ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS OF CESENA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Second Cycle Degree in Computer Science and Engineering

SELF-ASSEMBLY IN VOXEL-BASED ROBOTS

Thesis in
INTELLIGENT ROBOTIC SYSTEMS

Supervisor
Prof. ANDREA ROLI

Co-Supervisor
Prof. ERIC MEDVET
Dott. MICHELE BRACCINI
Dott. PAOLO BALDINI

Presented by
FILIPPO BENVENUTI

Academic Year 2022 – 2023

To my beloved Lucia Sacchetti

Contents

Introduction

Self-assembly is an innovative domain, a process whereby individual robotic units, without centralized control, autonomously organize into desired structures or patterns to achieve complex tasks. Cherished in fiction films and cartoons, its significance in literature appears to be awaiting further exploration, as it has not quite reached its pinnacle or received the level of formal recognition it deserves. One aim of this master thesis is to provide a comprehensive understanding of self-assembly mechanisms, beginning with a detailed examination of its definitions, properties, and historical advancements, as outlined in pertinent literature. Additionally, this thesis aims to advance the field through experimental research for further validating the feasibility of self-assembly mechanisms. Through rigorous analysis of the experimental outcomes, we aim to offer valuable insights that can enrich the existing body of knowledge in this area. Furthermore, we aim to lay the groundwork for potential future developments, thereby fostering continued exploration and expansion of the research initiated in this thesis.

The thesis is divided into three main chapters:

1. **Self assembly**: the first chapter elucidates properties, categories, and definition of self-assembly, encompassing similar terms to disambiguate among them. An historical overview, spanning from initial developments to the latest advancements, is provided to furnish the reader with a comprehensive understanding of the projects and experiments related to self-assembly documented in the literature.

2. **Technologies**: the second chapter comprises a detailed description of the software and hardware used to undertake the experiments outlined in this thesis. It includes an illustration of *2D Robot Evolution*, the Java framework employed for describing and executing evolutionary experiments. Additionally, Python scripts utilized for processing and visualizing raw data extracted from experimental results are provided. Furthermore, a detailed description of the cluster utilized to run the experiments is included.

3. **Research experiments**: The third chapter delineates all computational research work undertaken, including descriptions of all preliminary experiments categorized by methodology (Genetic evolutionary and CMA-ES). It encompasses a detailed account of the final experiment, both in terms of setup (e.g., experiment description, parameters, fitness) and the results obtained(e.g., videos screenshots and numerical graphs). Moreover, it includes an exposition of potential future avenues for research that can be pursued based on this foundation.

Chapter 1

Self assembly

In this chapter, we provide a comprehensive description of the definitions found in the literature about **Self-assembly** and its peculiarities, followed by a collection of robotic projects that exploit those kinds of potentialities or proof of concepts.

1.1 Definitions

The definition of *self-assembly* is not always provided in the contributions that can be found in the literature. Most of the time, articles avoid elaborating on the theory behind it, concentrating instead on the project being developed. In the rare cases in which a definition is provided, the same one from a general point of view is reported: "Self-assembly is the phenomenon in which a collection of particles spontaneously arrange themselves into a coherent structure" [19].
This definition refers to the behavior we can observe in natural contexts like biology[4] (ants, wasps..) and chemistry[19] (molecule formation..), the same one we would like to be able to program into our robots. When it comes to the argument of robotics, this definition leaves an important characteristic implicit: "It is assumed that the individual units have onboard power and the ability to locomote, or are deterministically moved to the appropriate place by other powered units"[37]. This significantly impacts how we think of robots capable of self-assembly and dismisses the visionary idea given to us by films or cartoons, where those kinds of robots eventually learn how to fly by being attracted by a non-specified force.

1.1.1 Properties

"The design of components that organize themselves into desired patterns and functions is the key to applications of self-assembly" [19], remarks the importance of taking into consideration the structure of robots: the ability to move, the need of a particular environment, forces used to ensemble 1.1.2 and communications involved 1.1.2, will change the whole use case scenarios due to a different set of abilities which can be exploited.
In a general manner, calling each building component a unit and dividing units into groups, the properties describing a self-assembly robotic system can be summarized as[32]:

- **Replaceability**: All the units in a group are the same, have the same functions, and can be replaced by any other units of the same group.
- **Design freedom**: One can freely design the system by connecting the units like LEGO blocks, the only limits should be physics rules (feasibility).
- **Scale extensibility**: Capacity of scale extension or contraction; the scale of the system can be changed by adding or removing units.

It is important to keep in mind that we are now only considering self-assembly systems in the field of robotics, in other contexts not all of those properties can be found (e.g., with molecules we cannot play as we do with LEGOs, or can we?).

1.1.2 Categories

Just a few examples of self-assembly applied to robotics can be found in the literature, most of them being described in section 1.2, each introducing a new category in terms of: "Self-assembly comes in two modes, category A and category B". For the ease of the reader, we list a description of all of these categories:

- **Homoheterogeny**: robot units are divided into groups, and each group contains only one type of unit being all the same replicated.
  - **Homogeneous**: only one group exists, units are all the same; for example, ants[4] building a bridge or piling up.
  - **Heterogeneous**: multiple groups exist, and each group contains replicated fundamental units for each "subtask"; for example, in assembling a bike, we could have a group of resistant units devoted

to structure, a group of soft ones for rubber wheels, and a group of strong units for the chain.

- **Actipassivity**: the kind of energy involved between units, the way energy is used to accomplish the ensemble[19],[38].
  - **Passive**: units interact according to their geometry or surface chemistry and tend towards a thermodynamic equilibrium in which they are assembled, then stop dissipating energy; for example, an instant tent, that once released reaches its stable position mounting up itself. Keep in mind that instant tents are not self-assembled, but the energy involved in this process gives a good example to better understand this kind of passivity.
  - **Active**: units may expend energy to accept some interactions with other particles while rejecting others according to a program, never reach a final stable form, always dissipate energy, for example, units equipped with magnets that can be turned on and off to attach and detach from other units.
- **Stynamicity**: if we call “body” the final aggregation of units, whether or not the body contains moving parts.
  - **Static**: a formed body that does not require any moving units to be functional; for example, furniture in a house, once a chair is built it only needs to stay built, it can be moved, but it does not need to have moving units.
  - **Dynamic**: once the body is formed, it still contains moving units which are useful for the functionality of him self; for example, a walking robot needs to have moving units in order to be able to walk. Another perfect but visionary example is in “Big Hero 6” [3], the robot of “Hiro Hamada” during the bots-fight scene, dynamically self-assembled from “Microbots”.
- **Link type**: the way units link to or attract each other, permanently or intermittently (whether or not once attached they can be detached), softly or hardly (if connection fixes two units movements or leave at least one rotational free axe).
  - **Grab**: units link mechanically, through moving parts able to lock one or more units, usually intermittently and both softly and hardly. For example, “ants link their legs and bodies with their tarsal claws, forming layer upon interlocking layer of chains and nets of workers”[4].

  - **Magnetic**: units attract magnetically, through permanent magnets or electromagnets, softly or hardly based on the power used and the surface conformation, for example, kid toys Geomag[1].
  - **Bond**: units "glue" to each other, both permanently and intermittently, both softly and hardly, for example, magnetism can be seen as a bond or more specifically velcro which effectively glues units.

- **Centralization**: the place where units control logic is put.
  - **Centralized**: all the control logic is grouped in a single point, giving units an external controller.
  - **Decentralized** or **distributed**[32]: every unit has its controller and shares the same control logic with units of the same group. In this case, units usually require the ability to communicate with each other, distinguishing from local and global, if units can only communicate with other neighboring units or the whole of them indistinctly.

1.1.3 Disambiguation

Self-assembly is closely associated with other terms in the form "self-*". For the ease of the reader, we have listed the most related terms to provide a concise definition and elucidate the primary distinctions with self-assembly:

- **Self-organizing**: "The main questions in the area of programmed self-organization concern the ability to engineer the global behavior of a system by means of local rules" [19], at a general point of view self-organization underline the paradigm of how robots in a swarm are programmed: a bottom-up approach where the behavior of single entities defines the global result. Self-organizing limited to robotics leads to the usage of "modular robots that can latch on to each other, rotate or translate with respect to each other, and communicate with each other by means of peer-to-peer communication devices. Through local interactions, a group of modular robots can reconfigure into a variety of shapes, repair itself, and even self-replicate" [19], constrained like this self-organizing can be seen as a synonym of self-assembly; however, keep in mind that to avoid ambiguities, it is important to always specify the context of swarm robotics.

[1]`https://www.geomagworld.com/`

- **Self-reconfiguration**: "the ability to transition from one topology to another using a series of attachments and detachments" [37], ineluctably a fundamental aspect of self-assembly, but not one of his synonym: self-assembly also includes the ability to reach a configuration without a starting stable form or the ability to dynamically move portion of body1.1.2, which are not included in self-reconfiguration.
- **Self-repair**: "Self-repair robots are modular robots that have the capability of detecting and recovering from failures. It consists of detecting the failure of a module, ejecting the bad module and replacing it with one of the extra modules" [13], an ability that can be exploited by self-assembly, it is reasonable to think that if the system can build himself from a random initial configuration it can also build itself again changing one or more defective units (obviously replacements must be available).
- **Self-replicate**: "Fully autonomous machines that can construct functional copies of themselves from many very basic components" [20], a theoretical concept not necessary for self-assembly, but an ambitious idea that self-assembly could exploit to greatly enhance its capabilities in self-repair or scalability. A visionary example of a robot with appreciable ability of self-assembly and self-replication is "Gort" from "The Day The Earth Stood Still" [2].

In culmination of the comprehensive insights presented in Section 1.1, the concept of self-assembly can be refined and enriched as follows:

> "Self-assembly in robotics refers to the capacity of a random distributed swarm to spontaneously arrange himself into a coherent structure through self-organization, in pursuance of a determined task, encompassing properties such as replaceability, design freedom, and scalability. The implicit attainment of self-reconfiguration and self-repair is feasible. Additionally, while not intrinsic to the process, the strategic incorporation of self-replication can further enhance the system's capabilities."

1.2 History of self-assembly

At a high level of conceptualization, self-assembly is the ability to deal with all feasible tasks by the manner of the same units, an idea seen in films and cartoons as a representation of a revolutionary technology able to adapt to any situation and to overcome every problem. Despite that, about self-assembly not a lot can be found in literature, therefore in this section we would like to provide an overview of the most relevant projects, to clarify the current state of the art.

1.2.1 First step

In this section, we commence an exploration into the foundational steps of self-assembly, beginning with an examination of the seminal work that initiated theoretical discourse, followed by an in-depth analysis of the pioneering project marking the transition from theory to practical implementation.

CEBOT

Cebot [14] (Cell Structured Robot) is a distributed robotic system consisting of separable autonomous units, called cells. *Cebot* represents the first robot designed for **Dynamically Reconfigurable Robotic System (DRRS)**, a new kind of robotic system which is able to reconfigure itself to optimal structure depending on purpose and environment. Namely, self-assembly.
The functional cells are able to communicate with each other, approach, connect and separate automatically. If single cells of CEBOT are damaged, they can be repaired or replaced automatically to maintain system function. This makes *CEBOT* a self repairing and fault-tolerant robot system, *CEBOT* is capable to adapt itself to changing environments, it is a flexible system applicable in space, factory, hostile environments and other conditions.
In the paper [14] is proposed the approaching, connecting and separating method (**rendezvous-docking method**) and optimal structure decision method which can determine cell type, arrangement, degree of freedom and link length. Also, experiments proposed shows that the automatic approach: connection and separation, can be done successfully.

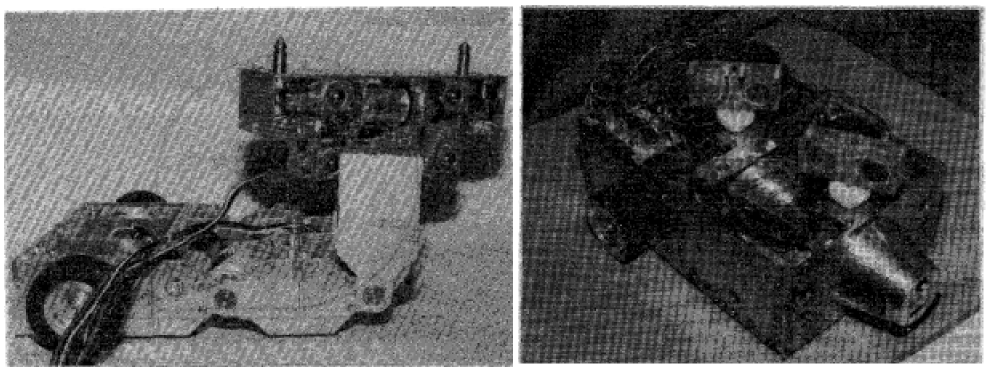


Figure 1.1: Cebot **I** version and two Cebot **II** version attached

In Figure 1.1 respectively *Cebot* version **I** and **II**. Version **II** has three infrared photodiodes, three ultrasonic sensors (one transmitter and two receivers), eight strong focused photodiodes in eight different directions in 45 degree intervals.
Experiments with those robots were conducted in a flat area without obstacles, the task was to assemble into a coherent structure, results are described in paper [14].

First project

It is important to notice that we refer to one of the first experiments to actually provide a practical way to implement self-assembly in real life, through simulations of robots which could effectively being built in mass, under the proper name of self-assembly. The project, described in "Self-Assembly and Self-Repair Method for a Distributed Mechanical System"[32] published in 1999, simulate a method for self-assembly through the usage of a homogeneous group of units and explore the possibilities of self-repair.

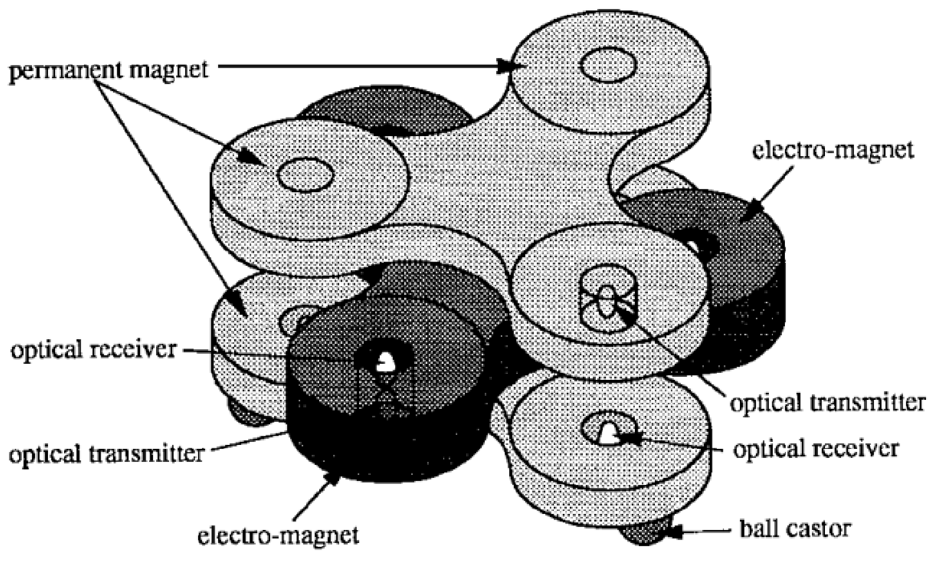


Figure 1.2: Unit for self-assembly and self-repair

The unit is composed of six arms as shown in Figure 1.2, each containing a pair of optical receiver/transmitter to communicate with other attached units; the three arms in the middle (darker) has electromagnet, with which they can decide to ensemble or disassemble from other units. Units can only move by the mean of repulsive and attractive forces of magnets, forcing at least an arbitrary shape as a starting position. The article propose two different ways to develop self-assembly:

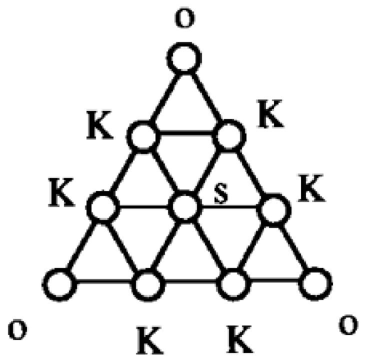


Figure 1.3: Project globally shared between units

The first is based on a common knowledge of the final form to be achieved, as we can see in Figure 1.3 every unit gain his logical type by mean of interactions with other attached units, if his logical type is in accordance with neighbor types nothing happen, in other case the unit takes a random moving action (if possible). In a simulation, they claim to have reached a successful rate of 97% assembling a triangle made of 10 units (failure includes two cases: the terminated assembly of a different shape and a nonterminating assembly activity in a fixed time), but only a successful rate of 73% in the case of a triangle composed of 15 units, evidently making this method non-scalable.

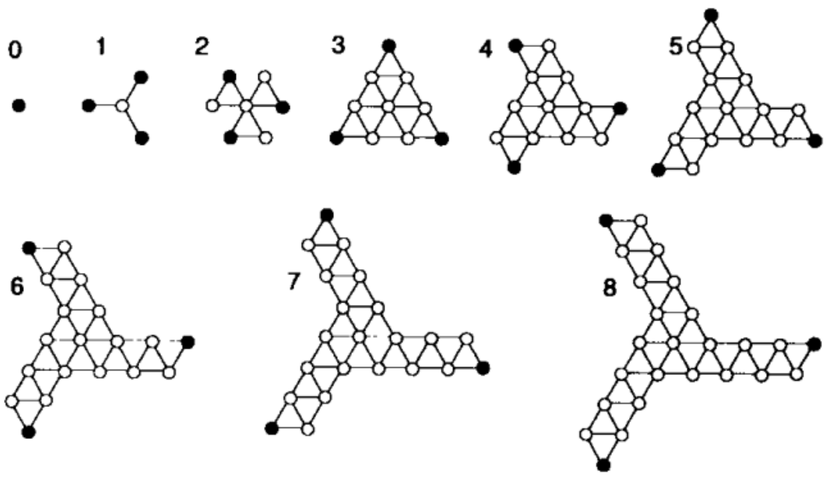


Figure 1.4: Stages of kernel layer formation

The second method is quite complex, for the ease of the reader we summarize the conceptual outline of the method as follows. Starting from an arbitrary connected shape, a unit is chosen, being the kernel and the starting point of the shape, now his neighbor are adjusted as expected from the project, when the kernel is completely covered by correct placed units, the process repeat satisfying new neighbors until completion (it builds the shape following the same concept of a BFS) as shown in Figure 1.4. A simulation with 75 units succeeded in 979 cases out of 1000, the required steps ranged from 1702 to 4775 and the average was 2600. This last method is certainly more complicated, but shows promising and interesting results.

1.2.2 Modular Robots

Starting from 2002, when "Modular Robots"[41] was published, researchers studied self-assembly under the name of modular robots, focusing on the ability of a swarm of robots to adapt to outdoor ambient. Thanks to their ability to change form, they could adapt to constantly varying tasks and environments without any external support, being really useful, for example, in remote exploration of hostile environments (E.g., under the sea, at the scene of a natural disaster, on other planets, ..), where you can't know what will be needed to proceed in advance. Follows a short list of the most interesting modular robots projects selected from "Modular Self-Reconfigurable Robot Systems"[40] published in 2007:

PolyBot

PolyBot from "PolyBot: a modular reconfigurable robot"[39] published in 2000, is a modular reconfigurable robot system composed of two types of modules, one called a segment and one called a node: the segment module has 1 DOF and 2 connection ports, the node module is rigid with no internal DOF and 6 connection ports. Both generation 1 (G1) and generation 2 (G2) of Polybot aim to achieve the versatility desired from self-reconfiguration systems, G1 and G2 shares the same structure while G2 is a superset of G1 with enhanced capabilities. The structure of G2 is made of laser-cut stainless steel sheet and is basically cube shaped, weighing 416g. Two opposing faces of the cube have connection plates. The module's one DOF allows these two faces to be rotated so they are no longer parallel, they can be rotated up to +90 or -90 degrees and allows two connection plates to mate in 90 degree increments.
For the programming part of Polybot, gait control tables with a set of finite possible configurations were used, this method has been tested over an obstacle course while under semi-teleoperated control, showing that a finite set of

movements is sufficient to achieve tasks of many applications.
Finally, the Polybot G3 is again an enhancement of the G2, this time with a major improvement brought by changing the central DC motor which was deforming the shape of the robot, as can be seen in Figure 1.5, with a DC pancake motor completely contained inside the robot.

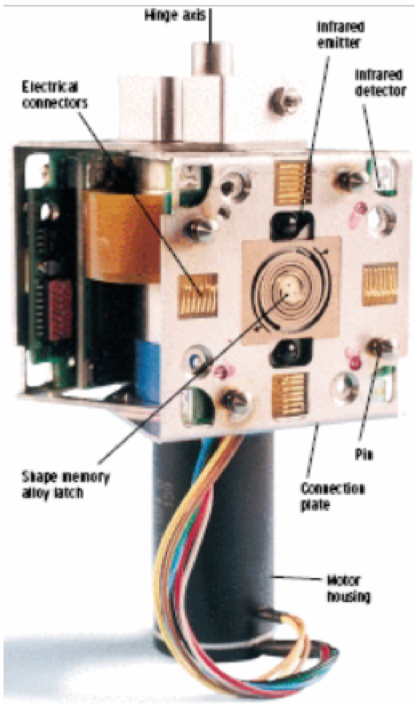


Figure 1.5: G2 Polybot unit

Programmable Parts

A programmable part described in “Programmable Parts”[9] and shown in Figure 1.6 consists of an equilateral triangular chassis that supports three controllable latching mechanisms, three IR transceivers, and a PIC18F242 based control circuit mounted on a custom-made PC board. Each edge of the chassis is 12 cm long and the chassis is 1.2 cm high (although the motors add another 3 cm to the height of the part). Each latch consists of three NeFeB permanent magnets: one fixed and the other two mounted on the end of a small geared DC motor. The position of the movable magnet is determined using Hall Effect sensors and mechanical switches. If they mutually decide to remain bound to each other, they do nothing. If at any point they mutually decide to detach from each other, each temporarily rotates its movable magnet 180º, forcing the parts apart. In the cited paper, is also introduced an experimental self-organizing robotic system and showed how *graph grammars* can be used to direct its self-organization.

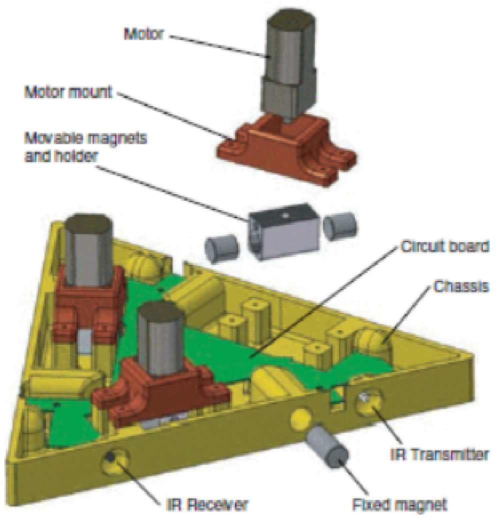


Figure 1.6: Programmable part design

The project described in “Programmable Self-Assembly”[19] published in 2007 makes use of *Programmable Parts* to conduct self-assembly experiments guided by a graph grammar. A set of rules is distributed among all the units, those describe the behavior related to attach and detach, for example, “two particles labeled a and that are not attached to each other can attach to each other, in which case they change their labels to b”. The experiment described leaded to the formation of a hexagon starting from random positions of units (as shown in Figure 1.7), in the paper is shown the proof that this method can be used for self-assembly programming and also permit post programming optimization through parameters tuning.

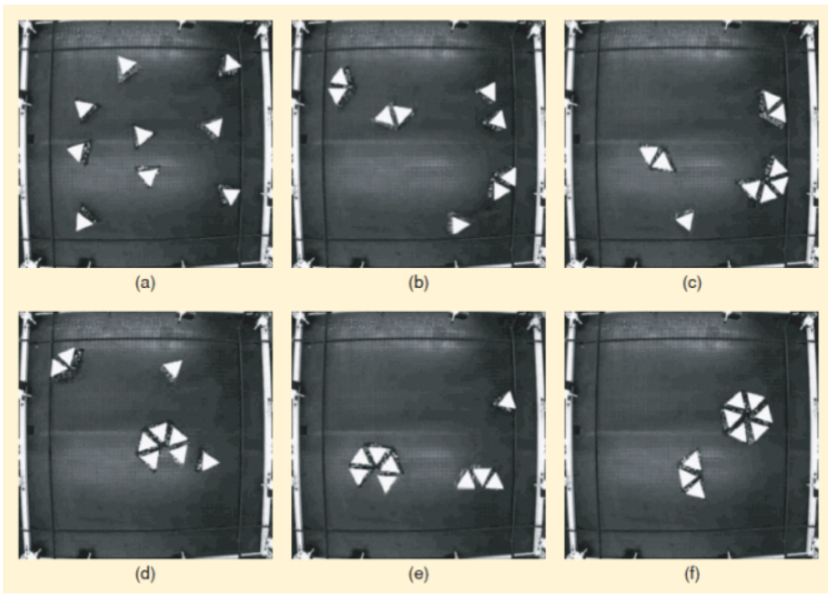


Figure 1.7: Programmable parts self-assembly time frames

SuperBot and Miche

SuperBot[29] in 2006 and *Miche*[15] in 2006 follows the same idea of *Polybot*[39] during the design of units' shape, maintains the cubic form but change the way they actually function.
SuperBot (Figure 1.8A) has being designed for NASA space exploration programs, the idea of realization is quite similar to *Polybot*[39], but it has three degrees of freedom with enhanced capabilities of torques. A network of *SuperBot* can share electricity power through its connections and communicate through high-speed infra-red LEDs.
Three different approaches were used to program *SuperBot*: (i) hormone inspired distributed control, (ii) table based control for fast prototyping, and (iii) phase automata for coordinating module activities. Experiments done showed a peculiar difference from single to dual module gaits, single gait favors actions of a single *SuperBot*, moving around, flipping and changing direction. Dual gaits enhanced the capabilities to synchronize movements between units using communications.

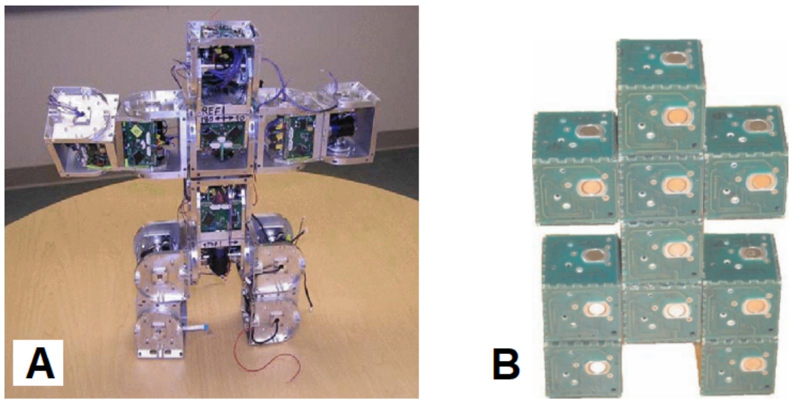


Figure 1.8: (A) SuperBot and (B) Miche

Miche (Figure 1.8B) is a modular lattice system capable of arbitrary shape formation. Each module is an autonomous robot cube capable of connecting to and communicating with its immediate neighbors. The connection mechanism is provided by switchable magnets. The modules use face-to-face communication implemented with an infrared system to detect the presence of neighbors. The group of modules collectively decides who is and is not on the final shape using algorithms that minimize the information transmission and storage. Finally, the modules not in the structure let go and fall off under the control of an external force, in this case gravity. All the algorithms controlling these processes are distributed and are very efficient in their space and communication consumption.

Follows a complete list (Figure 1.9) of modular self-organizing robot projects from 1988 to 2006 [40]:

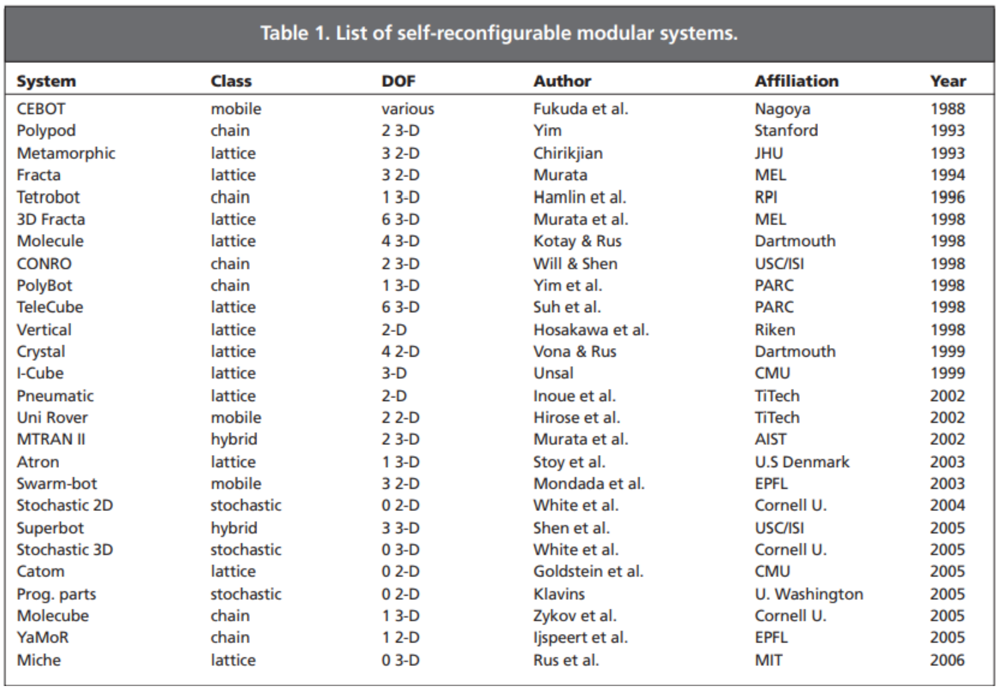


Table 1. List of self-reconfigurable modular systems.

System	Class	DOF	Author	Affiliation	Year
CEBOT	mobile	various	Fukuda et al.	Nagoya	1988
Polypod	chain	2 3-D	Yim	Stanford	1993
Metamorphic	lattice	3 2-D	Chirikjian	JHU	1993
Fracta	lattice	3 2-D	Murata	MEL	1994
Tetrobot	chain	1 3-D	Hamlin et al.	RPI	1996
3D Fracta	lattice	6 3-D	Murata et al.	MEL	1998
Molecule	lattice	4 3-D	Kotay & Rus	Dartmouth	1998
CONRO	chain	2 3-D	Will & Shen	USC/ISI	1998
PolyBot	chain	1 3-D	Yim et al.	PARC	1998
TeleCube	lattice	6 3-D	Suh et al.	PARC	1998
Vertical	lattice	2-D	Hosakawa et al.	Riken	1998
Crystal	lattice	4 2-D	Vona & Rus	Dartmouth	1999
I-Cube	lattice	3-D	Unsal	CMU	1999
Pneumatic	lattice	2-D	Inoue et al.	TiTech	2002
Uni Rover	mobile	2 2-D	Hirose et al.	TiTech	2002
MTRAN II	hybrid	2 3-D	Murata et al.	AIST	2002
Atron	lattice	1 3-D	Stoy et al.	U.S Denmark	2003
Swarm-bot	mobile	3 2-D	Mondada et al.	EPFL	2003
Stochastic 2D	stochastic	0 2-D	White et al.	Cornell U.	2004
Superbot	hybrid	3 3-D	Shen et al.	USC/ISI	2005
Stochastic 3D	stochastic	0 3-D	White et al.	Cornell U.	2005
Catom	lattice	0 2-D	Goldstein et al.	CMU	2005
Prog. parts	stochastic	0 2-D	Klavins	U. Washington	2005
Molecube	chain	1 3-D	Zykov et al.	Cornell U.	2005
YaMoR	chain	1 2-D	Ijspeert et al.	EPFL	2005
Miche	lattice	0 3-D	Rus et al.	MIT	2006

Figure 1.9: Modular self-organizing projects

1.2.3 S-Bots and Sambot

Starting from 2005 with *S-Bots* [11] and later in 2010 with *Sambot* [35] researchers moved their effort into find innovative and optimized ways to program self-assembly into distributed swarms composed of those two robots (separately), instead of trying to redesign a whole new robot suitable for the job. The two robots are first introduced, then a list of the following experiments is provided.

S-Bots

S-Bots are fully autonomous mobile robots lonely able to accomplish simple task such as navigation, grasping objects and perceiving environment.

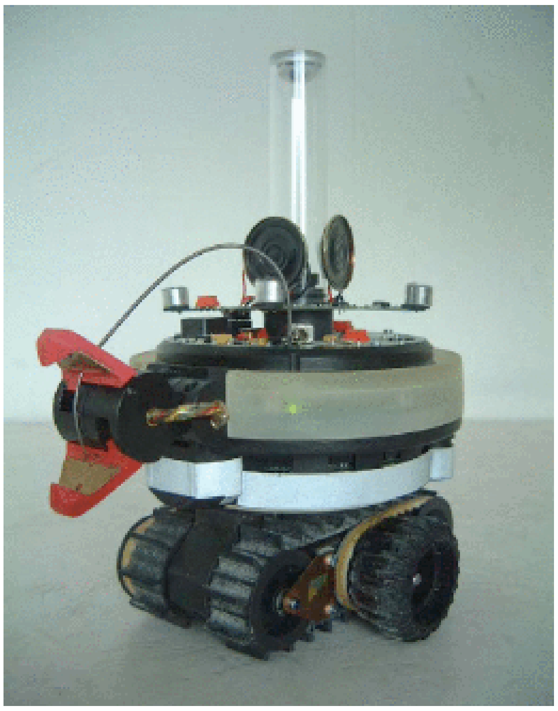


Figure 1.10: A photo of an *s-bot*

As we can see in Figure 1.10, *s-bots* are round robots with tracks and wheels used for navigation, a gripper on the front to grasp objects or other *s-bot*, multiple sensors to perceive themselves and the environment within they are, communication devices to detect and communicate with other *s-bots*, such as an omnidirectional camera, colored LEDs around the robot's turret, and sound emitters and receivers.
In order to be able to make experiments, a 3D dynamics simulator called **Swarm-bot3d**[2] was implemented. The simulator provides *s-bot* models with the functionalities available on the real *s-bots*. **Swarm-bot3d** is able to simulate realistic dynamics and collisions of rigid bodies in 3 dimensions, with a four level differentiation of details: the less detailed aimed to speed up training processes, the most detailed aimed to validate resulting controllers.

Sambot

Sambot is a completely autonomous robot designed to satisfy requirements such as *autonomous mobility*, *self-assembly* (with an active docking mechanism, so that it can realize autonomous connection and separation), *locomotion*

[2]`https://www.swarm-bots.org/index.php@main=3&sub=33.html`

(the formed robotic structure should be able to move) and *self-reconfiguration* (one robotic structure can transform into another).

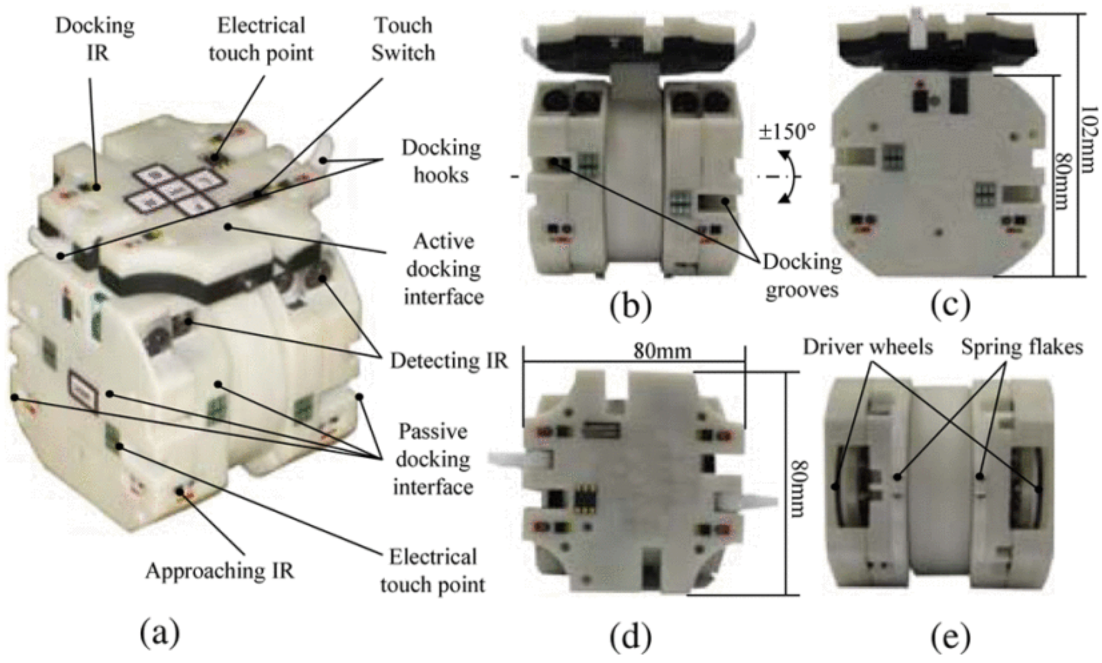


Figure 1.11: Technical details of a *sambot*

As we can see in Figure 1.11, *sambot* is a complex box with wheels to navigate and a mobile active docking station to attach to other *sambots*. On the active docking interface, there is a pair of active docking hooks, which can dock with any pair of docking grooves in the front, back, left, or right passive docking interface on the autonomous mobile body of another *Sambot*. The interface of the active docking robot rotates 90° forward or backward. The docking touch switch and docking infrared sensors on the active docking interface are used to guide the docking and judge whether the two *Sambots* are in the state of docking.
Thanks to his structure, *sambot* can be considered a fully autonomous mobile robot. But once the robotic structure is composed, it can have excellent abilities of locomotion and reconfiguration.

Projects

It is interesting to have a complete vision on *self-assembly* tasks that has been tried to be solved, and with which methodology. It is the objective of the next table, to show a summarized and comprehensive list of projects which brought a contribution for *self-assembly* within the usage of *s-bots* and *sambot*.

Year	Robot	Task	Method	Cite
2005	s-bot	heavy object transportation	sub-task ad hoc or neural	[11]
2006	s-bot	flat and rough terrain attaches	swarm int. and evolutionary	[16]
2006	s-bot	cooperative hole avoidance	artificial evolution	[33]
2007	s-bot	hill crossing	finite state machine	[28]
2007	s-bot	morphology creation	growing from seed	[27]
2010	sambot	snake-like and quadruped configuration	graph theory behavior based	[36]
2011	sambot	self docking and locomotion	behavior motor schema	[35]
2013	sambot	exploration and locomotion	behavior based	[21]
2014	sambot	cross formation	timed automata	[34]
2015	sambot	pass over a barrier	evolutionary config and controller	[22]

1.2.4 Latest advancements

Once again, maybe driven by latest technologies improvements, researchers in self-assembly moved their effort into the creation of new or better robots, follows the description of latest works and results.

Sambot II

Sambot II is a new self-assembly robot, actually an improvement of the older version *Sambot I*, as we can see in Figure 1.12, *Sambot II* has a couple of new sensors and LEDs. The LED-camera is a customized HD CMOS, it can sense images and colors from real world.

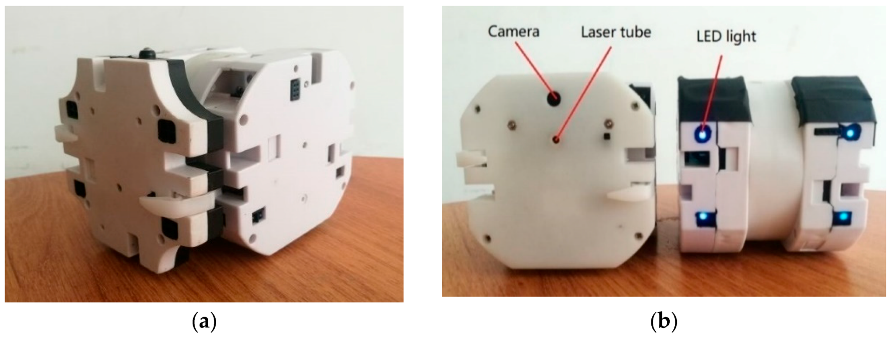


Figure 1.12: Sambot II new structure and sensors

The Laser-camera unit consists of a laser tube and a camera, placed like described in Figure 1.13, with a slight angle α and β pointing to the middle. Knowing those parameters, thanks to the position coordinates of the laser in the image, it is possible to mathematically estimate the distance between the laser-camera and the wall the laser hits.

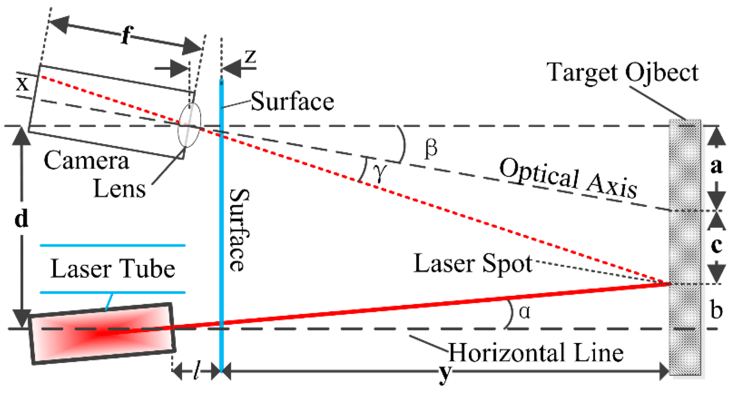


Figure 1.13: Sambot II laser-camera details

Those two new sensors, combined with LEDs on the passive docking surface of *sambot*, increased the efficiency and accuracy of the self-docking system. Those sensors have a different load of data to be elaborated, to make it possible to be used, the older *stm-32* processor has been replaced by an *x86 dual-core* CPU. The new processor brings with him new capabilities of image elaboration and path finding, two keys which effectively improved the new *sambot II* abilities, up to 80% of success in docking [31].

FireAntV3

FireAntV3 [30] uses a refined version of the *3D Continuous Docks* to attach to other such docks at any location at any orientation with simple control and without alignment. The robot can sense forces, the direction of a light source, and contacting neighbors using vibrations.

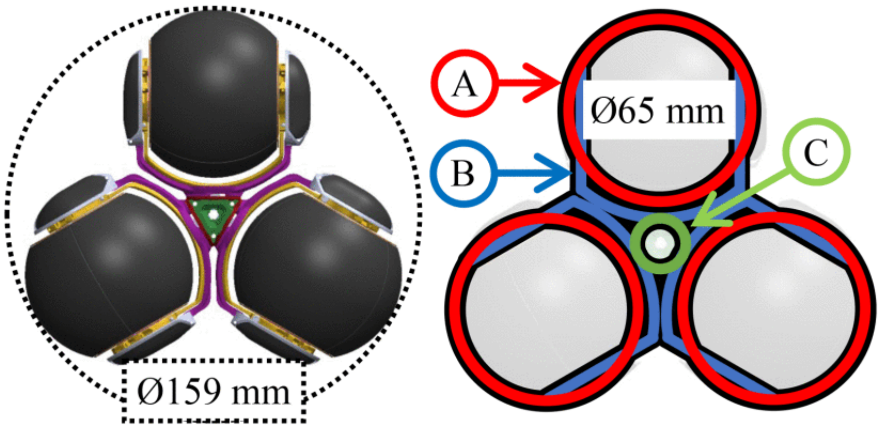


Figure 1.14: FireAntV3 design

As seen in Figure 1.14, *FireAntV3* consists of three spheres mounted on forks and joined by a center body. These spheres contain the majority of the system electronics and sensors and are coated in a continuous docking surface that allows peer robots to form strong attachments to each other regardless of the location or orientation of contact. Placing the spheres closely together helps ensure that any approach by a like robot will result in dock-to-dock contact, though this greatly constrains the size of the center body.

Experiments with *FireAntV3* demonstrated his functioning, the first experiment, phototaxis, was made in an arena as shown in Figure 1.15, the robot starting in the opposite side of the light, was able to reach the light in 5 steps (images are ordered steps by number). The robot was randomly placed on the opposite side, with no effort into placing him in the correct way or the correct angle.

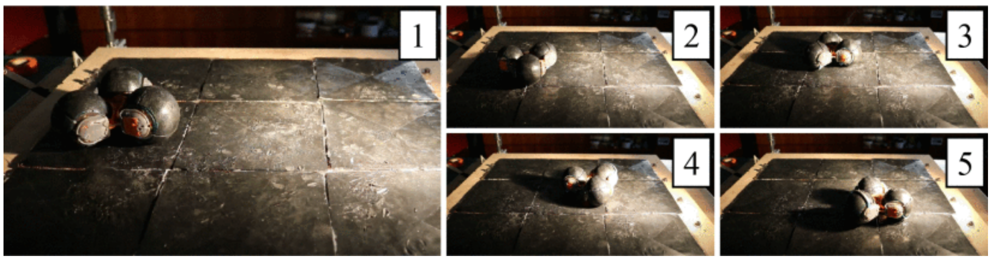


Figure 1.15: FireAntV3 phototaxis experiment

The next experiment is *Step-on, step-off*, with this experiment authors meant to demonstrate effectiveness of communication with vibrations and continuous docking surface. The experiment is composed of an arena, a complete

FireAntV3 robot and fixed sphere component, the task for the complete robot is to, starting from the floor, step on the fixed sphere object and then step off on the other side. The robot was able to accomplish the task in steps as described in Figure 1.16.

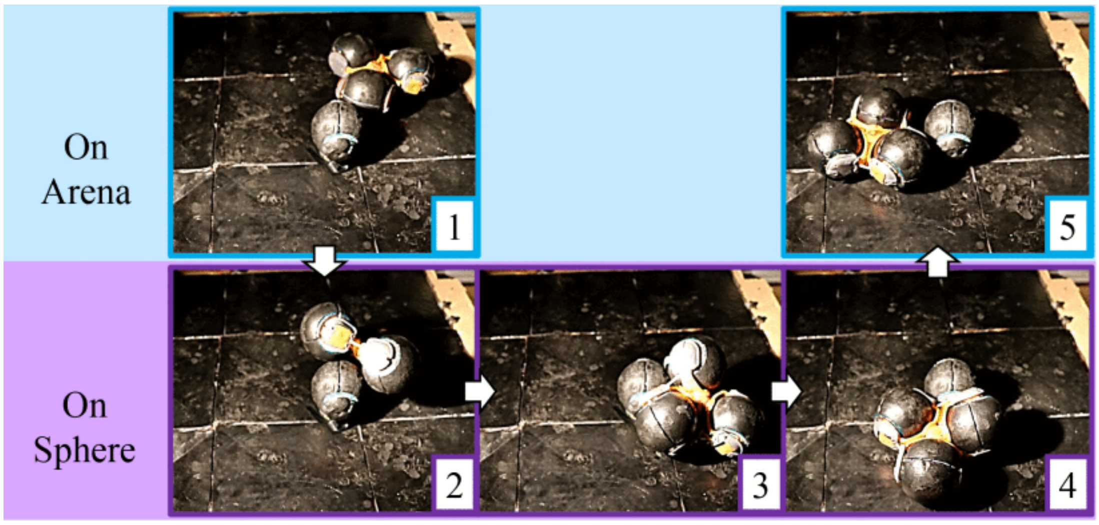


Figure 1.16: FireAntV3 Step-On Step-Off experiment

Chapter 2

Technologies

In this chapter, we describe **2D-Robot-Evolution** [23], a Java framework for experimenting with the evolutionary optimization of 2D simulated robotic agents. We also introduce the data describer, later used to verify and visualize results from our experiments.

2.1 2D-Robot-Evolution

2D-Robot-Evolution is a java framework which merges *voxel-based soft robots* (VSRs) simulation capabilities of **2dmrsim**2.1.1 with evolutionary computations capabilities of **jgea**2.1.2. *2D-Robot-Evolution* makes usage of **jnb**2.1.3, a Java library for building instances of classes given textual descriptions, to describe experiments to be executed (properly formatted), avoiding the necessity to recompile sources when the experiment is changed.

2.1.1 2-D Multi Robot Simulator

2D-Multi Robot Simulator (2dmrsim) is a Java framework capable of VSRs simulations, it is a newer version of *2D-VSR-Sim* (2dhmsr) [24][25] capable of simulate more kinds of robots and decoupled from the inner physics simulator. *2D-VSR-Sim* is a simulator of one or more 2-D **VSR**s that perform a task, i.e., some activity whose degree of accomplishment can be evaluated quantitatively according to one or more indexes. The simulation is discrete in time, using a fixed time step, and continuous in space: the position and configuration of each *voxel* of the **VSR** is updated at each time-step according to the mechanical model and to the **VSR** controller.

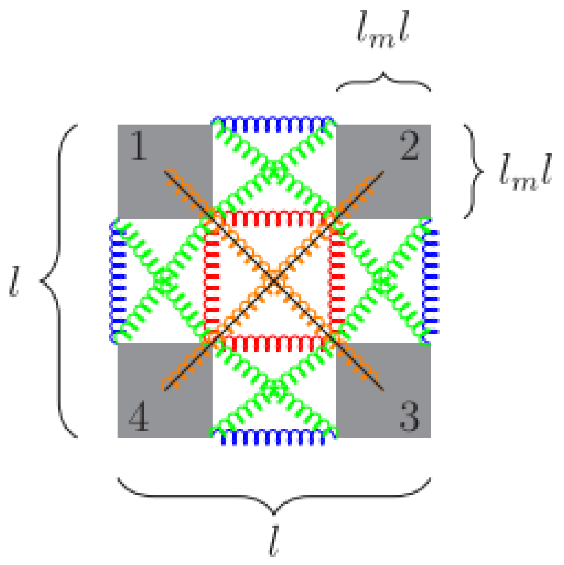


Figure 2.1: **VSR** inner composition

In *2D-VSR-Sim* a *voxel* is a soft 2-D block, i.e., a deformable square modeled with four rigid bodies (square masses), a number of spring–damper systems (SDSs) that constitute a scaffolding, and a number of ropes. SDSs and ropes have zero mass; ropes act as upper bounds to the distance that two bodies can have. Figure 2.1 shows the mechanical model of a single voxel: the four masses are depicted in gray, the different components of the scaffolding are depicted in blue, green, red, and orange, the ropes are depicted in black. Every parameter of the *voxel* can be configured: masses, dimensions, friction, and other aspects are adjustable. The scaffolding is also configurable, selecting a subset of springs groups (the ones being colored in Figure 2.1).
The way a **VSR** behaves is determined by a controller. Whenever it is invoked, the controller determines, for each *voxel vi* of the **VSR**, the control value $fi \in [-1, 1]$ to apply. The control value is applied by the physics engine and results in a change in the area of the corresponding voxel and hence a change in the shape of the VSR. The control value can be calculated in any way, can be a constant, can be time-dependent, usually is the result of the combination of sensors readings.

2.1.2 Java General Evolutionary Algorithm

Java General Evolutionary Algorithm (jgea) [26] is a modular Java framework for experimenting with *Evolutionary Computation*, designed to be aimed at providing a general interface to potentially all *Evolutionary Algorithms*,

solid and easy to use for people who rely on *Evolutionary Computation* as a tool.
Given a *problem*, an *Evolutionary Algorithm* act as a *solver* for it, with the goal to find one or more solution to the *problem*. In **jgea** those concepts are modelled with interfaces, follows details on structure and components.

Problem

Any problem can be described by a class implementing the *Problem* interface, that defines the solution space S and a way of comparing two solutions, by extending the `PartialComparator<S>`.

```
public interface Problem<S> extends PartialComparator<S> {}
```

When solutions are compared according to their quality (fitness value), a `QualityBasedProblem<S, Q> extends Problem<S>` is used, this interface adds two functionalities, one to obtain the quality value from a solution, the second to compare those qualities. Here, `Q` represents the quality- or fitness-space.
When we want to enforce a total ordering between qualities of solutions, a `TotalOrderQualityBasedProblem<S, Q> extends QualityBasedProblem<S, Q>` interface is used, adds a `TotalOrderComparator()` and gives a default implementation for qualities comparator.
When the quality of the solution is already comparable (i.e., `Q extends Comparable`) the following interface is used:

```
ComparableQualityBasedProblem<S, Q extends Comparable<Q>> extends
    TotalOrderQualityBasedProblem<S, Q>
```

To model more specific classes of problems, it is sufficient to add interfaces extending or classes implementing *Problem* interface.

Solver

A problem can be solved by an implementation of the Solver interface, which is responsible for providing the caller with a collection of solutions upon the invocation of its method *solve()*.

```
public interface Solver<P extends Problem<S>, S> {
    Collection<S> solve(
        P problem,
        RandomGenerator random,
        ExecutorService executor
    ) throws SolverException;
}
```

The generic parameter `P` indicate the subset of problems a *Solver* can tackle. *solve()* takes two additional elements: a *RandomGenerator* and an *ExecutorService*. The contract for *solve()* states that every random number has to be generated through *RandomGenerator*, thanks to this constraint *repeatability* is granted. However, *reproducibility* cannot be granted, due to concurrency. Similarly, the contract for *solve()* states that the *ExecutorService* instance will be used for distributing computation across different workers of the executor, easily done due to the nature of the problems to solve, addressing also on efficiency.
Jgea provides some prominent Evolutionary Algorithms (Grammatical Evolution, Hierarchical Grammatical Evolution, Weighted Hierarchical Grammatical Evolution, Context-free Grammar Genetic Programming), Evolutionary Strategies, OpenAI ES, CMA-ES, Map Elites, Speciation Evolution, Diversity Driven Grammar-guided Genetic Programming, Differential Evolution, and NSGA-II implementations.

Individual

The notion of *individual* is used, modeled in the `Individual` record, to capture the genotype-phenotype representation. Two generics parameters, `G` and `S`, define the genotype and the phenotype spaces, respectively. A generic parameter `Q` to store the quality (or fitness) of the solution.

```java
public record Individual<G, S, Q>(
    G genotype,
    S solution,
    Q fitness,
    long fitnessMappingIteration,
    long genotypeBirthIteration
)
```

To create an instance of `Individual`, we need to (i) obtain a genotype, (ii) map it to the corresponding phenotype, and (iii) evaluate the fitness of the candidate solution. Note that an Individual also stores the iteration at which the fitness is evaluated (*fitnessMappingIteration*), and the iteration at which the genotype is obtained (*genotypeBirthIteration*): these values model the "evolutionary age" for the individual in the evolutionary optimization run it belongs to. A genotype can either be created from scratch, or it can be the result of the application of genetic operators on pre-existing genotypes.
Several Evolution Algorithms require selecting individuals for reproduction or survival. The `Selector` interface is used to model the selection process.

```java
public interface Selector<T> {
    <K extends T> K select(
        PartiallyOrderedCollection<K> ks,
        RandomGenerator random
    );
}
```

A few concrete selector implementations are provided, such as the `Tournament`, replicating the tournament selection, or the `First` and `Last`, returning the best or worst individual (or a random one among them, in case of fitness ties).

Listener

Even though problems could in principle be solved in-the-void, it is often necessary to track the execution of the solver, extracting and saving information during the run. To this extent, the `Listener` interface is provided.

```java
public interface Listener<E> {
    void listen(E e);
    default void done() {}
}
```

A `Listener` has the duty to monitor, i.e., `listen()` to, the updates of the state during the execution of the *solve()* method. The monitoring of the execution, either parallel or sequential, of multiple instances of `Solver` solving multiple instances of `Problem`, distinguishes individual executions while saving or printing all information on the same target (e.g., the same CSV file for all the evolutionary runs).
Concerning the information to be extracted from the state, one might be interested in the size of the population, the quality of the best individual, some function of the best individual, and so on. To allow the users to easily define the information they want to extract, and associate a name, and possibly a display format, to it, the `NamedFunction` interface is provided. Typically, a List of `NamedFunctions` is passed to the constructor of a `ListenerFactory`, and each of them is invoked on a state within the *listen()* method to extract the needed information.

2.1.3 Java Named Builder

Java Named Builder (jnb[1]) is a Java library for building instances of classes given textual descriptions properly formatted. The core concept is the one of

[1]`https://github.com/ericmedvet/jnb`

named builder, which can build instances of classes given a **named parameter map** (or named dictionary, using a different term). A named parameter map is simply a collection of (key, value) pairs with a name.
More specifically, **jnb** provides a few interfaces and classes for doing the following key things:

1. annotating an existing class or method to be used as a builder: the key artifacts for this are the annotations `@Param` and `@BuilderMethod`.
2. parsing a textual description into an object storing the information needed to invoke a builder: the key artifact here is the interface `NamedParamMap`.
3. building a builder automatically from annotated class: the key artifact here is the `NamedBuilder`.

You can annotate a method or a constructor (also of a *record*) to make it discoverable by the methods *fromClass()* and *fromUtilityClass()* of `NamedBuilder`.

```
public static Person young(@Param("name") String name, @Param(value
   = "age",dI = 43) int age) {
      return new Person(name, 18);
}
```

will result in a named builder where the name is young (possibly with a prefix, as in the previous example) and the expected parameters are name and, optionally (in the sense that there is a default value of 43), age.

A named parameter map is a map (or dictionary, in other terms) with a name. It can be described with a string adhering to the following human- and machine-readable format, described by the following grammar:

```
<npm>  ::= <n>(<nps>)
<nps>  ::= ∅ | <np> | <nps>;<np>
<np>   ::= <n>=<npm>|<n>=<d>|<n>=<s>|<n>=<lnpm>|<n>=<ld>|<n>=<ls>
<lnmp> ::= (<np>)*<lnpm>|<i>*[<npms>]|+[<npms>]+[<npms>]|[<npms>]
<ld>   ::= [<d>:<d>:<d>] | [<ds>]
<ls>   ::= [<ss>]
<npms> ::= ∅ | <npm> | <npms>;<npm>
<ds>   ::= ∅ | <d> | <ds>;<d>
<ss>   ::= ∅ | <s> | <ss>;<s>
```

Where:

- $< npm >$: is a named parameter map.
- $< n >$: is a name, i.e., a string in the format `[A-Za-z][.A-Za-z0-9_]*`.
- $< s >$: is a string in the format `([A-Za-z][A-Za-z0-9_]*)|("[^"]+")`.
- $< d >$ is a number in the format `-?[0-9]+(\.[0-9]+)?`.
- $< i >$ is a number in the format `[0-9]+`.
- $\varnothing$: is the empty string.

An example of a syntactically valid named parameter map is:

```
car(dealer = Ferrari; price = 45000)
```

where *dealer* and *price* are parameter names and *Ferrari* and *45000* are parameter values. *car* is the name of the map.
Two examples with the $*$ operator:

```
2 * [dog(name = simba); dog(name = gass)]
// Corresponds to:
[
  dog(name = simba);
  dog(name = gass);
  dog(name = simba);
  dog(name = gass)
]

(size = [m; s; xxs]) * [hoodie(color = red)]
// Corresponds to:
[
  hoodie(color = red; size = m);
  hoodie(color = red; size = s);
  hoodie(color = red; size = xxs)
]
```

An example of combined use of $*$ and $+$ is:

```
+ (size = [m; s; xxs]) * [hoodie(color = red)]
+ [hoodie(color = blue; size = m)]
// Corresponds to:
[
  hoodie(color = red; size = m);
  hoodie(color = red; size = s);
  hoodie(color = red; size = xxs);
  hoodie(color = blue; size = m)
]
```

2.2 Data description and visualization

Raw data extracted during experiments describes, for each seed and iteration, the best genotype in terms of fitness among the population being evaluated. Taking in consideration that the best genotype is not guaranteed to be preserved in the next generation, final fitness curves result in an oscillating curve as shown in Figure 2.2.

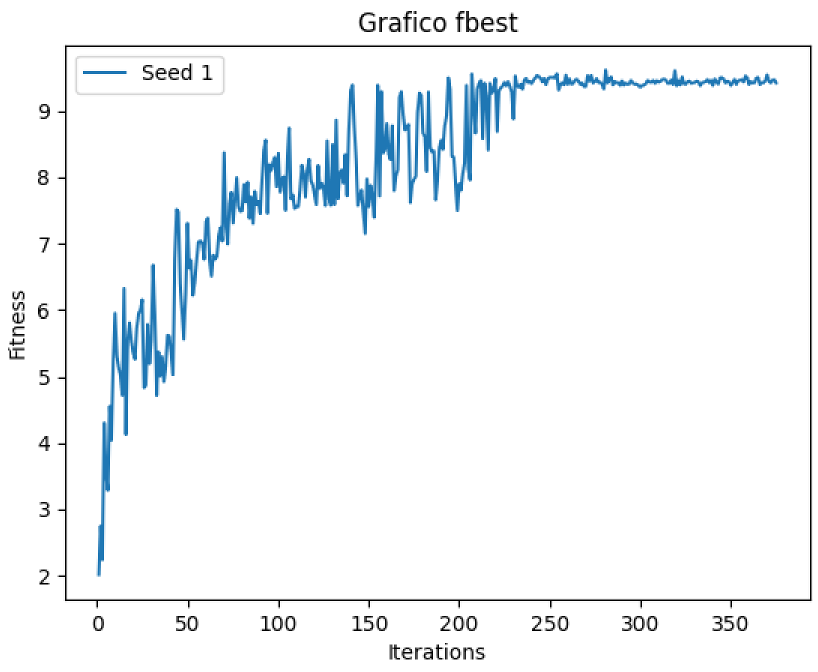


Figure 2.2: Example of oscillating fitness line

For statistical purposes, it was necessary to obtain a non-decreasing fitness line, which could be utilized to extract relevant data. To accomplish this objective, the **processor** script2.2.1 was developed. It takes raw data from experiments and produce processed data which maintain, for each iteration and seed, the best genotype with his details found until the current iteration. With processed data, it was possible to extract interesting statistic results, in detail we analyzed the **Run Length Distribution** (**RLD**) of fitness, to study the learning curve of experiments, and generated the **Violin plot** of fitness across iterations and seeds, to observe its distributions trends in different experiments. From processed data, to generate graphs described above, the **plotter** script2.2.2 was developed.

2.2.1 Processor script

The *processor* script takes in input the CSV file containing raw data and produces in output another CSV with processed data. In detail, the CSV generated has headers: *seed*, *iterations*, *evals*, *fitness*, and *genotype* as shown in Figure 2.4. Rows contain, for each iteration and seed, the best genotype details found until that iteration, ensuring for each seed a non-decreasing fitness line along all iterations, an example provided in Figure 2.3.

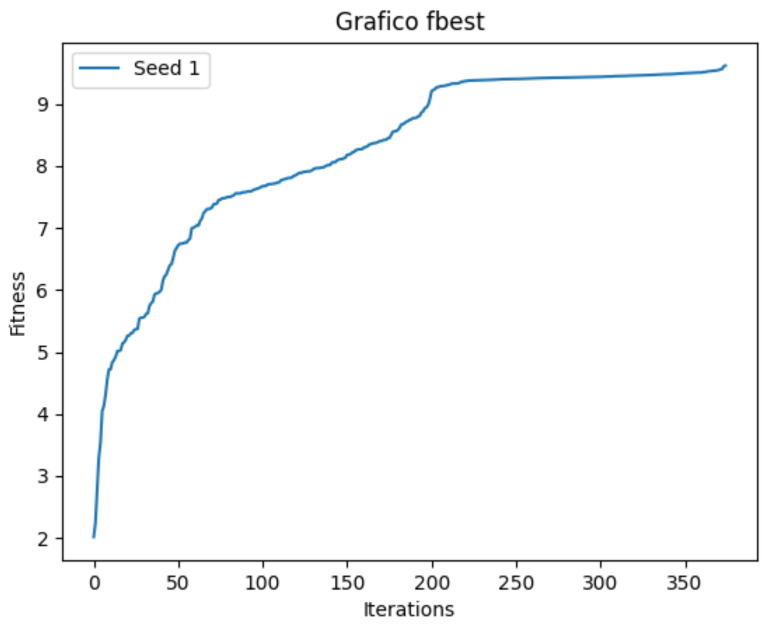
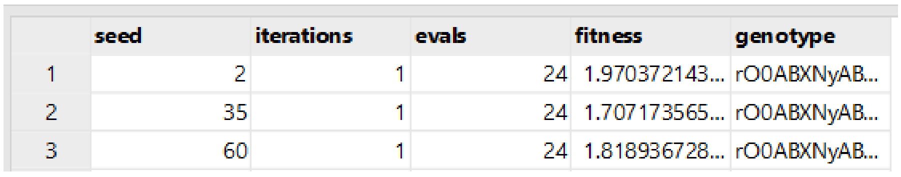


Figure 2.3: Non-decreasing fitness line



	seed	iterations	evals	fitness	genotype
1	2	1	24	1.970372143...	rO0ABXNyAB...
2	35	1	24	1.707173565...	rO0ABXNyAB...
3	60	1	24	1.818936728...	rO0ABXNyAB...

Figure 2.4: Processed data CSV file headers

The *processor* script with inline comments2.1 are self-explanatory, but it is interesting to notice that, despite iterations should be naturally ordered inside the file, ordering for iteration is forced to ensure the correct functioning of the script. Notice that, some redundant data could be left out (keeping only rows describing greater and not equal fitness than before), but for later usage, it was convenient to maintain it in the file. Follows the *processor* script definition.

```
import pandas as pd
import glob

# Searching for files to process.
files = glob.glob('dataM*e.csv')

for file in files:
    # Read the CSV file
    print(f'Reading {file}... ')
    df = pd.read_csv(file, delimiter=';')

    # Initialize lists to store processed data
    new_rows = []
    current_best_fitness = {}
    current_best_genotype = {}

    # Iterate through the rows sorted by iterarions
    print(f'Processing... ')
    for index, row in df.sort_values(by=['iterations']).iterrows():
        seed = row['seed']
        iterations = row['iterations']
        best_fitness = row['best?fitness?s.task.1.xDistance']
        best_genotype = row['best?genotype?base64']

        # Update current best fitness and genotype for the seed
        if (seed not in current_best_fitness) or (best_fitness >
            current_best_fitness[seed]):
            current_best_fitness[seed] = best_fitness
            current_best_genotype[seed] = best_genotype

        # Append the processed row to the new data
        new_rows.append([seed, iterations, row['evals'],
            current_best_fitness[seed], current_best_genotype[seed]])

    # Create a new DataFrame with processed data
    new_df = pd.DataFrame(new_rows, columns=['seed', 'iterations', 'evals',
        'fitness', 'genotype'])

    # Write the new DataFrame to a CSV file
    print(f'Saving processed CSV...')
    new_df.to_csv(file.replace('M', 'P'), sep=';', index=False)
```

Listing 2.1: process_data.py

2.2.2 Plotter script

The *plotter* script takes in input the processed CSV generated with *processor* script and produces *RLD* and *Violin* graphs. In detail, the *RLD* graph2.5 shows in percentage how many replicas had success until a certain iteration, it is easy to understand that if the best genotype is kept among iterations; that line can only be non-decreasing.

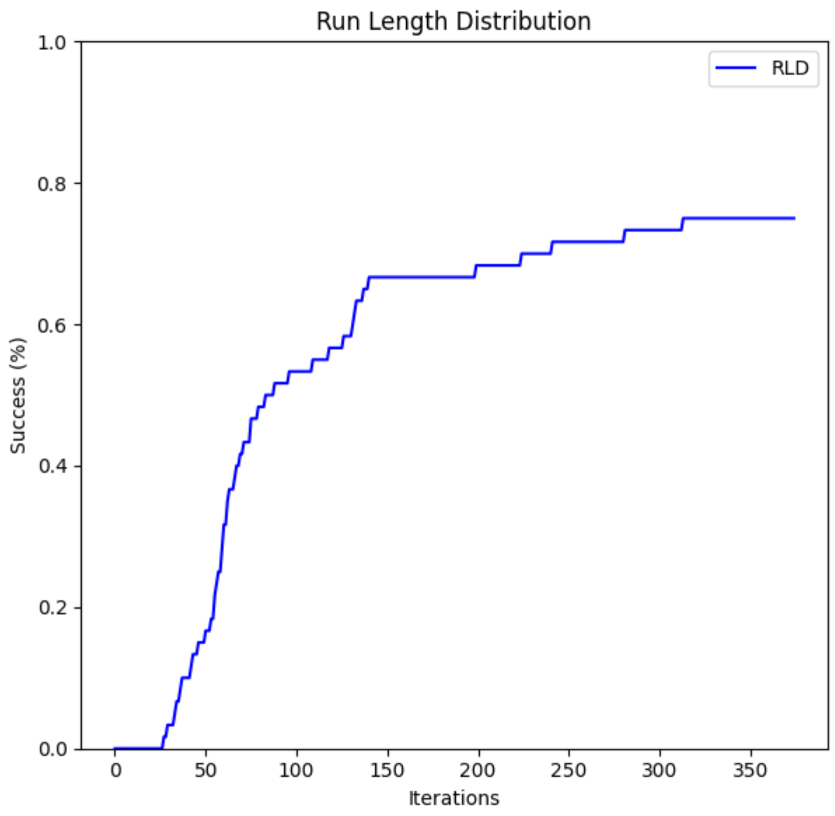


Figure 2.5: Example of an RLD graph

The *Violin* graph2.6 shows the distribution of fitness along all iterations and seeds, the graph is tighter or wider corresponding to minor or major fitness concentration. That graph is called "Violin" cause of the classic form it usually assumes.

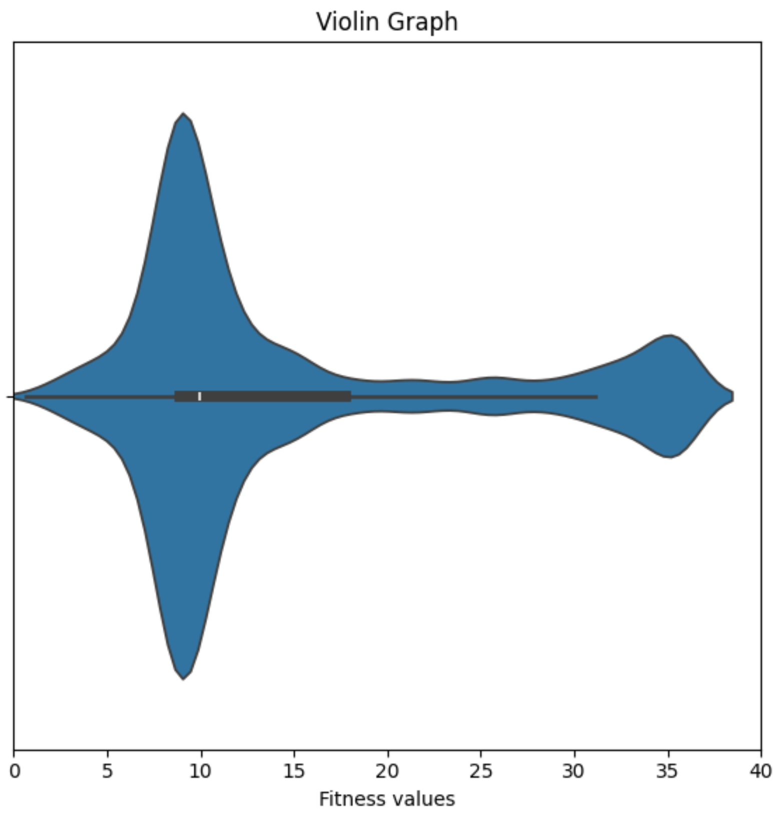


Figure 2.6: Example of a Violin plot

Once again, the *plotter* script with inline comments2.2 are self-explanatory, but it is interesting to notice the complexity of *RLD* calculation, which involves a **set**, that was developed before the realization of the *processor* script, and was a solution to avoid the necessity of processed data. Follows the *plotter* script definition.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from itertools import groupby
import glob

# Constants
MIN_FITNESS = 0
MAX_FITNESS = [40, 23, 13]
RLD_SUCCESS_THRESHOLD = 9

# Searching for files to show.
files = glob.glob('dataP*e.csv')

for idx, file in enumerate(files):
    # Read the CSV file
    print(f'Reading {file}... ')
    df = pd.read_csv(file, delimiter=';')

    # Group the data by iteration
    grouped = df.groupby('iterations')
    n_seeds = len(df.groupby('seed'))

    # Run Length Distribution (RLD) calculation
    print('Plotting ... ')
    rld = []
    rld_s = set()
    for iteration, group in grouped:
        rld_s.update(group.loc[df['fitness'] >= RLD_SUCCESS_THRESHOLD
            + idx]['seed'].values.tolist())
        rld.append(len(rld_s) / len(group))

    # Plot graphs
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))

    # Run Length Distribution (RLD)
    ax_rld = axes[0]
    ax_rld.plot(rld, label='RLD', color='blue')
    ax_rld.set_title(f'Run Length Distribution ({idx+1}+epsilon)')
    ax_rld.set_xlabel('Iterations')
    ax_rld.set_ylabel('Success (%)')
    ax_rld.set_ylim(0, 1)
    ax_rld.legend()
```

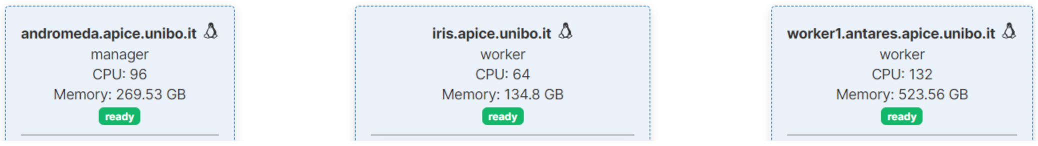
```
# Violin Graph
ax_violin = axes[1]
sns.violinplot(x=df['fitness'], ax=ax_violin)
ax_violin.axvline(x = RLD_SUCCESS_THRESHOLD+idx, color = 'r', label
    = f'Success line: {RLD_SUCCESS_THRESHOLD+idx}')
ax_violin.set_title(f'Violin Graph ({idx+1}+epsilon)')
ax_violin.set_xlabel('Fitness values')
ax_violin.set_ylabel('Fitness')
ax_violin.set_xlim(MIN_FITNESS, MAX_FITNESS[idx])
ax_violin.legend()

# Save the plots to files
print('Saving plotted figures ... ')
plt.tight_layout()
plt.savefig(file.replace('P', 'I').replace('.csv', '.png'))
plt.close()
```

Listing 2.2: plot_processed_data.py

2.3 Cluster configuration

The usage of population-based algorithms notoriously requires a lot of time and computation capabilities, besides the firstly experiments which ran on a simple portable pc, it was quite obvious from the beginning that a hardware with more capabilities was necessary to continue with longer and more complex experiments. A machine with strong **CPU** performance was needed, we so decided to rely on **Cluster 4.0** at the disposal of students and researchers of Unibo Cesena Multicampus. *Cluster 4.0* is a small cluster composed of three high-performance machines (Figure 2.7), for our experiments we were assigned to a machine with 64 physical cores and 134.8 GB of RAM (*iris.apice.unibo.it*). Thanks to the new hardware, we were able to scale up our experiments, incrementing complexity in terms of number of *voxel* or number of *run* parallelly running, which eventually lead us to the results we obtained3.2.2.


andromeda.apice.unibo.it
manager
CPU: 96
Memory: 269.53 GB
ready

iris.apice.unibo.it
worker
CPU: 64
Memory: 134.8 GB
ready

worker1.antares.apice.unibo.it
worker
CPU: 132
Memory: 523.56 GB
ready

Figure 2.7: Cluster 4.0 machines composition

2.3.1 Portainer

For the ease of use among multiple users, the cluster has been initialized and can be controlled through **Portainer.io**[2], a versatile container management software, which simplify and orchestrate operations like containers deployments, management, troubleshooting, and security. *Portainer.io* consists of two main parts: a *Server* and an *Agent*2.8. The *Portainer Agent* runs on the node on a cluster and communicates with the *Portainer Server*. The *Portainer Server* centrally manages up to 25,000 nodes it connects to.

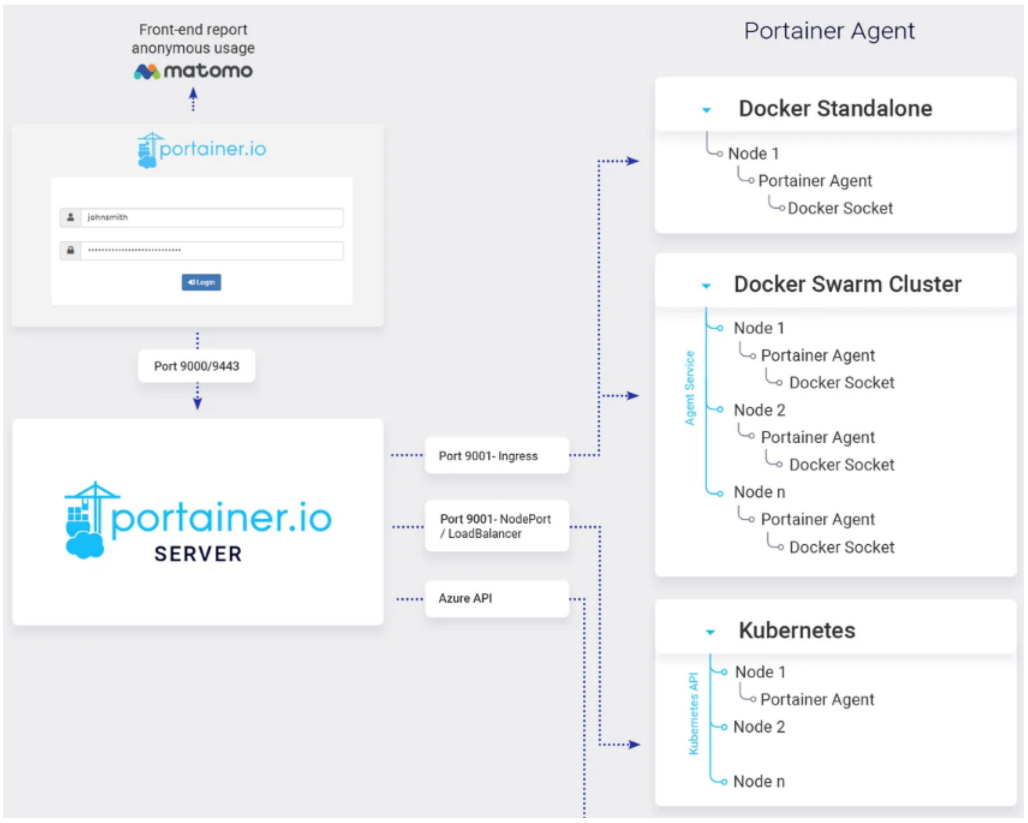


Figure 2.8: Portainer server-agent architecture

The best fit for our experiments, it's been the separation in two macro parts: the **Worker** and the **Publisher**. The *Publisher* component, designed as a stack swarm, is accessible externally through the link iris.apice.unibo.it, and contains a simple **HTTP server** created from the implementation given with python:

```
python -m SimpleHTTPServer 30022
```

[2]https://www.portainer.io/

SimpleHTTPServer creates a web server which, by default, serves the file system. In our case, this was useful to download in local experiments results, to elaborate and observe them, operations computationally feasible without the need of such a lot of CPU. It is important to know that the cluster is not publicly accessible, to be able to see and to use it, you must before gain the access to the **VPN** of the research department, then your credentials has to be activated by system administrators. So, the opening to the exterior of a simple HTTP server doesn't impact on security of the cluster.
The *Worker* is a docker container with a fresh *Ubuntu* installation, packets like `openjdk-17-jdk openjdk-17-jre maven tmux htop` are installed to run experiments and to monitor resources usage. This container born to run experiments, it is so configured to use all the CPU (64 cores) and 50 GB of RAM (experiments shown that was enough). *Worker* shares a volume where data files are saved with the *Publisher*, in this way they are always synced and results are observable without slowing down the *Worker*. Thanks to the usage of an external volume, persistency of data is also granted, meaning that even if the cluster get shutdown for any reason, results saved in the volume are not lost.

Chapter 3

Experiments

In this chapter we illustrate the entire work done in terms of preliminary tests3.1, firstly with **Genetic Evolutionary** algorithms and later with **Covariance Matrix Adaptation Evolution Strategy**, which guided us from the correct setup and execution of our research experiment3.2.1, until the collection and presentation of results3.2.2. There is also a section devoted to **Future Developments**3.3 created to guide paths that could be taken starting from this work.

3.1 Preliminary tests

Population-Based algorithms [7] offer great capabilities for optimization problems in which a proven optimal solution is not needed and for which exact algorithms might be excessively demanding in terms of computational resources. This property makes them suitable for our experiments, but, as for many other algorithms, they require component fine-grain design and parameter tuning. The choice of hyperparameters, the experiment setup and the definition of fitness have a great impact on efficacy and efficiency of algorithms.
Those statements, hence the importance of experience and the necessity of early tests, to properly set up a longer and exhaustive training experiment. In this section, we explain all the choices made, from first tests to the final experiment, remarking errors and successes which guided us to the final setup and results.

3.1.1 Genetic evolutionary algorithm

We started our tests by reproducing a working and verified experiment provided with **2D-Robot-Evolution** [23] described in the dedicated **GitHub**[1] repository. The example provided aim to train a *biped*-**VSR**, showed in Figure 3.1, onto right walking in a flat terrain, later observing through created video, his behavior in a hilly terrain.

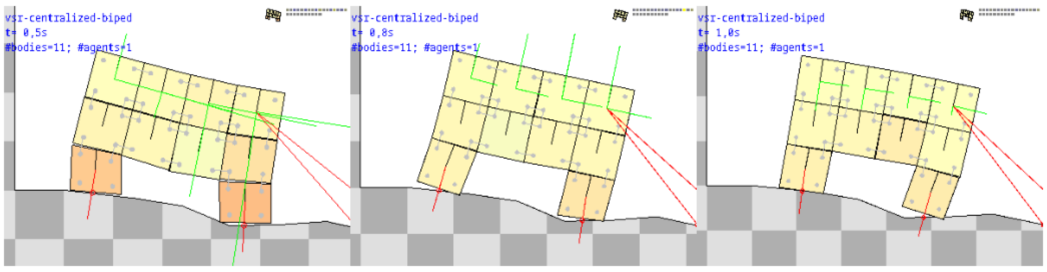
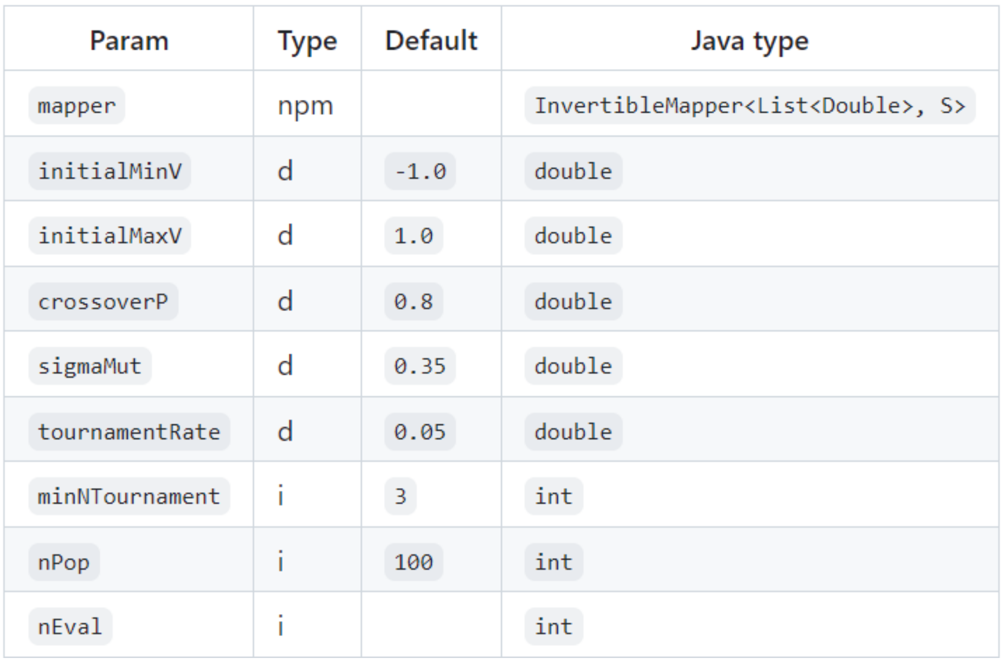


Figure 3.1: example biped VSR

The example uses an implementation of a Genetic Algorithm based on the search of correct weights for multiple **MLP** networks, each assigned to a specific *voxel* as his controller, forming a heterogeneous distributed controller. Each **MLP** takes in input data provided by sensors "placed" on his *voxel*, in this case a total of six sensors, then those signals are processed through a single hidden layer toward the output layer, which gives values to be used by actuators of the *voxel*, which can be both forces on *springs* or values to be transmitted to near attached *voxel*s.

[1]`https://github.com/ericmedvet/2d-robot-evolution`



Param	Type	Default	Java type
`mapper`	npm		`InvertibleMapper<List<Double>, S>`
`initialMinV`	d	`-1.0`	`double`
`initialMaxV`	d	`1.0`	`double`
`crossoverP`	d	`0.8`	`double`
`sigmaMut`	d	`0.35`	`double`
`tournamentRate`	d	`0.05`	`double`
`minNTournament`	i	`3`	`int`
`nPop`	i	`100`	`int`
`nEval`	i		`int`

Figure 3.2: Biped VSR parameters

In Figure 3.2 are shown configurable parameters of the evolutionary algorithm implementation and his default values. In particular, we denote:

- **nEval**: The number of evaluations, which is **not** the number of iterations, it indicates how many individuals are evaluated, it provides an idea of computation load needed to finish the experiment. If *nEval* is set to 1.000 and *nPop* (population size) is set to 100, the resulting iterations number will be: `1.000 / 100 = 10` iterations.

- **mapper**: Which defines parameters relative to the experiment: *unitNumber* (how many *voxel* are present), *nSignals* (number of communication channel between *voxel*), *sensors* and the *function* (in this case MLP) definition. We also decide if controllers are heterogeneous or homogeneous between *voxel*.

After successfully observed the *biped*-**VSR** right-walking, we implemented a new kind of *voxel*, **SA-VSR** (self-assembly VSR), also able to *attach* and *detach* to/from neighbors. Keeping everything else untouched and placing the 64 new *voxels* in a square starting position as shown in Figure 3.3A, we restarted the experiment. This test brought a simple result, *voxels* initially attach just to be able to orderly detach and correctly fall to the right side, as shown in Figure 3.3B.

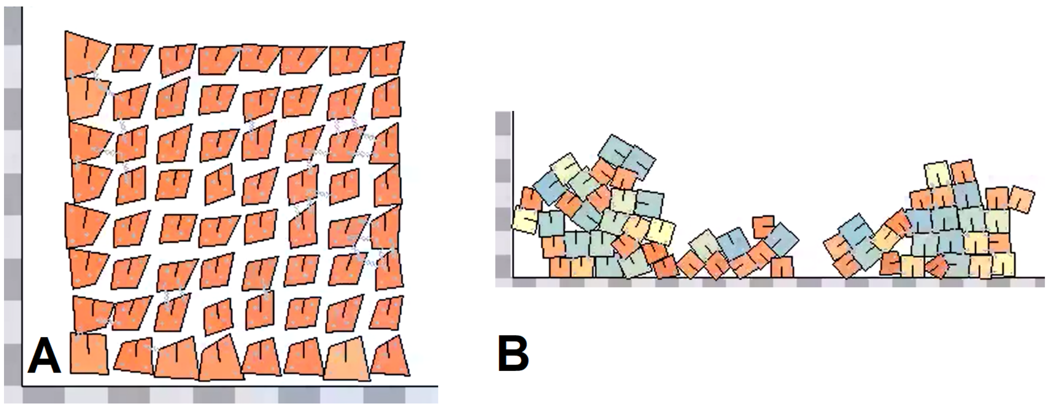


Figure 3.3: SA-VSR square starting position

Subsequent experiments, made with more sensors or changing the task, from right locomotion, to stand piling (reach the maximum possible height), gave similar results as before, falling apart or trying to stand still as long as possible. Sometimes spectacular results shown up, an interesting example is the formation of a spinning top, which used the momentum force to launch a *voxel* into the air, maximizing the height reached. To avoid those solutions, clearly caused by some exploit of the physic engine behind the experiment, we limited actions per time that a *voxel* could make, effectively observing more "stable" (in terms of physic correctness) results.
SA-VSRs brought with them a computation problem, experiments with those *voxel*s were six to seven times slower. In the table below are presented results from experiments with few evaluations, to better understand reasons behind this slowness:

N units	Attach/detach	Threshold	Nearest voxel	Time (m)
64	No	–	–	3.20
64	Yes	0.1	Search	19
64	Yes	0.5	Search	15.30
64	Yes	0.5	Fixed	3.40
8	Yes	0.5	Search	(8 runs) 1.50

A simple run without attach and detach operation, took 3.20 minutes, the same run but with attach and detach operation enabled took 19 minutes. The problem was related to the attachment operation, which brings the search operation to find the nearest *voxel*, to verify it, we tested attach and detach with a fixed *voxel* target, this effectively brought back times like the original one: 4.40 minutes. We could not disable attachments, and we could not implement more efficient research methods (due to time constraint), so we took a two-step solution:

1. Increment the threshold for attach and detach operations, output responsible for that is a value (v) between -1 and 1, if v is greater than the threshold, an attachment attempt is made, contrarily, if v is less than negative threshold, a detachment is made. With a threshold of 0.1, random initial values from MLP caused a lot of attachment/detachment attempts, uselessly incrementing the time computation, with a threshold of 0.5 time reduced from 19 to 15.30 minutes.

2. Lower the number of *voxel*s, this definitely incremented the efficiency, running 8 runs of the same experiment (with different seed for random numbers), each with 8 *voxel*s, lead to a run time of 1.50 minutes, even if the total number of units is the same.

Latest experiments, running 8 runs in parallel, with 8 *voxel*s each and a homogenous controller, starting with laid down units on a line as in Figure 3.4A, shown interesting but not perfect results, shown in Figure 3.4B, achieved by somersaults, climbing, and jumps.

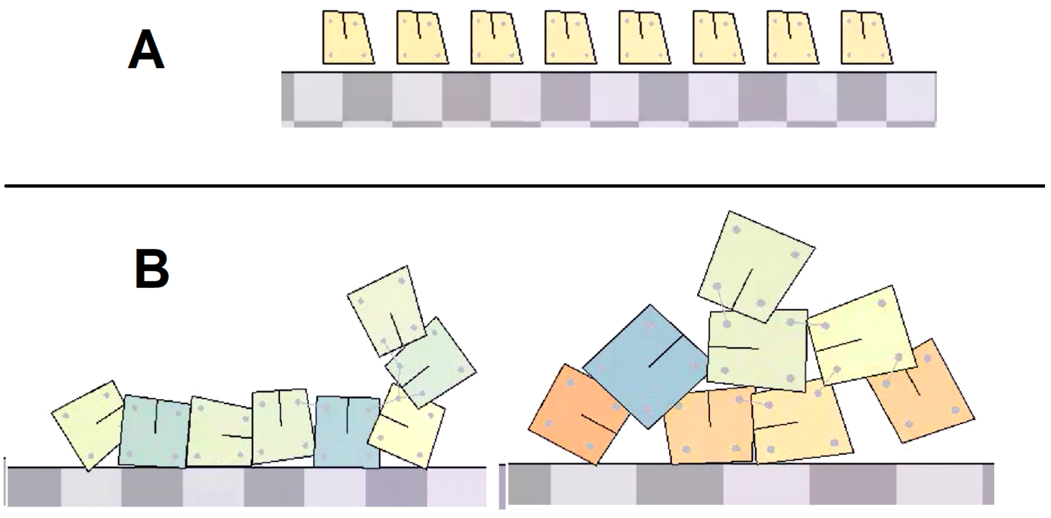


Figure 3.4: SA-VSR starting line position

3.1.2 Covariance Matrix Adaptation-ES

Covariance Matrix Adaptation Evolution Strategy (**CMA-ES**) [18] is an evolution algorithm for numerical optimization. **CMA-ES** has shown more stable and efficient than a classical genetic algorithm[10][5]. We decided to change our optimization method from a generic genetic algorithm to the usage of **CMA-ES**.

For the first tests, we reproduced the same experiments setup as before, within the tasks of right-walking and piling up, obtaining more or less the same results, but with a total different efficiency. Observing the *Fitness* trend in Figure 3.5 it is clear that, both for right walking (left) and piling up (right) experiments, a lot less iterations can or are enough to find a solution comparable with the maximum found all along.

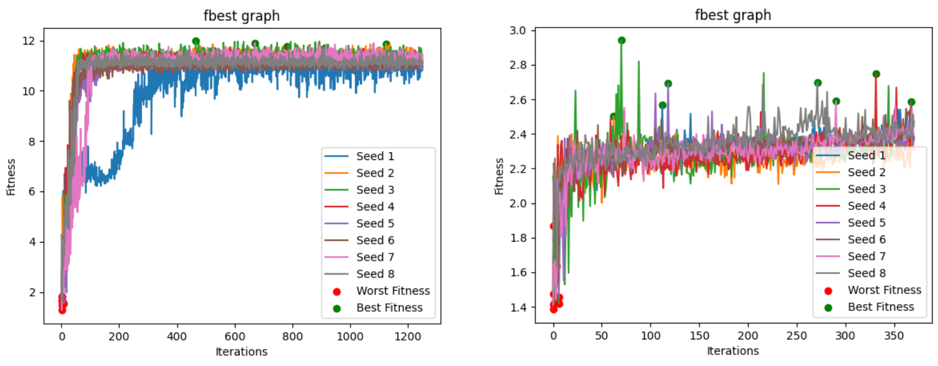


Figure 3.5: Right walking and piling up fitness trend with CMA-ES

This reduction on needed iterations, brought benefits to our experiments; now we were able to add more sensors to *voxel*s and to use a larger MLP, to obtain more complicate controllers which potentially can lead to more interesting solutions or, like in our case, the possibility to solve more complex tasks. This opportunity led us to the definition of the final experiment, later described in 3.2, which briefly consists of **SA-VSR** *voxel*s which needs to collaborate to be able to cross a flat area with a hole on their way.

3.2 Final experiment

In this section, we provide a full description of the experiment done. The idea was to find a task that requires self-assembly to be successfully completed, but at the same time a feasible task. The adequate task was found reading literature described in section 1.2, some projects aim to create robots able to navigate in general unknown and irregular terrains, some more specific simplifies that in sub-problems like locomotion over an inclined plane and hole avoidance. This last scenario, taken from “Cooperative Hole Avoidance in a Swarm-bot”[33] inspired us for the formulation of our research experiment, we thought that a complication of this setup would perfectly fit our needs. We decided to continue research work with the same method (artificial evolution) and to explore a new task: instead of hole avoidance, hole passing over. Pass over a hole involves multiple solutions to be exploited, totally different from obstacle-hole avoidance, in this scenario singular behavior are also viable (jump over), but in larger holes collaboration may be the only way out to complete the task. This experiment, new in the literature, makes it possible to address questions such as: Will units collaborate or split up? As little holes can be passed with small groups of units, will robots prefer small or big groups? Is the ability of attachment and detachment useful? Once formed, will they ever change their shape again?

3.2.1 Setup

The experiment aims to collect relevant data for statistic purpose. We defined parameters for *voxel* and controller and created three similar scenarios to observe different learning curve and success rate. Each scenario starts with the eight *voxel*s positioned on a line placed on the floor, as shown in Figure 3.6A. In front of them a hole which differs in width for each scenario, sizes are $\{[xw + \epsilon, xw + \epsilon, xw + \epsilon]\forall x \in [1, 2, 3]\}$3.6B where w is the width of a single *voxel* and ϵ is a small enough amount to prevent x *voxel*s to be able to cover the hole, but also small enough to permit $x + 1$ *voxel*s to bridge over it, in principle.

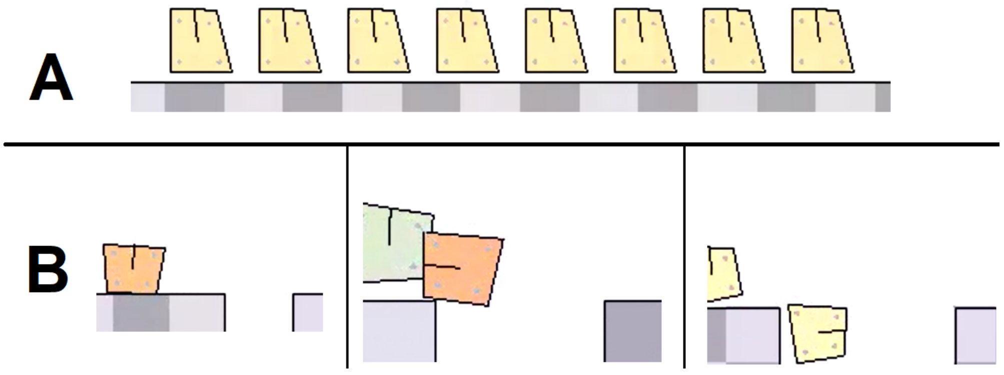


Figure 3.6: Scenarios starting position [A] and holes widths [B]

There are 3 scenarios, each scenario has been tested for a total of 60 *run*s, each *run* has a length of 30 virtual seconds (time is also simulated), for a total of 15 real days of consecutive computation (more details are provided in section 2.3). The general *task* is to reach the greatest distance right-walking. No particular rewards are gained by the effective overcoming of the hole or the velocity, but it is clear that those things indirectly contribute to the fitness (due to time constraint, if a *run* is faster overcoming the hole, it probably will reach greater distances).

Parameters

Each *run* in the same scenario shares the same configuration, the only difference is the *seed* used to generate random numbers, so, differences are the random initial set of weights assigned to the MLP and the selection during the mutation phase of the algorithm. *Voxel* s is homogeneous, it consists of the same MLP network distributed in each one of them. Voxel are situated, i.e., they only know what they perceive and can make actions with their actuators: stretching and releasing springs, communicate with, and attach and detach from neighbors.
In particular, each MLP has an input layer of 18 values, formed like this:

- **14 sensor values**: angle (the current rotation angle of the body), area ratio (current area of the body divided by the rest body area), contact (1 if body has at least one contact with another body, 0 otherwise), distance to body (distance to the nearest body in a specified direction), rotated velocity (current body rotation velocity), side attachment (one for each face, 1 if attached on that side, 0 otherwise), side compression (one for each face, distance between center and the side), sinusoidal (the sin of time passed until *run* beginning, useful for *voxel*s synchronization).

- **4 input communications**: for each side of the *voxel*, a signal from other attached *voxel*s. This can be used to communicate with neighbors. If none *voxel* is attached to a particular side, 0 value is returned.

Three hidden fully connected layers of size 17, 17 and 14.
An output layer of 12 values, formed like this:

- **4 output communications**: for each side of the *voxel*, a signal to send to other attached *voxel*. This kind of value can be written even if none neighbor is attached, it will simply be ignored. Useful to communicate with neighbors.

- **4 attach actuation**: for each side of the *voxel*. If the value is greater than the attachment threshold, then an attempt to attach is made, otherwise if the value is less than the negative attachment threshold, then detach is made. If an attachment or detachment is made without the proper presence of a neighbor, nothing happen.

- **4 *voxel* actuation**: for each spring of the *voxel*, the force to be applied on it.

The MLP has a total of 1029 parameters to be adjusted.
The *CMA-ES* parameters are automatically calculated through formulas provided in Table 1 of cited paper[17]. In detail, we have:

- **p**: number of adjustable parameters in MLP = 1029
- **populationSize**: $4 + \lfloor 3 \cdot \ln p \rfloor = 24$
- **chiN**: $\sqrt{p} \cdot \left(1 - \frac{1}{4 \cdot p} + \frac{1}{21 \cdot p^2}\right) = 33.9458$
- **mu**: $\lfloor \text{populationSize}/2 \rfloor = 12$
- **unnormalizedWeights[mu]**: $\ln\left(\frac{\text{populationSize}+1}{2}\right) - \ln(i+1) \forall i \in (0 \leq i \leq \text{mu}) = [6.93147, 3.2581, 2.19722, 1.60944, 1.20397, 0.916291, 0.721348, 0.587787, 0.491536, 0.420195, 0.365946, 0.324236]$
- **sumOfWeights**: $\sum_{i=0}^{mu} \text{unnormalizedWeights} i = 19.6075$
- **sumOfSquaredWeights**: $\sum i = 0^{mu} \text{unnormalizedWeights}_i^2 = 48.0591$
- **weights[mu]**: $\frac{\text{unnormalizedWeights}}{\text{sumOfWeights}} = [0.353553, 0.166667, 0.112707, 0.082004, 0.0613254, 0.0467226, 0.0368196, 0.0300095, 0.0251202, 0.0214262, 0.0184962, 0.0160526]$
- **muEff**: $\frac{\text{sumOfWeights}^2}{sumOfSquaredWeights} = 7.70162$
- **cSigma**: $\frac{(\text{muEff}+2)}{(p+\text{muEff}+5)} = 0.40118$
- **dSigma**: $1 + 2 \cdot \max(0, \sqrt{(\frac{\text{muEff}-1}{p+1}) - 1}) + \text{cSigma} = 1.51666$
- **cc**: $\frac{4+\frac{\text{muEff}}{p}}{p+4+\frac{2 \cdot \text{muEff}}{p}} = 0.123362$
- **c1**: $\frac{2}{(p+1.3)^2+\text{muEff}} = 0.00192997$
- **cMu**: $\min(1 - c1, 2 \cdot \frac{\text{muEff}-2+\frac{1}{\text{muEff}}}{(p+2)^2} + \text{muEff}) = 9.69833$

Fitness

The fitness function for this experiment is defined as: the difference between the current X coordinate of center (as the mean of means of *voxel*s vertexes coordinates) of *voxel*s and the initial position X coordinate of center. To better understand the process of fitness calculation, we can split it in sub-steps as follows. First, for each *voxel* the center *point* is calculated as the mean of the four vertexes of the current polyline which defines his borders, as shown in Figure 3.7A, the red marks indicate the four vertexes, the blue one is the calculated center. Then the center of *voxel*s together is calculated as the mean of centers, as shown in Figure 3.7B, the green point is the actual point used for fitness, in particular the X coordinate is used. From this value is subtracted the first X center calculated, in this way the first fitness evaluated is always 0 and followings are positive if the experiment is going as we want, or negative if it is not.

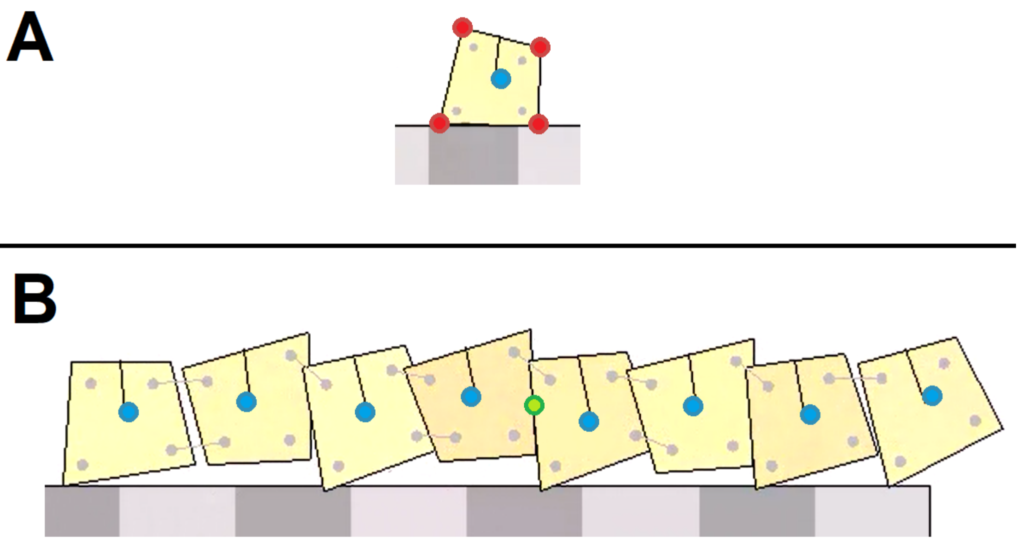


Figure 3.7: *Voxel* fitness calculation, red points are vertices of *voxel*s, blue points are centers, the green point is the calculated final center used for fitness

Major benefits of this fitness function derive from the facility of calculation and definition of success. Thanks to this fitness function and the hole environment, it is possible to simplify the definition of success of a *run*: it is enough to observe that if all the *voxel*s fall down onto the hole, the maximum fitness reachable is a value slightly lesser than the right wall X coordinate (if they pile up against the wall); this means that if the final fitness of the *run* is greater than this threshold, then it is obvious that the hole has been passed, and it can be counted as a success. It is important to notice that there exist cases in which the hole is being passed, but the fitness is below the threshold (e.g., a small hole is being passed from 2 *voxel*s, the other 6 stays in place), those runs will be counted as non successes, but it is not a problem: if they are too slow, or they don't cooperate to reach the task objective, we can consider those as *run* which did not correctly solve the task. Another advantage of using the mean between *voxel*s is that it intrinsically forces the cooperation between them, as a suggestion on how to solve the task.

3.2.2 Results

The final experiment took about 15 days of continuous computation on the cluster2.3 to complete. The results achieved are in the form of videos and numerical data. In short, we can say that the experiment undertaken was a success and provided interesting and useful insights on self-assembly tasks in general. Videos commenting3.2.2 with screenshots and statistic analisys3.2.2 with graphs are described below.

Video results

We think that for evolution experiments, visualization of behavior as videos is both interesting and important for validation. Watching units playing in the ground usually permits to understand how they decided to solve the task, and more important, if their behavior is possible due to some *glitch* in the engine, or it is logically and physically correct. For these reasons, we decided to add this section enriched with screenshots, primarily to detect features of voxel behavior hardly observable using only numerical data.
For all three configurations, if *voxel*s decided not to assembly, the result was the same in each case: a trail of *voxel*s jumping onto the hole once per time3.8, eventually reaching the bottom and failing the task. An intrigue behavior is shown in Figure 3.8A(exp21_42), an example with the smallest hole, a falling *voxel* (marked with a red dot) tries to stick to walls, probably to fill up the hole stopping midair, and enable following *voxel*s to pass over him. Unluckily, the *voxel* did not stick to his position long enough and eventually fell to the bottom.

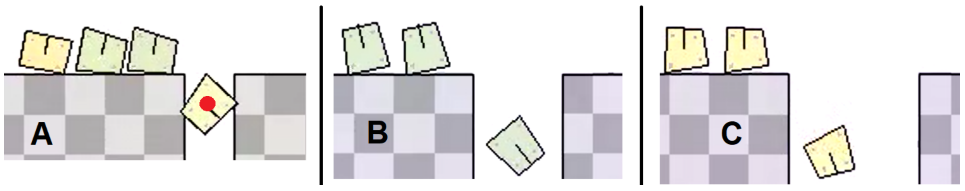


Figure 3.8: Single voxel falling examples in the three hole configurations

When assembly was chosen, two, four or eight long worms were created, between those only the eight-long worm form reached the top as the best in each experiment. With the smallest hole, the decision to assembly was almost always enough to solve the task (in rare cases, an eight-long worm still fell down3.9A) due to wrong disassembly (exp21_32). With the medium hole, a greater effort than simply forming was required, when approaching the hole, it was essential to keep up the head3.9B to be able to bridge over the hole.

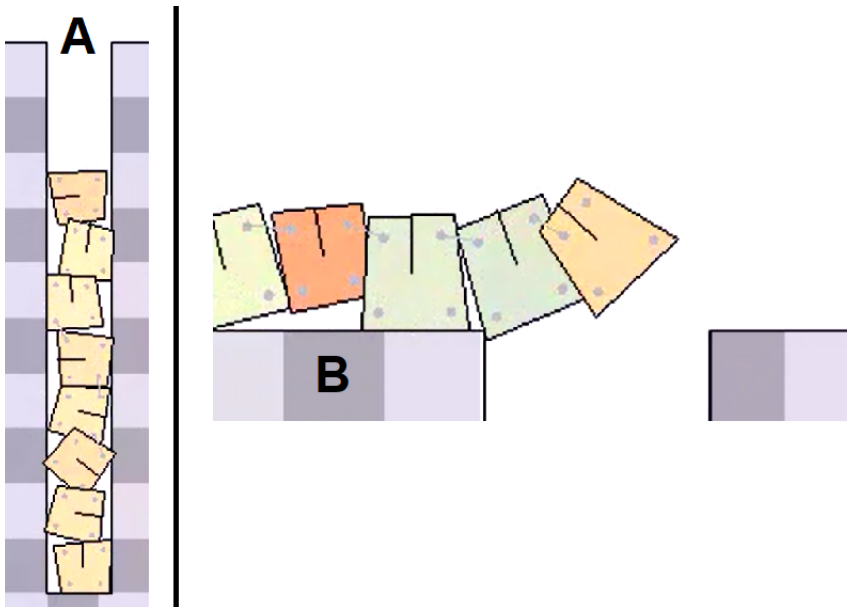


Figure 3.9: Eight-long worms peculiar use cases

With the largest hole, keeping the head up was not sufficient to solve the task, the weight of *voxel*s was too much and this technique lead to the fall of the worm inside the hole. But how can be seen in section 3.2.2, some *run*s with the biggest hole, even if just a few, has succeeded. Solution found to pass a too much large hole is quite intriguing. The eight-long worm is formed, when the hole is approached a quite fast fall onto it begin3.10A, when only few *voxel*s on the tail remains out of the hole, two of them detach3.10B and thanks to the momentum gained from the falling of the head, they successfully reach the other shore, solving the task3.10C.

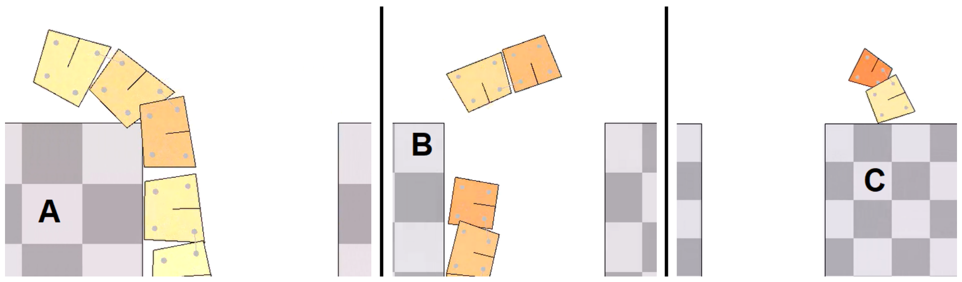


Figure 3.10: Largest hole solution frames

In general, excluding special cases with the largest hole, the most common solution found was to form an eight-long worm, which permit *voxel*s to bridge over the hole without falling into it. That solution was quite expected, due to

initial positions of *voxel*s and his simplicity to be formed, the unexpected is the general slowness of the worm, despite *voxel*s could move faster and reach further distances, they decide to move slow and safe. This could be the result of too short trainings, e.g., not enough time to learn how to make the worm move faster. Run the same experiment with more time given is part of future developments3.3.
The phenomenon, wherein *voxel*s opt to attach to one another, signals their learned capacity to adopt a form configuration suitable to task resolution. This outcome exactly align with the expectations of self-assembly, underscoring the feasibility and potential of such systems.

Statistical results

For statistic purposes, we performed a total of 60 replicas for each experiment configuration3.2.1 ($1+\epsilon, 2+\epsilon, 3+\epsilon$). For each replica, the only changing parameter is the seed used to generate random numbers, resulting in different sets of starting parameters in the MLP network. Doing so, we reached a high enough number of experiments to be statistically relevant, for a total of ∼1.2 GB of CSV files. With the help of python scripts2.2, we post-processed raw data and generated significant graphs, described below.
The violin graphs3.11, for each configuration, show trends of fitness along all iterations. The red line shows the threshold of success, which is different for each graph due to different hole sizes. In the first graph3.11($1+\epsilon$) it is clear that the experiment was a success, the mean of fitness is greater than the success threshold, which indicates, in this case, that the task was simple and fewer iterations could be used to find a merely working solution, surplus iterations are used to improve the speed walking. In the second graph3.11($2+\epsilon$) the mean of fitness is under the success threshold, this does not directly mean that the experiment was unsuccessful, but that it was harder for *voxel*s to learn how to pass over the hole. The success of configurations, for $1+\epsilon$ and $2+\epsilon$ is not directly verifiable from those graphs, but looking at trends of both configurations we can say that few iterations were taken to understand how to reach the hole, the majority of iterations finished falling into the hole (the tallest graph part), then succeeding iterations distributes over the threshold with a descending inclination. In $1+\epsilon$ graph, reaching greater fitness, the curve tends to get larger, this is due to a wrong configuration of width of the path, the best *run*s reach the end of the arena hitting the right most wall. This does not directly impact on the results of our experiments, the only observation is that without this limit *voxel*s could have learned to move faster than they actually do.
The third graph3.11($3+\epsilon$) clearly shows a failing experiment, apart from the

starting iterations which has to learn how to walk, almost the whole of *run*s finishes falling into the hole. Success threshold being over the graph, indicates that, even few, some *run* succeed, as shown in section3.2.2.

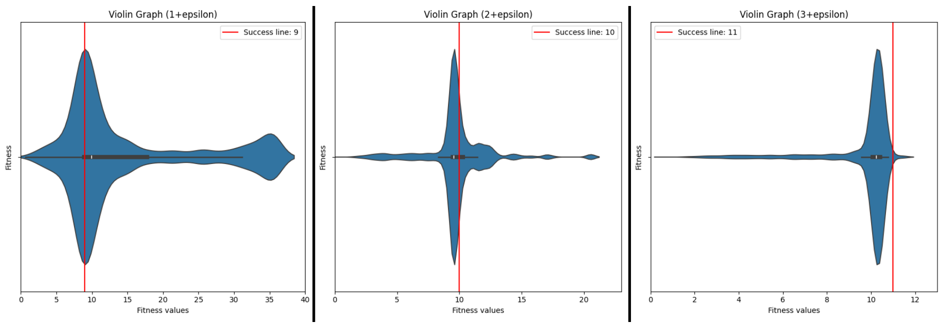


Figure 3.11: Violin graphs for the three configurations

The *Run Length Distribution* (**RLD**) graph3.12 shows the percentage of success (number of succeeding *run*s over the total) over iterations. From this type of graph, it is possible to understand how training went, e.g., if graph shows a long steady line, it means that training has wasted a lot of time. Usually, the line formed has the aspect of a logarithmic function, fast-growing at the beginning and then reaching a more stable value in the end.
The first graph3.12($1 + \epsilon$) shows an almost perfect curve to observe, starts growing early in iterations (∼25) and reach a stable value at the end (∼75%). This idiomatic graph is the proof that experiment $1 + \epsilon$ can be considered as success, and maybe even too easy to be solved. It is interesting, anyway, to notice that adding more *evaluation*s would not have added a lot of contribution to rate success, it is visually correct to say that we stopped when an enough stable value was reached. The second graph3.12($2 + \epsilon$) shows a more realistic case, for the first iterations (∼50) no *run* has success, then the classic curve takes place and reach a stable point, once again, just before hitting the end of iterations. The final percentage of success reached is ∼50%. This means that this variant is harder than the previous one, but still more than feasible. Moreover, the success ratio can be further improved by a fine-tuning of the parameters of the algorithm. The last graph3.12($3+\epsilon$) shows that this variant is way harder than the previous ones. The curve formed begins to grow at the end of the allowed iterations and seems to be stable to 0.05% of success. The best run with this configuration reached a fitness of ∼11.46: considering that the threshold of success is 11, it means that the task was just sufficiently solved. Nevertheless, it is important to emphasize that due to the difficulty of the task, it was not enough to assembly and bridge over the hole. The few successes demonstrate the potential of self-assembly: voxels did not only learn

how to assembly in a suitable form to accomplish their task, but they also learned how to disassembly at the right moment, to exploit the environment in which they are (in this case exploiting gravity to create a catapult).

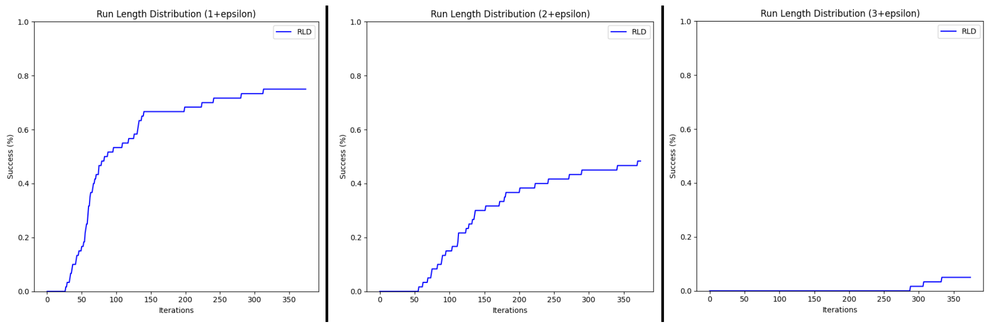


Figure 3.12: RLD graphs for the three configurations

As a further test, we took the succeeding genotypes from $1+\epsilon$ environment and tested them out in $2+\epsilon$ environment, our objective was to verify if the training was general enough to solve more complex cases. This test resulted in a not even one phenotype being able to solve the $2+\epsilon$ environment, also stopping us from trying with the $3+\epsilon$ environment. The result obtained is not so surprising: considering that each task is solved by a peculiar technique, it is quite obvious that the controller generated from the train in a single environment would not generalize over different configurations. Keeping that in mind, we decided to try a new environment, in which the controller could learn to generalize over different holes. We built the new configuration with the three different holes one after each other3.13, and started a new training with more than the triple of time for each *run*. Due to time constraints, we could not run for many iterations, but we achieved an interesting result(exp27_best): an eight-long worm is formed, the first two holes are passed over with the same techniques saw before, then a similar behavior to "catapult" over the third hole seems to happen, but no detachment occurs, and all the worm falls down into the hole.

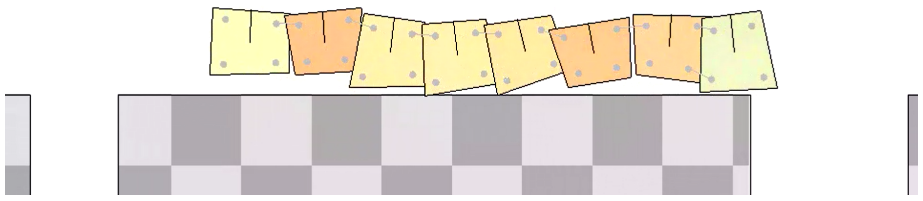


Figure 3.13: Multi hole environment example

In the following, the table with data for each configuration, aggregated per seed relative to the best fitness. Precise data calculation considering only best fitness of each replica confirm what we described above: configurations $1 + \epsilon$ and $2 + \epsilon$ has mean greater than their threshold ($19.212 > 9$ and $11.050 > 10$), they can be considered as success. Configuration $3 + \epsilon$, instead, has the mean lesser than his threshold ($10.391 < 11$) and also the 4^{th} percentile lesser than the threshold ($10.502 < 11$), those results confirm that the experiment has failed. *Max* fitness registered, anyway, shows that some replica has succeeded. Taken in consideration that the $3+\epsilon$ task was difficult, those few solved replica has great value, even if the configuration in its entirely can be considered a failure, we still be able to solve the task3.2.2. Probably, with more iterations, more replicas would have succeeded, further experiments would be necessary to confirm it3.3.

Dataset	Count	Mean	Std	Min	25%	50%	75%	Max
$1+\epsilon$	60.000	19.212	11.398	8.593	8.984	12.141	32.856	35.824
$2+\epsilon$	60.000	11.050	2.301	8.876	9.549	9.944	11.690	20.533
$3+\epsilon$	60.000	10.391	0.312	9.326	10.234	10.379	10.502	11.467

Final observations

Results obtained in those experiments are remarkable: they prove that the concept of self-assembly is not only a futuristic idea, but that it can be implemented and used to generate interesting solutions. It is important to notice that we never directly told the *voxel*s how they should have formed or even that there was a hole to pass over, we only defined that the objective was to maximize the `X` of their center. *Voxel*s learned that in the environment there was a hole to overpass, organized themselves to obtain the most suitable form to solve the task, and decided that bridging over the hole was a solution. All those facts can be considered a good starting point to continue the exploration of self-assembly, and demonstrate that the idyllic idea behind it is not so unachievable and should be pursued.

3.3 Future developments

In this section, we list a set of possible paths which can be chosen to continue the work described.

3.3.1 Time

It may sound trivial, but time is important. Most of the choices taken has been limited by time, the most crucial being in regard of parameters of experiments. Due to time constraints we could not explore in deep every single experiment we made, we had to move from one experiment to another with full knowledge of the facts. Taking in consideration that to achieve a good enough result an experiment needed around 3–4 days of training, it is easy to see that it was important to launch them knowing what will probably happen and to avoid errors like bad configurations (e.g., misspelled a parameter), bug in the code or steady learning line (training blocked on local minimum), it was important to be sure in advance that 3–4 days later some interesting facts eventually popped up. Due to this fact, it was crucial to test some scenario before the final experiment, which lasted for 15 days of continuous computation. Preliminary tests were shorter and necessary, took approximately 70 days of computation, but surely enabled us to experiment and recalibrate one's sights.
It could be interesting to dedicate more time on the final experiment, simply incrementing the number of the evaluations or even changing parameters to complicate it, for longer *run*s the population size could be incremented, the MLP network could be enlarged with more inner layers, or inputs in terms of sensors and communication could be increased. It could be interesting to even try to replace the MLP network in its entirety, i.e., with a delayed recurrent neural network or other kind of networks.
With more time, it could be wise to adopt strategies of incremental learning and robot shaping[12], changing the environment as time passed or when the task is successfully achieved. This kind of methodology could improve learning rate on harder tasks, taking in consideration our experiment, we could define success of the task when all the *voxel*s pass over the hole without falling, then increase the hole width without starting ahead the experiment, but preserving the weights learned until this point. Stop condition, instead of number of iterations, could be defined as a specific number of tries in a row without a scenario change. If this number is high enough and sufficient time is provided, this experiment setup could possibly find the limit of the greatest hole a certain number of *voxel* can overpass.
Another interesting test that could be done with more time given, could be the randomization of starting positions of the *voxel*s, instead of starting on a line, equally separated, laying down on the floor, it may be wise to define a starting region where they can spawn and generate them randomly inside it (without overlapping) for each iteration. This should augment the complexity of the task without changing it, and represents a more realistic situation, where initial setup is not static or known in advance. More time should be granted,

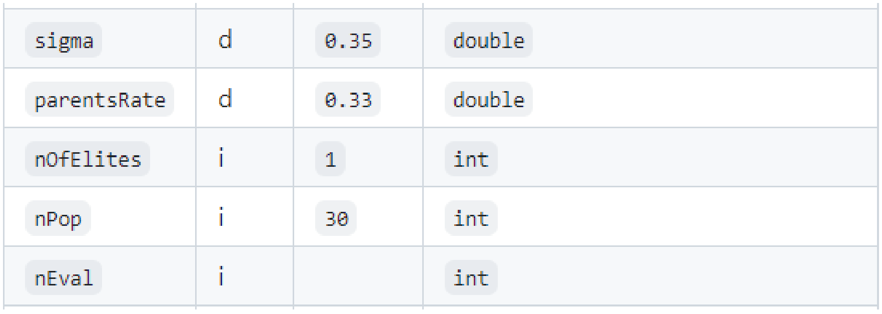
to permit *voxel*s being able to learn how to ensemble from random starting positions.

3.3.2 Algorithms

Within this experiment, only two optimization algorithms were tested, firstly approaching the problem with a *genetic algorithm*, later with *CMA-ES*. It is important to keep in mind that there exists a lot of other optimization algorithms, and it could be interesting to explore behavior with different approaches. A description follows of algorithms already implemented in *JGEA*, which could actually being tested simply changing the configuration file.

Simple evolutionary strategy

A simple implementation of an evolutionary strategy [8], the base concept from which genetic algorithms are born, the main difference resides on the way the next generation of population is selected and mutated. A mechanism of elitism in enabled to ensure the best genotype found is preserved along iterations, also speeding up the learning, reducing number of iterations needed. In Figure 3.14 are shown parameters for an evolutionary strategy.



`sigma`	d	`0.35`	`double`
`parentsRate`	d	`0.33`	`double`
`nOfElites`	i	`1`	`int`
`nPop`	i	`30`	`int`
`nEval`	i		`int`

Figure 3.14: Simple evolutionary strategy parameters

To better understand parameters shown in Figure 3.14, a detailed explanation is provided:

- **sigma**: is a parameter which regulates the random Gaussian value extracted during offspring generation, in contradiction to genetic algorithms where genetic operators are used, in basic evolutionary offspring is generated from mean of parents, randomized by a Gaussian.
- **parentsRate**: the percentage of population selected as parents for next generation, at every iteration `parentsRate%` of the population genotype is selected to form the offspring genotypes.

- **nOfElites**: elitism is a mechanism to preserve the best genotypes of each iteration to survive until the next generation, this parameter regulates how many bests are selected to be maintained, if 0 the elitism mechanism is disabled.
- **nPop**: the parameter which regulates the size of each offspring, the higher it gets, the more 'near" to best genotype genotypes are evaluated. An *nPop* of 0 is not possible.
- **nEval**: regulates the number of evaluations made in the entire *run*. An evaluation is the calculation of a phenotype fitness.

The generation of the offspring for the new iteration starts with the selection of the elites, the first *nOfElites* individuals are saved:

```
elites = state.listPopulation().stream().limit(nOfElites).toList();
```

Then parents are selected:

```
parents = state.listPopulation().stream().limit(nOfParents).toList();
```

Then the parameters means of parents genotypes is calculated:

```
means =
   meanList(parents.stream().map(Individual::genotype).toList());
```

Finally the offspring is generated, a number of $(populationSize - -elites.size())$ new individuals are created starting from the mean of parameters of parents. To slightly change the mean genotype, for each new individual each parameter is multiplied by a random Gaussian value attenuated by the *sigma* factor, generating individuals genetically near to parents. New individuals are merged with elites before selected, forming the offspring:

```
   offspringGenotypes = IntStream.range(0, populationSize -
      elites.size())
      .mapToObj(i -> sum(means, buildList(means.size(), () ->
         random.nextGaussian() * sigma)))
      .toList();
```

Differential evolution

Differential Evolution(**DE**) is a population-based optimization algorithm, the main difference with previous is that it does not use the gradient of the problem being optimized so it does not require the optimization problem to

be differentiable or continuous[2].
DE[1] requires the definition of upper and lower bounds for each parameter, these $2D$ values can be collected into two, D-dimensional initialization vectors, b_L and b_U, respectively lower and upper bounds. Once those bounds are defined, each parameter is given a random value within his range, for example, the initial value for the i^{th} parameter can be calculated as:

$$X_i = rand(0,1) * (b_{iU} - b_{iL}) + b_{iL}$$

Once initialized, DE mutates and recombines the population to produce a population of Np trial vectors. In particular, differential mutation adds a scaled, randomly sampled, vector difference to a third vector, a mutant vector V_i can be calculated as:

$$V_i = X_0 + F * (X_1 - X_2)$$

Where $F \in (0, 1+)$, is a positive real number that controls the rate at which the population evolves. The *base vector* X_0 can be determined in various ways (also randomly) and has to be different from *target vector* V_i. *Difference vectors* are randomly selected once per mutant.

Artificial intelligence

A different and innovative approach to the problem of optimizing could be the choice to use **AI** algorithms. This section could be infinitely long and could get off-topic, for this reason and for the sake of the reader, we limit it to the explanation on how **Deep Reinforcement Learning (DRL)** could be used for this experiment.
DRL is an innovative and general purpose *AI* method to train an **agent** to reach his **goal**, defining a **reward function** which assign a fitness or **reward** to each **state** of the *agent*. Formally, *DRL* [6] is described as a **Markov decision process (MDP)**, which consists of:

- a set of states S.
- a set of actions A.
- transition dynamics $T(s_{t+1}|s_t, a_t)$ that map a state-action pair at time t, onto a distribution of states at time $t + 1$.
- a reward function $R(s_t, a_t, s_{t+1})$.

[2]`https://en.wikipedia.org/wiki/Differential_evolution`

- a discount factor $\gamma \in [0,1]$, where lower values place more emphasis on immediate rewards.

In general, a policy, denoted as π, maps states to a probability distribution over actions, represented as $\pi : S \rightarrow p(A = a|S)$. In episodic *MDP*s, where the state resets after each episode of length T, a sequence of states, actions, and rewards within an episode forms a trajectory or rollout of the policy. Each rollout accumulates rewards from the environment, resulting in a return defined as $R = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$, where γ is the discount factor. The objective of *DRL* is to determine an optimal policy, denoted as π^*, that maximizes the expected return from all states:

$$\pi^* = \text{argmax}_\pi \mathbb{E}[R|\pi]$$

In our experiments the *agent* controller should be the same for each *voxel*, the *state* could be the set of data perceived by sensors, *action*s the *voxel* actuation, *reward function* could not remain the same as the actual fitness, some cautions should be taken in consideration, to avoid classic problems of *DRL*.

3.3.3 New tasks

Hole passing over is just one task that self-assembly can overcome, many use case could benefit from the usage of self-assembly. Keep in mind that self-assembly should become generic: theoretically and idiomatically the learning process should not depend on a specific task, but should enable roots to accomplish any feasible task they are assigned. A solution to achieve the idiom could be the merging of different task-trained robots, not physically putting different robots with different controllers together, but finding a way to generate a multitask controller by the mean of union of single task controllers. For this reason, it is important to continue the exploration of tasks which could take advantage from self-assembly and try to exploit them. An example follows of an interesting task to explore and test.

Ladder task

In contradiction to *piling up*, the *ladder* task taken from ant behavior[4], aim to the formation of a ladder starting from the top of an overhanging bank and adding units at bottom until ground is reached. When the ladder is formed, remaining units can use it to safely reach the ground3.15. This task forces in some way units to get assembled and find out how to safely create the ladder, climbing down each other, and avoid launching themselves down the hole.

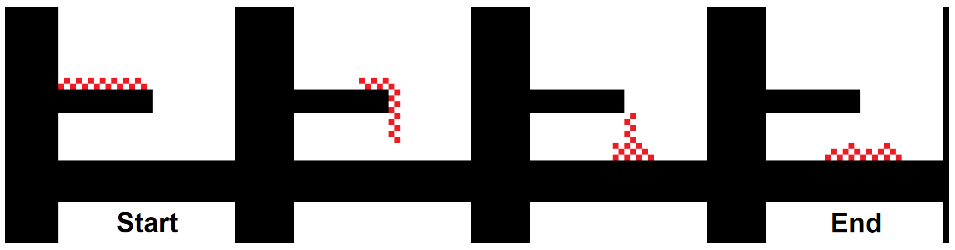


Figure 3.15: Hypothetical frames representing ladder task

To formalize the task, we need to define an *environment* and a *fitness* function. The *environment* is fairly easy to imagine, we can refer to Figure 3.15 first frame, to understand how the terrain could be formed, and where *voxel*s should be startlingly placed (represented by red squares). The *fitness* could be the mean of the eights of *voxel*s, and the objective to minimize it. This setup, anyway, would not avoid *voxel*s from simply launching themselves down the hill, and due to this solution being the simplest and fastest, it surely would be picked as the best. To prevent this behavior, we could abolish hitting the ground too heavily by taking away *voxel*s which, at a certain moment, has a sudden change in velocity (e.g., when they hit the ground after falling from too high), mathematically with a threshold on the acceleration (derivative of velocity), if absolute value of acceleration is greater than the predefined threshold, it means that the *voxel* was not prudent enough and gets removed. The *voxel* could both be physically removed or his controller disabled and his position not counted for *fitness*, it could be interesting to see if other *voxel*s would use "dead" ones for their profit, or simply ignore them.

Conclusion

This thesis has provided an in-depth exploration of self-assembly in *voxel*-based robots, opening new avenues to contribution and research. Through the study of self-assembly and the final experiment, we have demonstrated the potential of *voxel*-based units to autonomously organize and assembly, without the needs of a centralized controller, to achieve complex tasks. The final experiment, featuring homogeneous *voxel*-based robots in an environment with a hole, was successfully solved in each of its three configurations, including small, medium, and large holes, demonstrating the capability of robots to overcome the obstacle. The remarkable aspect lies in the fact that we never told *voxel*s the environment contained a hole, how to ensemble, or to ensemble. The only thing we asked *voxel*s was to maximize their horizontal distance to origin, they autonomously learned that in the environment there was a hole, and more important they autonomously decided to ensemble and what form to create to be able to succeed. The last observation corresponds to fundamental aspects of self-assembly and prove so its feasibility, underscoring the importance of further research in this direction.

In conclusion, the research presented in this thesis not only advances our understanding of self-assembly in *voxel*-based robots but also contributes a significant step towards the realization of more autonomous, adaptable, and intelligent robotic systems. As we move forward, the insights gained from this study will undoubtedly inform and inspire future research and development in the field of intelligent robotic systems.

Acknowledgements

A huge thanks to supervisor, Prof. Roli, for the contribution in each step of the thesis.
Another huge thanks to co-supervisor, Prof. Medvet, creator of *2D Robot Evolution*, for his continue and always gentle support.
A thanks to co-supervisors, Dott. Braccini and Dott. Baldini, for their contribution in tests and experiments.
A thanks to Prof. Pianini and Dott. Baiardi, system administrators of *Cluster 4.0*, for their active support in the usage of cluster.
An important thanks to Lucia Sacchetti, the one who always personally supported me and my studies in the last years.

Bibliography

[1] *The Differential Evolution Algorithm*, pages 37–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[2] The day the earth stood still, 2008.

[3] Big hero 6, 2014.

[4] C. Anderson, G. Theraulaz, and J.-L. Deneubourg. Self-assemblages in insect societies. *Insectes Sociaux*, 49(2):99–110, 2002.

[5] M. Anis, S. Pendurkar, Y.K. Yi, and G. Sharon. Comparison between popular genetic algorithm (ga)-based tool and covariance matrix adaptation, evolutionary strategy (cma-es) for optimizing indoor daylight. 2023.

[6] K. Arulkumaran, M.P. Deisenroth, M. Brundage, and A.A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[7] Z. Beheshti and S.M.H. Shamsuddin. A review of population-based metaheuristic algorithms. *Int. j. adv. soft comput. appl*, 5(1):1–35, 2013.

[8] H.G. Beyer and H.P. Schwefel. *Natural Computing*, 1(1):3–52, 2002.

[9] J. Bishop, S. Burden, E. Klavins, R. Kreisberg, W. Malone, N. Napp, and T. Nguyen. Programmable parts: a demonstration of the grammatical approach to self-organization. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3684–3691, 2005.

[10] C. Cai and H. Jiang. Performance comparisons of evolutionary algorithms for walking gait optimization. In *2013 International Conference on Information Science and Cloud Computing Companion*, pages 129–134, 2013.

[11] M. Dorigo. Swarm-bot: an experiment in swarm robotics. In *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005.*, pages 192–200, 2005.

[12] M. Dorigo and M. Colombetti. *Robot shaping: An experiment in behavior engineering*. MIT Press, 1998.

[13] R. Fitch, D. Rus, and M. Vona. A basis for self-repair robots using self-reconfiguring crystal modules. In *Intelligent Autonomous Systems*, volume 6, pages 903–910. Citeseer, 2000.

[14] T. Fukuda, S. Nakagawa, Y. Kawauchi, and M. Buss. Self organizing robots based on cell structures - ckbot. In *IEEE International Workshop on Intelligent Robots*, pages 145–150, 1988.

[15] K. Gilpin, K. Kotay, D. Rus, and I. Vasilescu. Miche: Modular shape formation by self-disassembly. *The International Journal of Robotics Research*, 27(3-4):345–372, 2008.

[16] R. Gross, M. Bonani, F. Mondada, and M. Dorigo. Autonomous self-assembly in swarm-bots. *IEEE Transactions on Robotics*, 22(6):1115–1130, 2006.

[17] N. Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[18] N. Hansen and A. Auger. Cma-es: evolution strategies and covariance matrix adaptation. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, page 991–1010, New York, NY, USA, 2011. Association for Computing Machinery.

[19] E. Klavins. Programmable self-assembly. *IEEE Control Systems Magazine*, 27(4):43–56, 2007.

[20] K. Lee and G.S. Chirikjian. Robotic self-replication. *IEEE Robotics & Automation Magazine*, 14(4), 2007.

[21] H. Li, T. Wang, H. Wei, and C. Meng. Response strategy to environmental cues for modular robots with self-assembly from swarm to articulated robots. *Journal of Intelligent & Robotic Systems*, 81(3):359–376, March 1 2016.

[22] H. Li, H. Wei, J. Xiao, and T. Wang. Co-evolution framework of swarm self-assembly robots. *Neurocomputing*, 148:112–121, 2015.

[23] E. Medvet, A. Bartoli, A. De Lorenzo, and G. Fidel. Evolution of distributed neural controllers for voxel-based soft robots. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, page 112–120, New York, NY, USA, 2020. Association for Computing Machinery.

[24] E. Medvet, A. Bartoli, A. De Lorenzo, and S. Seriani. 2D-VSR-Sim: A simulation tool for the optimization of 2-D voxel-based soft robots. *SoftwareX*, 12, 2020.

[25] E. Medvet, A. Bartoli, A. De Lorenzo, and S. Seriani. Design, Validation, and Case Studies of 2D-VSR-Sim, an Optimization-friendly Simulator of 2-D Voxel-based Soft Robots. *arXiv preprint arXiv:2001.08617*, 2020.

[26] Eric Medvet, Giorgia Nadizar, and Luca Manzoni. Jgea: a modular java framework for experimenting with evolutionary computation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '22, page 2009–2018, New York, NY, USA, 2022. Association for Computing Machinery.

[27] R. O'Grady, A.L. Christensen, and M. Dorigo. Self-sssembly and morphology control in a swarm-bot. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2551–2552, 2007.

[28] R. O'Grady, R. Gross, A.L. Christensen, F. Mondada, M. Bonani, and M. Dorigo. Performance benefits of self-assembly in a swarm-bot. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2381–2387, 2007.

[29] B. Salemi, M. Moll, and W. Shen. Superbot: A deployable, multi-functional, and modular self-reconfigurable robotic system. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3636–3641, 2006.

[30] P. Swissler and M. Rubenstein. Fireantv3: A modular self-reconfigurable robot toward free-form self-assembly using attach-anywhere continuous docks. *IEEE Robotics and Automation Letters*, 8(8):4911–4918, 2023.

[31] W. Tan, H. Wei, and B. Yang. Sambotii: A new self-assembly modular robot platform based on sambot. *Applied Sciences*, 8(10), 2018.

[32] K. Tomita, S. Murata, H. Kurokawa, E. Yoshida, and S. Kokaji. Self-assembly and self-repair method for a distributed mechanical system. *IEEE Transactions on Robotics and Automation*, 15(6):1035–1045, 1999.

[33] V. Trianni, S. Nolfi, and M. Dorigo. Cooperative hole avoidance in a swarm-bot. *Robotics and Autonomous Systems*, 54(2):97–103, 2006. Intelligent Autonomous Systems.

[34] R. Wang, P. Luo, Y. Guan, H. Wei, X. Li, J. Zhang, and X. Song. Timed automata based motion planning for a self-assembly robot system. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5624–5629, 2014.

[35] H. Wei, Y. Chen, J. Tan, and T. Wang. Sambot: A self-assembly modular robot system. *IEEE/ASME Transactions on Mechatronics*, 16(4):745–757, 2011.

[36] H. Wei, D. Li, J. Tan, and T. Wang. The distributed control and experiments of directional self-assembly for modular swarm robots. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4169–4174, 2010.

[37] P. White, V. Zykov, J. Bongard, and H. Lipson. Three dimensional stochastic reconfiguration of modular robots. In *Three Dimensional Stochastic Reconfiguration of Modular Robots*, pages 161–168, 06 2005.

[38] G.M. Whitesides and B. Grzybowski. Self-assembly at all scales. *Science*, 295(5564):2418–2421, 2002.

[39] M. Yim, D.G. Duff, and K.D. Roufas. Polybot: a modular reconfigurable robot. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 1, pages 514–520 vol.1, 2000.

[40] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G.S. Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):43–52, 2007.

[41] M. Yim, Ying Zhang, and D. Duff. Modular robots. *IEEE Spectrum*, 39(2):30–34, 2002.