular, the comparison between `Maple` and `Cedrata` gives interesting observations but leaves unanswered questions.

Firstly, `Cedrata` introduces both a new set of modules and a new tree structure and the results provided make it hard to attribute a better or worse performance to one of the changes. For example, in the Stop mission, the `Cedrata` reference model should have lead to better performances since it adds the possibility to exchange signals; however `Cedrata` led to lower results than `Maple`. The problem may reside in the set of modules, the flexibility of the tree structure, or both of them. Without further experiments, it is difficult to attribute the results to one of the proposed causes. One solution to isolate the structure flexibility would be to include removed modules from `Maple` back in `Cedrata`, and redo the comparison on missions where `Maple` is performant, like Foraging. However, we already stated that the optimisation process has difficulties searching in the control software space of `Cedrata`, and this addition will probably worsen the performances of `Cedrata` instead of improving them. To dig deeper in that direction, experiments can be extended to include new flavours that use `Cedrata` modules but with a restricted architecture or `Maple` modules with a more flexible architecture. Another solution would be to use different missions, that are designed in a way such that `Maple` and `Cedrata` come up with similar strategies, to evaluate how easily the optimisation process find these strategies.

Another observation draw from the results is that `Cedrata` includes more superfluous modules in its architecture than `Maple`. This leads to the hypothesis that the number of such modules is related to the freedom given on the control structure. An interesting experiment to confirm or infirm that would be to run different optimisation processes on the same set of modules but with increasing degrees of freedom.

More generally, Iterated F-Race, the optimisation algorithm used by `Cedrata`, seems to be unable to efficiently explore the control software space. Increasing budget would be a first solution, but will likely not work on all missions, since the results in the Foraging and the Stop mission did not show to improve with the budget. Another idea would be to assess the use of other optimisation algorithms, like the Simulated Annealing that already has been tested in `IcePop` [34] or the Novelty Search [23] that is a divergent algorithm that promotes exploration. Research can also be oriented with the idea of reducing the number of superfluous modules, because it will reduce the size of the search space without reducing its possibilities since superfluous modules are, by definition, not altering the behaviour that emerge from their tree.

# Bibliography

[1] J Andrew Bagnell et al. “An integrated system for autonomous robotics manipulation”. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 2955–2962.

[2] Levent Bayindir and Erol Şahin. “A review of studies in swarm robotics”. In: *Turkish Journal of Electrical Engineering & Computer Sciences* 15.2 (2007), pp. 115–147.

[3] Gerardo Beni. “From swarm intelligence to swarm robotics”. In: *International Workshop on Swarm Robotics*. Springer. 2004, pp. 1–9.

[4] Mauro Birattari et al. “A Racing Algorithm for Configuring Metaheuristics.” In: *Gecco*. Vol. 2. 2002. 2002.

[5] Mauro Birattari et al. “Automatic off-line design of robot swarms: a manifesto”. In: *Frontiers in Robotics and AI* 6 (2019), p. 59.

[6] Mauro Birattari et al. “F-Race and iterated F-Race: An overview”. In: *Experimental methods for the analysis of optimization algorithms*. Springer, 2010, pp. 311–336.

[7] Christian Blum and Xiaodong Li. “Swarm intelligence in optimization”. In: *Swarm intelligence*. Springer, 2008, pp. 43–85.

[8] Iva Bojic et al. “Extending the JADE agent behaviour model with JBehaviourTrees Framework”. In: *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*. Springer. 2011, pp. 159–168.

[9] Manuele Brambilla et al. “Swarm robotics: a review from the swarm engineering perspective”. In: *Swarm Intelligence* 7.1 (2013), pp. 1–41.

[10] Michele Colledanchise and Petter Ögren. “Behavior trees in robotics and AI: an introduction”. In: *arXiv preprint arXiv:1709.00084* (2017).

[11] Michele Colledanchise and Petter Ögren. “How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees”. In: *IEEE Transactions on robotics* 33.2 (2016), pp. 372–389.

[12] Charles Darwin. *The origin of species.* 1859.

[13] Edsger W Dijkstra. “Letters to the editor: go to statement considered harmful”. In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

[14] Stephane Doncieux and Jean-Baptiste Mouret. “Beyond black-box optimization: a review of selective pressures for evolutionary robotics”. In: *Evolutionary Intelligence* 7.2 (2014), pp. 71–93.

[15] Marco Dorigo, Mauro Birattari, et al. “Swarm intelligence.” In: *Scholarpedia* 2.9 (2007), p. 1462.

[16] Marco Dorigo, Alberto Colorni, and Vittorio Maniezzo. *Distributed optimization by ant colonies.* 1991.

[17] Miguel Duarte et al. “Evolution of collective behaviors for a real swarm of aquatic surface robots”. In: *PloS one* 11.3 (2016).

[18] Gianpiero Francesca et al. “Analysing an evolved robotic behaviour using a biological model of collegial decision making”. In: *International Conference on Simulation of Adaptive Behavior.* Springer. 2012, pp. 381–390.

[19] Gianpiero Francesca et al. “AutoMoDe-Chocolate: automatic design of control software for robot swarms”. In: *Swarm Intelligence* 9.2-3 (2015), pp. 125–152.

[20] Gianpiero Francesca 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.

[21] David Garzón Ramos and Mauro Birattari. “Automatic Design of Collective Behaviors for Robots that Can Display and Perceive Colors”. In: *Applied Sciences* 10.13 (2020), p. 4654.

[22] Stuart Geman, Elie Bienenstock, and René Doursat. “Neural networks and the bias/variance dilemma”. In: *Neural computation* 4.1 (1992), pp. 1–58.

[23] Jorge Gomes, Paulo Urbano, and Anders Lyhne Christensen. “Introducing novelty search in evolutionary swarm robotics”. In: *International Conference on Swarm Intelligence.* Springer. 2012, pp. 85–96.

[24] Plerre-P Grassé. “La reconstruction du nid et les coordinations interindividuelles chez *Bellicositermes Natalensis* et *Cubitermes Sp.* La théorie de la stigmergie: Essai d’interprétation du comportement des termites constructeurs”. In: *Insectes sociaux* 6.1 (1959), pp. 41–80.

[25] Inman Harvey, Philip Husbands, Dave Cliff, et al. *Issues in evolutionary robotics.* School of Cognitive and Computing Sciences, University of Sussex, 1992.

[26] Ken Hasselmann, Frédéric Robert, and Mauro Birattari. "Automatic design of communication-based behaviors for robot swarms". In: *International Conference on Swarm Intelligence.* Springer. 2018, pp. 16–29.

[27] Sabine Hauert, Jean-Christophe Zufferey, and Dario Floreano. "Evolved swarming without positioning information: an application in aerial communication relay". In: *Autonomous Robots* 26.1 (2009), pp. 21–32.

[28] Nick Jakobi, Phil Husbands, and Inman Harvey. "Noise and the reality gap: The use of simulation in evolutionary robotics". In: *European Conference on Artificial Life.* Springer. 1995, pp. 704–720.

[29] Anja Johansson and Pierangelo Dell'Acqua. "Emotional behavior trees". In: *2012 IEEE Conference on Computational Intelligence and Games (CIG).* IEEE. 2012, pp. 355–362.

[30] Simon Jones et al. "Evolving behaviour trees for swarm robotics". In: *Distributed Autonomous Robotic Systems.* Springer, 2018, pp. 487–501.

[31] Simon Jones et al. "Onboard evolution of understandable swarm behaviors". In: *Advanced Intelligent Systems* 1.6 (2019), p. 1900031.

[32] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[33] James Kennedy and Russell Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95-International Conference on Neural Networks.* Vol. 4. IEEE. 1995, pp. 1942–1948.

[34] Jonas Kuckling, Keneth Ubeda Arriaza, and Mauro Birattari. "Simulated Annealing as an Optimization Algorithm in the Automatic Modular Design of Control Software for Robot Swarms". In: *BNAIC 2019: Artificial Intelligence.* 2019.

[35] Jonas Kuckling et al. "Behavior trees as a control architecture in the automatic modular design of robot swarms". In: *International Conference on Swarm Intelligence.* Springer. 2018, pp. 30–43.

[36] Steve Lawrence, Ah Chung Tsoi, and Andrew D Back. "Function approximation with neural networks and local methods: Bias, variance and smoothness". In: *Australian conference on neural networks.* Vol. 1621. Australian National University. 1996.

[37] Antoine Ligot and Mauro Birattari. “Simulation-only experiments to mimic the effects of the reality gap in the automatic design of robot swarms”. In: *Swarm Intelligence* 14.1 (2020), pp. 1–24.

[38] Manuel López-Ibáñez et al. “The irace package: Iterated racing for automatic algorithm configuration”. In: *Operations Research Perspectives* 3 (2016), pp. 43–58.

[39] Alejandro Marzinotto et al. “Towards a unified behavior trees framework for robot control”. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 5420–5427.

[40] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2009.

[41] Francesco Mondada et al. “The e-puck, a robot designed for education in engineering”. In: *Proceedings of the 9th conference on autonomous robot systems and competitions*. Vol. 1. CONF. IPCB: Instituto Politécnico de Castelo Branco. 2009, pp. 59–65.

[42] Aadesh Neupane and Michael A Goodrich. “Learning Swarm Behaviors using Grammatical Evolution and Behavior Trees.” In: *IJCAI*. 2019, pp. 513–520.

[43] Miguel Nicolau et al. “Evolutionary behavior tree approaches for navigating platform games”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 9.3 (2016), pp. 227–238.

[44] Stefano Nolfi, Dario Floreano, and Director Dario Floreano. *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT press, 2000.

[45] Stefano Nolfi et al. “How to evolve autonomous robots: Different approaches in evolutionary robotics”. In: *Artificial life iv: Proceedings of the fourth international workshop on the synthesis and simulation of living systems*. MIT Press. 1994, pp. 190–197.

[46] Petter Ogren. “Increasing modularity of UAV control systems using computer game behavior trees”. In: *Aiaa guidance, navigation, and control conference*. 2012, p. 4458.

[47] Carlo Pinciroli et al. “ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems”. In: *Swarm intelligence* 6.4 (2012), pp. 271–295.

[48] Michael Ruzhansky, Niyaz Tokmagambetov, and Berikbol Torebek. “A projection model of COVID-19 pandemic for Belgium”. In: *medRxiv* (2020).

[49] Erol Şahin. “Swarm robotics: From sources of inspiration to domains of application”. In: *International workshop on swarm robotics*. Springer. 2004, pp. 10–20.

[50] Muhammad Salman, Antoine Ligot, and Mauro Birattari. "Concurrent design of control software and configuration of hardware for robot swarms under economic constraints". In: *PeerJ Computer Science* 5 (2019), e221.

[51] Kirk YW Scheper et al. "Behavior trees for evolutionary robotics". In: *Artificial life* 22.1 (2016), pp. 23–48.

[52] Gaëtan Spaey et al. "Comparison of Different Exploration Schemes in the Automatic Modular Design of Robot Swarms." In: *BNAIC/BENELEARN*. 2019.

[53] Valerio Sperati, Vito Trianni, and Stefano Nolfi. "Self-organised path formation in a swarm of robots". In: *Swarm Intelligence* 5.2 (2011), pp. 97–119.

[54] R Core Team et al. "R: A language and environment for statistical computing". In: (2013).

[55] Guy Theraulaz et al. "The formation of spatial patterns in social insects: from simple behaviours to complex structures". In: *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 361.1807 (2003), pp. 1263–1282.

[56] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. "Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test". In: *Selected tables in mathematical statistics* 1 (1970), pp. 171–259.

[57] Juan Cristóbal Zagal, Javier Ruiz-del-Solar, and Paul Vallejos. "Back to reality: Crossing the reality gap in evolutionary robotics". In: *IFAC Proceedings Volumes* 37.8 (2004), pp. 834–839.