Projet de Recherche

Novelty Search and Emergence of Communication in a swarm of robots

Author : Matteo Denis **Promotion :** 2024

Mention de confidentialité
Rapport non-confidentiel

Spécialité : Informatique
Année scolaire : 2022/2023

Stage effectué du 15 mai au 18 août 2023

Enseignant référant ENSTA: Alexandre CHAPOUTOT
U2IS ENSTA Paris, 828 boulevard de Maréchaux, 91120 Palaiseau, France

Maître de stage: Paulo Urbano
LASIGE - Faculdade de Ciências da Universidade de Lisboa, Campo Grande 016, 1749-016, Lisbonne, Portugal

Note de confidentialité

Le document est non confidentiel et consultable au format électronique uniquement sur place à la bibliothèque de l'ENSTA Paris.

Acknowledgements

I would like to express my gratitude to Professor Paulo Urbano, for his guidance all along this internship in FCUL. He guided me well, from the beginning, showed and explained to me anything that could help me with this research, so that we were able to talk and understand each other at best. His support was a motivation, and it was very interesting to learn everything alongside him.

Abstract

In Deep-Learning, there are many ways to train an Artificial Neural-Network (ANN). Among this methods, the most frequent ones are the **gradient descent**, and the **neuroevolution**. This research is about the neuroevolution approach, which consist of evolutionary algorithms, inspired by the evolution of species in real life. In order to say if one ANN is efficient or not, we commonly use a **fitness function**, that can evaluate the performances of an ANN. In this research, we make one of this evolutionary algorithm work without a fitness function, but with a **novelty parameter**, that tells if this ANN had a different results from the previous ones, or not. If it has, the ANN is rewarded and has more chances to crossover with other ones, like **natural selection** in real life.

The second part of this research is to establish this method on a problem that contains a form of communication. For this, we adress the problem of **orientation consensus**. A number of robots are randomly disposed on a map. The goal for them is to have all the same orientation at a time. To achieve this, a minimal way of communication between them is allowed, and all are controlled by the same ANN.

Keywords

- Deep-Learning
- Artificial Neural-Network
- Evolutionary Algorithms / Neuroevolution
- Maze
- Orientation consensus

Contents

Note de confidentialité **2**

Acknowledgements **3**

Abstract - Keywords **4**

Table of contents **5**

Tables of figures **7**

Introduction **8**

Description of my work **9**

1 Part 1: Learning about the NEAT algorithm **10**
1.1 Basis of neuroevolution . . . 10
1.2 Genetic operators . . . 10
Mutation operator . . . 10
Crossover operator . . . 11
1.3 NEAT algorithm . . . 11
Genetic encoding . . . 11
Crossover with an innovation number . . . 12
Speciation . . . 13
1.4 Python library : NEAT-Python . . . 14
1.5 First experiment : the XOR experiment with NEAT . . . 16
The fitness function . . . 16
Running the experiment . . . 16
1.6 The maze experiment with NEAT . . . 18
The deceptive nature of the problem . . . 18
The maze-navigating agent . . . 19
The maze simulation environment . . . 20
Calculation of the fitness score . . . 20
Running the experiment . . . 21
With the medium maze . . . 21
With the hard maze . . . 23

2 Part 2 : The Novelty Search algorithm **24**
2.1 Novelty Search general insight . . . 24
2.2 The maze experiment with Novelty Search . . . 24
The novelty score . . . 24
Running the experiment . . . 25
With the medium maze . . . 26
With the hard maze . . . 29

3 Part 3 : The orientation consensus problem **30**
3.1 Description of the orientation consensus problem 30
3.2 The communication system 30
3.3 Running the experiment via NEAT algorithm 32
The fitness function 32
Running the experiment 32
Without the message 32
With the message 34
Error made and what to change 37
3.4 Running the experiment via Novelty Search algorithm 37

Conclusion **38**

Planning of the internship **39**

Glossary **39**

Bibliography **40**

Annexe **41**

List of Figures

1 The evolutionary process . . . 11
2 The NEAT genome scheme . . . 12
3 A genome scheme for the XOR experiment (1.5) . . . 12
4 Recombination process in NEAT Algorithm . . . 13
5 The speciation algorithm in NEAT . . . 14
6 The first lines of the configuration file providing the hyperparameters of the XOR experiment (1.5) . . . 15
7 Initial and optimal XOR phenotypes . . . 16
8 The phenotype found for solving the XOR problem . . . 17
9 Evolution of fitness and species during a XOR solving ANN evolution . . . 18
10 A two-dimensional maze, with its deceptive local maximum in dead ends . . . 19
11 The schema for our maze agent . . . 19
12 The text file describing the maze . . . 20
13 The medium maze configuration . . . 22
14 The phenotype found for solving the maze experiment via NEAT . . . 22
15 Evolution of fitness and species during this experiment via NEAT . . . 22
16 The hard maze configuration . . . 23
17 The optimum wells of the hard maze . . . 23
18 Average and best fitness in 10 trials of the maze experiment with Novelty Search 26
19 The phenotype found for the maze experiment via Novelty Search . . . 27
20 Evolution of fitness and species during this experiment via Novelty Search . . . 27
21 The record of every final positions the NS algorithm wen through . . . 28
22 The path of the successful maze solver agent . . . 28
23 The Novelty Search exploration in the hard maze . . . 29
24 The random disposition of robots in an orientation consensus problem . . . 30
25 The sensors configuration of an E-Puck robot . . . 31
26 Relative heading between two robots during a communication between both . 31
27 The phenotype of the solving ANN . . . 33
28 Fitness and species evolution during the orientation consensus without the message 33
29 Initial and final orientations of the robots controlled by the best genome's ANN 34
30 The phenotype of ANN solving the orientation consensus problem with a direct mean of communication . . . 35
31 Fitness and species evolution during the orientation consensus with a direct mean of communication . . . 35
32 Initial and final orientations of the robots controlled by the best genome's ANN 36
33 The proportion estimations of each symbol used in communications between robots (3) . . . 37

Introduction

Deep-learning and **Artificial Neural Networks (ANN)** are the innovation of the future in the domain of AI. They can treat almost every problem and there is no doubt that in the future, they will replace other methods of machine-learning. However, there is still a lot to understand about it, and training these networks is not an easy thing. This research focuses on a new way to train them.

This research is divided in two parts :

1. the algorithm of **Novelty Search**, a new way to do neuroevolution.
2. the implementation of this algorithm in the **orientation consensus** problem, and emergence of **communication** in a swarm of robots.

Neuroevolution is a form of artificial intelligence that uses evolutionary algorithms to generate an Artificial Neural-Network. The most common of these algorithms, and the one that we will use for this research, is **NeuroEvolution of Augmenting Topology (NEAT)**. This algorithm treats a large population of **genomes** (ANNs), and through generations, make them have **mutations and crossovers** between them, to imitate **natural evolution**. In this process, it is necessary to evaluate the performances of each genome of the population. For this evaluation, the classical way is to use a **fitness function**. In the example of a maze, the fitness function could be the distance between the maze exit and the final position of the robot controlled by this ANN. The algorithm of Novelty Search replaces this fitness function by a **novelty score**, that focuses on whether or not the behavior of the ANN is different from the previous ones. In the example of a maze, the novelty score could be the difference between the final position of the robot controlled by this ANN, and the mean position of the previous ones (but this is not what we will use in this research). We will describe in details how the novelty score is calculated in Chapter 2.2.

After having experimented the novelty search algorithm on different maze problems, we have implemented them on a more complex one : the **orientation consensus** problem. In this problem, a number of robots are randomly disposed on a map. The goal for them is to have all the same orientation at a time. To achieve this, all robots are controlled by the same ANN, and they can communicate between them. The mean of communication is minimal : 16 symbols are allowed and they have additional information on which radar of the sender did emit the signal, and which radar of the receiver did receive the message. The communication system will be described more in details in chapter 3.2 . The robots of our simulation are inspired by **E-Pucks robots**, which characteristics are described in figure 25. We will then be able to see the results of our Novelty Search algorithm in a more complex problem than a maze, and also give insights of how the robots communicate between them, and think of how this communication can be improved.

Description of my work

In this research led by Professor Paulo Urbano, my role was to fully implement the algorithms in autonomy. With his frequent help and guidance, I was able to fully understand the Novelty Search algorithm given in Chapter 6 of Hands-On Neuroevolution with Python by Iaroslav Omelianenko (1), to test it on different mazes, with different parameters and try and have the best results. To achieve this, Prof. Urbano gave me the book (1) that guided me in this phase of learning. The first chapters give general knowledge about neuroevolution, until Chapter 5 where the NEAT algorithm is described and tested on a maze experiment. Chapter 6 describes the Novelty Search variant. Going through this book, I was able to test these algorithms on different mazes, and with different parameters, slightly changing them to improve them. Once I was able to fully understand these algorithms, I implemented additional functions of visualization, in order to have a better view on the results of our experiments, which are sometimes difficult to read.

In the second part of the internship, Prof. Urbano gave me enough resources, including Emergence of Communication Through Artificial Evolution in an Orientation Consensus Task in Swarm Robotics by Rafael Sendra-Arranz and Alvaro Gutiérrez (3),to understand the new problem of orientation consensus, and I had to implement the algorithm from scratch. This task being rather long, I was not able to complete what I wanted to do and to implement Novelty Search on this problem. However, I implemented NEAT algorithm, so I was able to see some results about the communication system between robots, which was one of the principal goals of this research.

Even if most of the time I was in autonomy, Prof. Urbano always answered my questions when I didn't understand something about the book, or needed more technical information about anything. His support has been instrumental and always proved beneficial to my work.

1 Part 1: Learning about the NEAT algorithm

The most of this part is drawn from chapters 1 to 5 of (1), as it was the most important resource given by Prof. Urbano.

1.1 Basis of neuroevolution

The term Artificial Neural Networks (ANN) stands for a graph of nodes connected by links where each of the links has a particular weight. It remotely resembles the way in which neurons in the brain are organized. The training process of an ANN, consists of selecting the appropriate number of nodes, links, and the weight values of each links within the network. With this kind of structure, an ANN can approximate any function, even if they are non-linear, and can be considered an **universal approximator**.

The most popular training method in the current decade is based on the **backpropagation** of prediction error through this network, with various optimization techniques built around **gradient descent** of the loss function. Although this methods succeeded in demonstrating the outstanding performance of deep neural networks, it has significant drawbacks. The first one is the fixed network architecture that is created manually by the experimenter, which results in being limited by the imagination of its creator, but also inefficient use of computational resources, as many nodes will finally not participate in the process. Another drawback of this method is the vast amount of training samples required to learn something useful from a specific dataset.

But this training method is challenged by some very promising evolutionary algorithms. These are inspired by **Darwin**'s theory of evolution and use natural evolution abstractions to create artificial neural networks. The basic idea behind neuroevolution is to use **stochastic, population-based** search methods. We make these neural networks evolve to be optimal, and to accurately address a specific task, using the evolutionary process.

1.2 Genetic operators

Genetic operators are the operation made on genomes of a population from one generation to another. There are two major genetic operators : **mutation and crossover**.

Mutation operator

The mutation operator alters one or more genes from the genome. In our ANN, it can be many things :

- Changing a weight value
- Adding a new connection between nodes
- Removing a connection between nodes
- Adding a new node to the network
- Removing a node of the network

The mutation operator serves the essential role of preserving the genetic diversity of the population and prevents from stalling in a local minima when the genomes of a population become too similar. By introducing random changes, mutation allows the evolutionary process to explore new areas in the search of possible solutions and find better ones over generations.

Crossover operator

The crossover operator consists in the recombination of two genomes. It's the equivalent of reproduction from one generation to the next one in real life. The stochastic recombination between two **parents** can lead to a better **offspring**. In addition, to add even more to the stochastic process, the offspring is often mutated before being added to the population of the next generation. We will see in the next section how to know in which way to recombine two parents in order to have the best offspring.

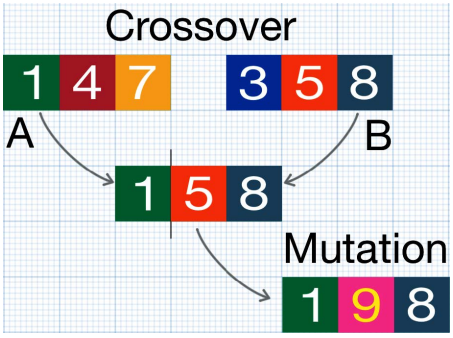

Figure 1: The evolutionary process

The mutation and crossover probability are informed in the **config file**, along with the probability of adding and removing nodes. These probabilities are called **hyperparameters**. This allows the experimenter to guide the evolution of its ANN in many ways.

1.3 NEAT algorithm

The method of NEAT for evolving complex ANNs was designed to reduce the dimensionality of the parameter search space through the gradual elaboration of the ANN's structure during evolution. The evolutionary process starts with a population of **small, simple genomes (seed)**, and gradually increases their complexity over generations. The seed genomes are minimalist : only the input and output nodes are present, with a bias neuron. Such a genome can solve only linear problems. By increasing the complexity of the ANN, we allow it to solve non-linear problems, until it becomes optimistic for our task. The interest in beginning with such small genome is to begin with the lowest possible dimensional parameter space. With each generations, new genes are added, however, it is easier to search for an optimal solution in a small space. The algorithm will add new generations when necessary, with the genetics operator that we talked about in the previous section.

Genetic encoding

The genetic encoding scheme of NEAT is designed to allow easy matching between corresponding genes during crossover operation. A genome is a linear representation of the connectivity between nodes as shown in the following genome scheme :

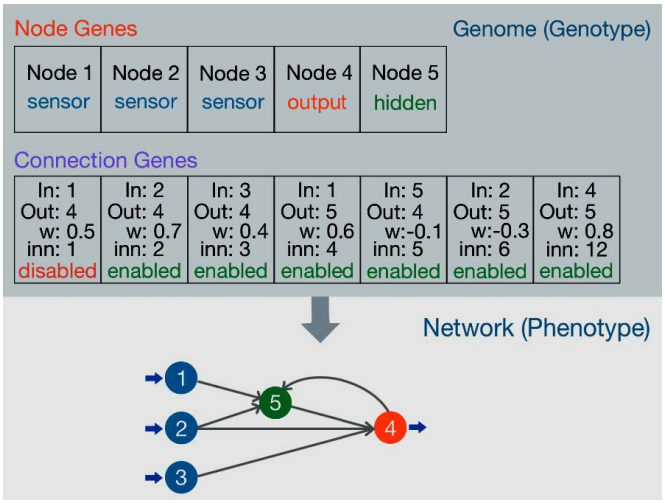

Figure 2: The NEAT genome scheme

```
Nodes:
        0 DefaultNodeGene(key=0, bias=-4.747618724790716, response=1.0, activation=sigmoid, aggregation=sum)
        1947 DefaultNodeGene(key=1947, bias=-0.7698840505510516, response=1.0, activation=sigmoid, aggregation=sum)
        3279 DefaultNodeGene(key=3279, bias=-0.944436898477297, response=1.0, activation=sigmoid, aggregation=sum)
Connections:
        DefaultConnectionGene(key=(-2, 0), weight=5.774527564782061, enabled=True)
        DefaultConnectionGene(key=(-2, 1947), weight=-2.6520780531451424, enabled=True)
        DefaultConnectionGene(key=(-1, 0), weight=-2.440683691667631, enabled=True)
        DefaultConnectionGene(key=(-1, 1947), weight=0.7457286458013794, enabled=False)
        DefaultConnectionGene(key=(-1, 3279), weight=1.6454237278305204, enabled=True)
        DefaultConnectionGene(key=(1947, 0), weight=9.223829385934884, enabled=True)
        DefaultConnectionGene(key=(3279, 1947), weight=2.112926274559135, enabled=True)
```

Figure 3: A genome scheme for the XOR experiment (1.5)

As it is shown in these two figures, genomes are represented as a list of genes that encode connections between the nodes of the ANN. Each node gene encodes information about the neuron :

- node identifier
- node type
- activation function

And each connection gene also encodes information about the connection :

- identifier of the input neuron
- identifier of the output neuron
- weight of the connection
- a bit to encode whether or not the connection is activated
- the innovation number, which makes the crossover operation easier

Crossover with an innovation number

We talked about an **innovation number** to make crossover operation easier, but how does it work ? This number tells us from which ancestor gene this gene was derived. Two genes with the same historical origin represent the same structure, even if they can have different connection values. At each crossover, the offspring gene will inherit the innovation number of the gene of the parent from which it was inherited. If, during a crossover, innovation numbers of some genes between parents don't match, it means that the gene is from the **disjoint or excess** part of the genome. Thus, the offspring inherits from genes that have the same innovation number, randomly chosen from one of the parents. Then, it inherits the disjoint or excess genes of the parent with the higher fitness between both. This feature allows the NEAT algorithm to perform crossover between genomes without the need of complex topological analysis. This idea of smart recombination is shown in the next figure :

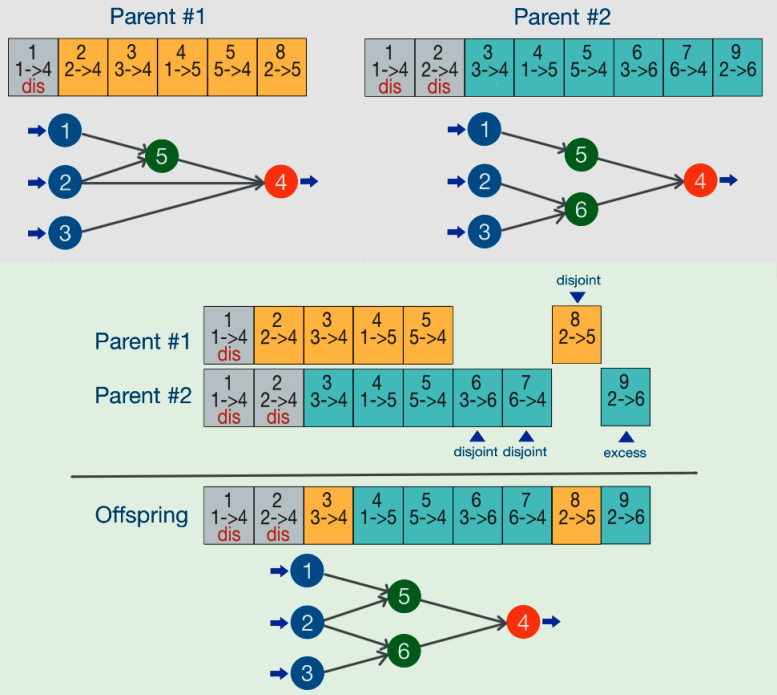

Figure 4: Recombination process in NEAT Algorithm

Speciation

As it is, the evolutionary process will fail to produce and maintain topological innovations on its own. Smaller network are easier to optimize than larger ones, which means that chances are minor that a descendant genome after adding a node or a connection will survive. At the same time, novel topologies can introduce innovations that lead to a winning solution on the long term. For this reason, the concept of **speciation** was introduced in the NEAT algorithm.

The speciation feature limits the range of organisms that can mate together, by introducing groups of genomes, that have the same features, resembling the concept of species in real evolution. With this, only genomes belonging to the same species will compete together, instead of competing with the whole population. This allows our population to diversify, and have many various possible topologies, regrouped in species. The following pseudo-code shows how the NEAT algorithm is capable of regrouping genomes in species, and create new species if needed :

Algorithm 1: Clustering Genomes into Species

Input: A *Population* of organisms and known *Species*
Result: Organisms will be clustered among *Species*. New *Species* will be created as appropriate.

```
foreach genome ∈ Population do
    foreach S ∈ Species do
        if genome.IsCompatible(S) then
            // Add compatible Genome to the current species
            S.AddGenome(genome);
        else if S is the last known species then
            // Create new species for a given genome
            S_new ⟵ create_new_species(genome) ;
            // Add new species to the list of known Species
            Species ⟵ Species ∪ S_new ;
```

Figure 5: The speciation algorithm in NEAT

1.4 Python library : NEAT-Python

In this research, the most important libraries used was **NEAT-Python**. It provides the implementation of the standard NEAT methods, and convenient methods to load and save the genome configurations, and **NEAT hyperparameters**. These hyperparameters give information about many things to run the algorithm, among which :

- population size
- number of inputs, outputs and hidden (bias) nodes to create the seeds
- probabilities of adding and removing a node
- fitness threshold above which the algorithm must stop
- activation function of the neurons
- minimal number of identical genomes to create a new species
- minimal size of a species, under which the species will be extinct
- species threshold, above which two different genomes are considered to be different species
- and many others ...

```
#--- Hyper-parameters for the XOR experiment ---#

[NEAT]
fitness_criterion     = max
fitness_threshold     = 15.5
pop_size              = 150
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.0
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

```

Figure 6: The first lines of the configuration file providing the hyperparameters of the XOR experiment (1.5)

The NEAT-Python library also provides methods that can collect statistics about the evolutionary process, that will be very useful to create visualization functions. Moreover, it is very well comprehensively on `https://neat-python.readthedocs.io/en/latest/`, and is available through the PIP package manager, for an easy installation. Its only weak spot concerning our research is that there isn't a direct implementation of the Novelty Search algorithm, but this is not a problem for us because the book (1) describes a full implementation of it via this library, by converting the fitness evaluation function into a novelty score calculation function.

The implementation using this library is rather simple, thanks to some rather practical functions. First thing is to create a config instance with :
`config = neat.Config(config_file)`
Then we must create the population instance, with :
`p = neat.Population(config)`
Once these two instances are created, we can run the evolution over a specific number of generations, with :
`winner = p.run(eval_genomes, gen_number)`
`winner` is finally the best genome for our task, over `gen_number` generations.
We have to create the function `eval_genomes(genomes, config)`, that calculates the fitness of every genomes in `genomes`. Thus, the main work for us to make the evolution run is to create this function, that can be very complex depending on the task.

There are many other libraries that can manage the NEAT algorithm, such as **Pytorch NEAT, multiNEAT and Deep Neuroevolution**, but most of them are not supported by the PIP package manager, which will complexify its installation.

To set up our environment, we used an **Anaconda distribution**, that allows us to easily create various environments for each of our experiments, as sometimes the libraries that were needed for specific experiments differed.

1.5 First experiment : the XOR experiment with NEAT

The XOR problem solver is a classical computer science experiment that cannot be solved without introducing a non-linear execution to the solver algorithm. We are going to create an ANN and make it evolve until it is able to solve this problem. XOR is a binary logical operator that only returns `True` (1) if only one of the two outputs is `True`. Here is a table defining the XOR features :

Input 1	Input 2	Output
1	1	0
1	0	1
0	1	1
0	0	0

The NEAT algorithm starts with a very simple seed, with 2 inputs, 1 output and one bias neuron (hidden neuron). The next figure shows the initial phenotype and the smallest possible one solving the XOR problem :

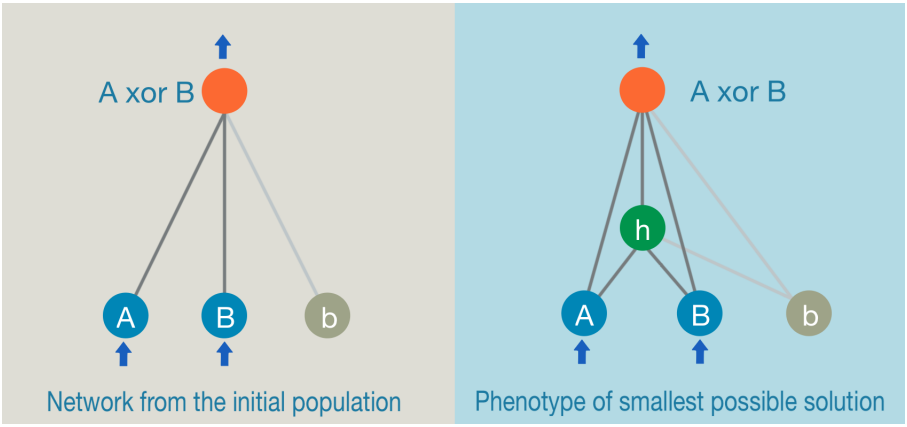

Figure 7: Initial and optimal XOR phenotypes

The fitness function

Our main job is to define the fitness function in the `evaluate_genomes` function. For the XOR problem, our fitness function will be the **squared distance between the correct answer and the sum of the outputs generated by our genome for all four input patterns**. It can be defined as :

$$f = \left(4 - \sum_{i=1}^{4} |y_i - ANN\,(x1_i, x2_i)|\right)^2$$

Thus, as you can see, all genomes are compared with all four possible input patterns. Unfortunately, our output neuron will return a float number between 0 and 1. We will fix our `fitness_threshold` at 15.5, to make sure that our algorithm stops when the solution found is maximally close to the hypothetical goal of 16.

Running the experiment

We run the experiment with the source code written in Annexe 2 : XOR experiment source code with the hyperparameters described in Annexe 1 : XOR configuration file. With these hyperparameters, we find a solution in 210 generations, with a fitness of 15.95. The phenotype has a complexity of (2, 5) and is shown in the next figure :

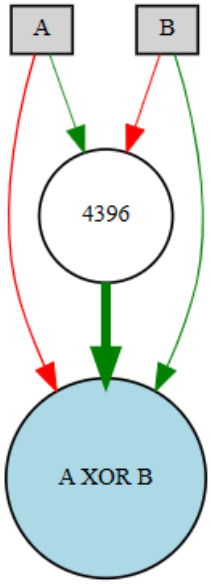

Figure 8: The phenotype found for solving the XOR problem

We can observe that this phenotype is close to the objective one described in Figure 7 And the next table shows the performances of this phenotype :

Input 1	Input 2	Output
0	0	1.1899e-05
1	0	1.0
0	1	0.9495
1	1	1.4048e-06

The next figures show the evolution of average fitness and of species throughout the evolution :

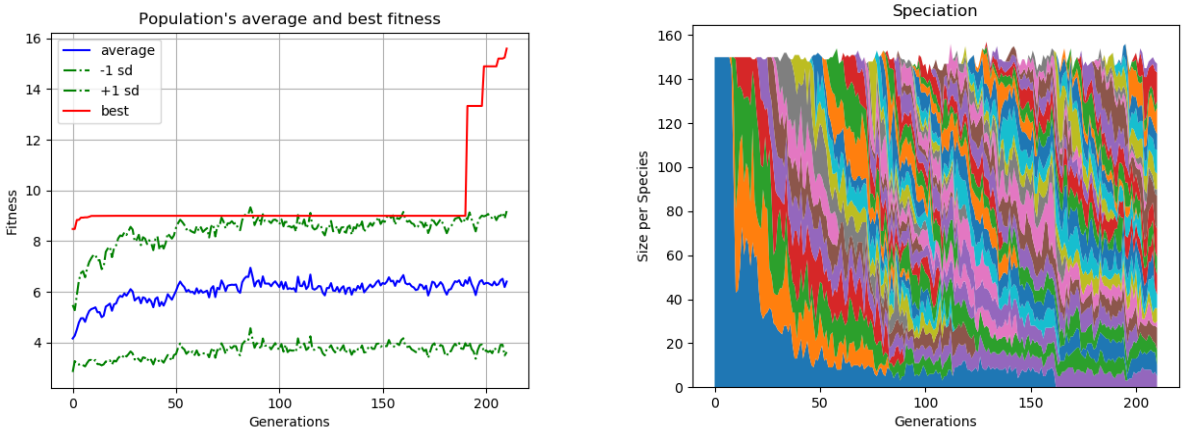

Figure 9: Evolution of fitness and species during a XOR solving ANN evolution

These two graphs show how the fitness of genomes and species have evolved during the process. We see that starting with one species, other ones appear and tend to be more effective, so the old ones get extinct. This cycle makes the average fitness grow, along with the apparition and extinction of new species, until one species (here species number 62) appears and manage to solve the task.

1.6 The maze experiment with NEAT

Now that we understood how to make the NEAT algorithm work, it is time to try bigger things, and to be introduced to our first important experiment : the maze. This experiment will introduce the deceptive nature that such a problem can have, with potentially very strong local maximum. The other complication is that we are going to have to implement a simulator of maze-navigating robot, with many geometry notions at stake, and many sensors and actuators to configure. Then, we will have to define and implement a goal-oriented fitness, to guide the process of creating an appropriate maze solver using the NEAT algorithm.

The deceptive nature of the problem

In such a problem, the goal-oriented fitness can have steep gradients of fitness score that lead to dead ends. The maze problem illustrates perfectly this complication :

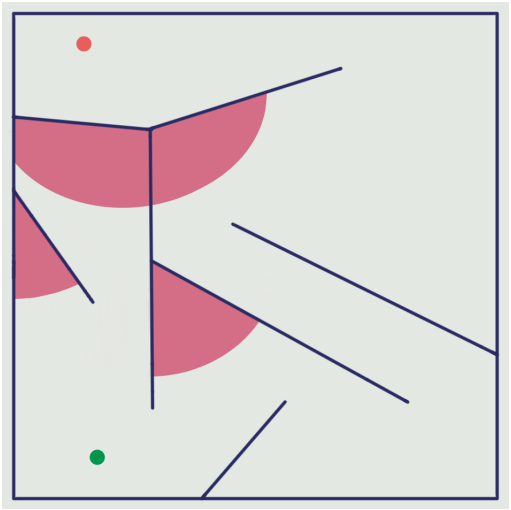

Figure 10: A two-dimensional maze, with its deceptive local maximum in dead ends

The maze-navigating agent

Our agent navigating from the starting point (bottom circle) to the exit point (to circle), is a robot equipped with a set of sensors allowing it to detect nearby obstacles and get the directions possible. There are two types of sensors :

- six rangefinder sensors, that indicate the distance to the nearest obstacle in a given direction. They are represented by the blue arrows in the following figure.
- four pie-slice radar sensors, that act as a compass towards the goal point. With them, the agent will be able to know in which direction is the maze exit, with a precision of 90°.

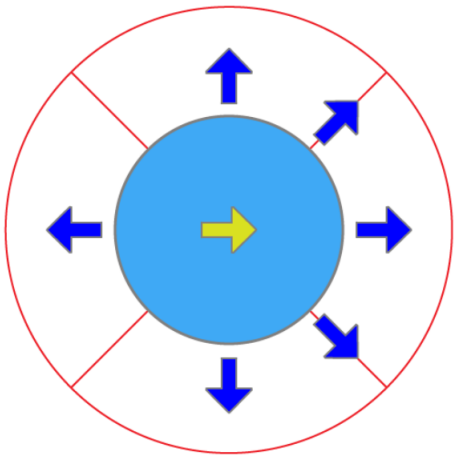

Figure 11: The schema for our maze agent

The **Field Of View and direction** of each sensors are given in the book describing the experiment (1). To implement this feature, we create an `Agent` class, that holds every information related to the maze navigator agent that is used by the simulation. These informations are :

- heading
- speed
- angular velocity
- radius
- rangefinders range
- location
- rangefinders angles
- radars angles and FOV
- a list to hold rangefinders activation
- a list to hold radars activation

The class is implemented in Annexe 4 : Maze NEAT experiment Agent class.

The maze simulation environment

We also need to define an environment that manages the configuration of the maze, tracks the position of the agent, and provides inputs to the agent's sensors. Our maze is described in a text file, looking like this :

```
11
30 22
0
270 100
5 5 295 5
295 5 295 135
295 135 5 135
5 135 5 5
241 135 58 65
114 5 73 42
130 91 107 46
196 5 139 51
219 125 182 63
267 5 214 63
271 135 237 88
```

Figure 12: The text file describing the maze

A method, implemented in the `MazeEnvironment` class, will read this file as follows :

1. first line : number of walls in the maze
2. second line : agent's starting position
3. third line : initial heading of the agent in degrees
4. fourth line : the maze's exit position
5. following lines : walls starting and ending points

All these features fit into the `MazeEnvironment` instance, with the following fields :

- walls
- exit point
- exit range
- agent instance
- a boolean exit_found
- the initial distance to the exit

The class is implemented in Annexe 5 : Maze NEAT experiment MazeEnvironment class

Calculation of the fitness score

To compute the fitness score of each genome, we calculate the loss function, which is simply the **Euclidian distance** between the final position of the agent and the maze exit :

$$\mathcal{L} = \sqrt{\sum_{i=1}^{2} (a_i - b_i)^2}$$

We can now compute the normalized fitness score (between 0 and 1) as follows :

$$\mathcal{F}_{\text{n}} = \frac{\mathcal{L} - D_{\text{init}}}{D_{\text{init}}}$$

with D_{init} being the initial distance.

We can now compute our final fitness score :

$$\mathcal{F} = \begin{cases} 1.0 & \mathcal{L} <= R_{\text{exit}} \\ 0.01 & \mathcal{F}_n <= 0 \\ \mathcal{F}_n & \text{otherwise} \end{cases}$$

With this calculation, our fitness score is a score between 0.01 and 1, the first case being if the agent ended more far away that it actually was at the starting point, 1 being if the agent found the exit point.

This calculation is implemented in the `evaluate_genomes()` function, which purpose was described in 1.4, but it requires many methods implemented in the `Agent`, `AgentRecord` (implemented in the same file as `Agent` class), and `MazeEnvironment` classes.

Running the experiment

You can find all the source codes needed for the run of this experiment in :

- `Agent` class : Annexe 4 : Maze NEAT experiment Agent class
- `MazeEnvironment` class : Annexe 5 : Maze NEAT experiment MazeEnvironment class
- the main code file : Annexe 6 : Maze NEAT experiment
- the configuration file :Annexe 7 : Maze NEAT experiment configuration file
- visualization methods : Annexe 8 : Maze NEAT experiment visualization methods
- geometry methods : Annexe 9 : Maze NEAT experiment geometry methods
- utils methods : Annexe 10 : Maze NEAT experiment utility methods
- a text file describing a medium maze : Annexe 11 : text file decribing a medium maze
- a text file describing a hard maze : Annexe 12 : text file describing a hard maze

With the medium maze

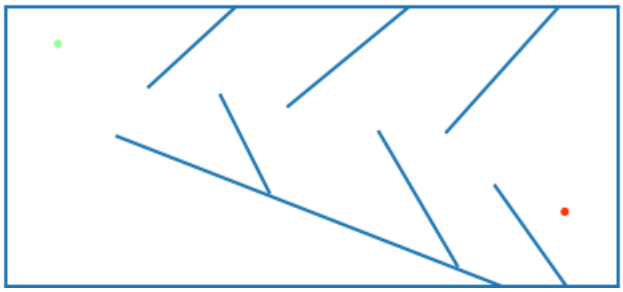

Figure 13: The medium maze configuration

After a few tries with the medium maze, we finally find a successful genome at generation 144, after 2348 seconds (40 minutes) of running, with a complexity of (2, 10) :

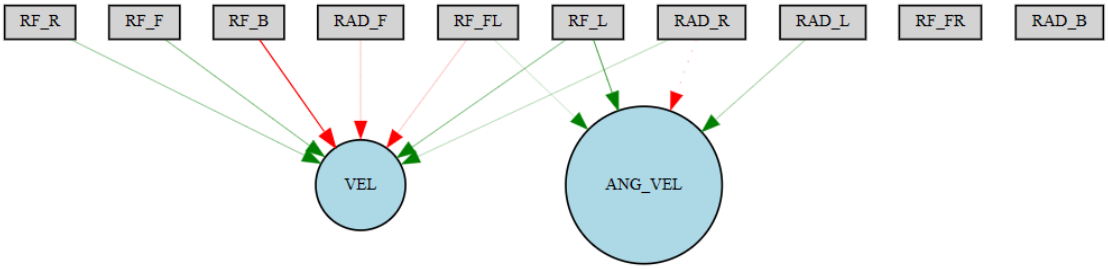

Figure 14: The phenotype found for solving the maze experiment via NEAT

Let's take a look at our evolution graphs :

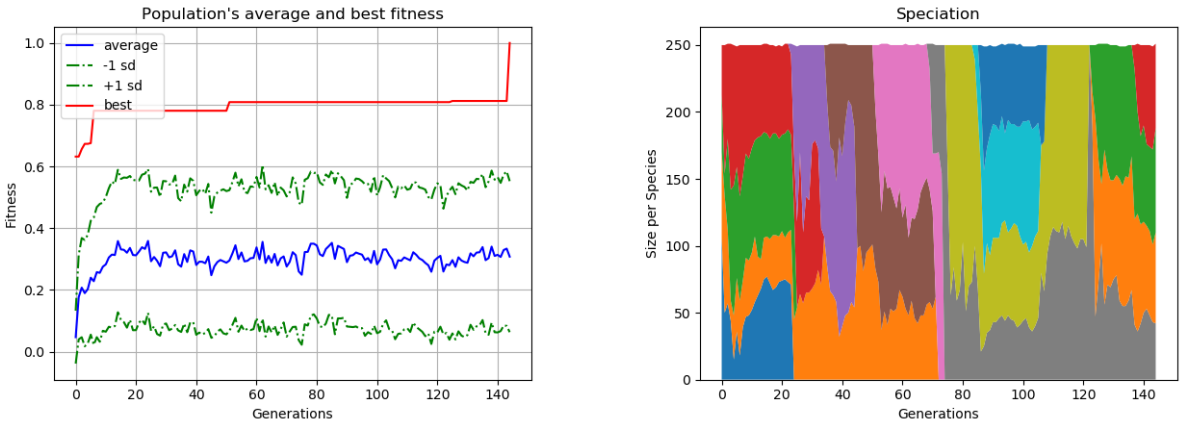

Figure 15: Evolution of fitness and species during this experiment via NEAT

We see the same patterns as we did with the XOR experiment of 1.5. The species are lasting longer, due to the difference in some hyperparameters. It is interesting to look at our phenotypes and see the role of each sensors. We can even see that two radars out of four are completely omitted. This is because our solver is not optimal at all. We stopped the process as soon as one agent could find a solution, but it is not meant to be the shortest solution. If we wanted this, we could have continued our experiment, and configure a loss

function proportional to the number of steps needed by our solver. In this research, our goal is to compare the performances between a goal-oriented function and a novelty score, so there is no need to have an optimal solver.

With the hard maze

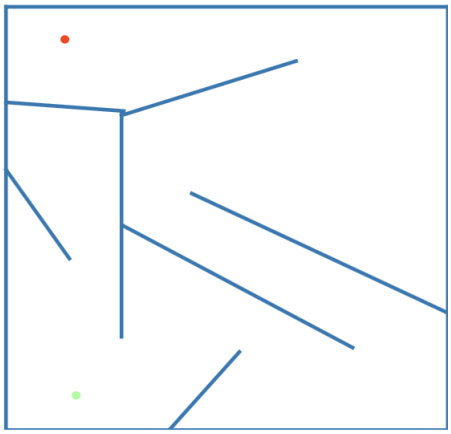

Figure 16: The hard maze configuration

The hard maze shows the limits of the NEAT algorithm. Even with 500 generations, it is impossible for the NEAT algorithm to solve this problem, because there are too many local optimum.

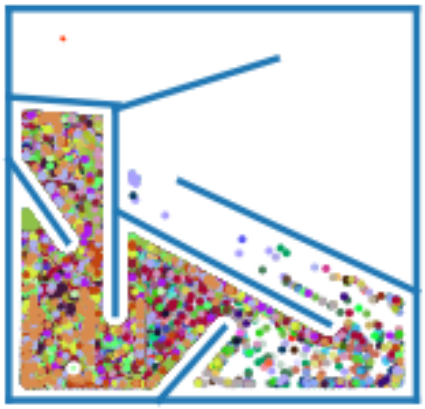

Figure 17: The optimum wells of the hard maze

This figure shows how our NEAT algorithm is helpless against the problem of local optimum. No agent is able to get around the walls to find the solution, because it would represent at first losing a lot in the fitness score, and these agents would not get through the natural selection process anymore. To solve this issue, we will need an improvement in the NEAT goal-oriented algorithm.

2 Part 2 : The Novelty Search algorithm

The most of this part is drawn from chapters 1 and 6 of (1). Chapter 1 gives a first general insight about the Novelty Search method, and its theory, and Chapter 6 takes the problem of maze navigating that we solved with a NEAT algorithm in the previous section, and compares it with the Novelty Search method.

2.1 Novelty Search general insight

In most evolutionary algorithms, fitness of the solver is measured by the closeness to the goal. Novelty Search introduces a different method. While direct fitness function optimization methods can work well in many simple cases, it can also fall victim to the local optima trap. The traditional NEAT algorithm can rely only on mutation to escape such a trap, but this method can sometimes be helpless, in deceptive problems, or it can take too long to find a successful solution. However, a solution can always be found by looking into **diversity**. In other words, any evolving species gains immediate evolutionary advantages over its rival by finding new behavior patterns. This force of evolutionary diversity is a **search for novelty**.

From this observation, Joel Lehman proposed a new method of search optimization for an artificial evolutionary process called **Novelty Search**, described in Revising the Evolutionary Computation Abstraction: Minimal Criteria Novelty Search (2). With this method, no fitness function is required to guide the evolution for solution search. Instead, the novelty of each solution is directly rewarded during the evolution process. It is no more the closeness to the goal that guides neuroevolution, but the novelty of solutions. Such an approach gives a chance to exploit the creative force of diversity independently of the pressure to fit a solution in a niche.

In the next section, we will demonstrate the effectiveness of Novelty Search with the maze navigation experiment, where an objective-based search can easily lead into deceptive local optimum.

2.2 The maze experiment with Novelty Search

The main idea is to simply replace the fitness function that we used in the NEAT experiment, and replace it by a function able to calculate the novelty score of each genome. For this reason, the source codes will be very similar to the ones of 1.6. Now, we have to define the novelty metric that can capture the amount of novelty in a particular solver agent. There are mainly two ways to calculate the novelty of a solver agent :

- the **structural** novelty : the novelty of the solver's genotype structure
- the **behavioral** novelty : the stepping stones found in the search space of the solution, how our robot moves in the maze.

In this research, we decided to focus on the behavioral novelty, because our primary interest is to create a successful maze navigator. In order to achieve that, it is in our interest that the robot explores every places in the maze.

The novelty score

To measure the novelty of the agent, the most straightforward way is to measure the sparseness at any point of its trajectory, which is the **average distance from this particular point to the k-nearest neighbors**.

$$\rho(x) = \frac{1}{k}\sum_{i=0}^{k} \text{dist}\,(x, \mu_i)$$

with μ_i being the i^{th} nearest neighbor of x, as calculated by the **novelty metric** : $\text{dist}\,(x, y)$.

This novelty metric is the difference in the behaviour of the two agents, determined by the item-wise distance between the two trajectory vectors. You can calculate this novelty metric with the following formula :

$$\text{dist}(x, \mu) = \frac{1}{n}\sum_{j=0}^{n} |x_j - \mu_j|$$

Fortunately, we didn't have to implement the calculation of the novelty score, and definition of novelty metric, because it is already described in (1)

Now, this score can directly be used as a fitness score in the NEAT algorithm that we described in 1.6. By changing the fitness score in a novelty score, the neuroevolution process tries to maximize the novelty of the produced individuals, which results in the agents better exploring the whole area. However, we still have to use the goal-oriented fitness score to test whether or not our agents did find the exit of the maze. Also, this value will still be recorded for visualization in similar graphs as the ones we presented for the XOR experiment and the maze with NEAT (figures 9 and 15).

Running the experiment

You can find all the source codes needed for the run of this experiment in the following Annexes. Most of them are very similar to the ones used with NEAT in 1.6, but two new classes were needed to compute the novelty score : `NoveltyArchive` and `NoveltyItem`. There is also new methods of visualization and the possibility to run many trials at a time, and have statistics about the global run.

- the main code : Annexe 13 : Maze NS experiment
- `Agent` class : Annexe 14 : Maze NS experiment Agent class
- `MazeEnvironment` class : Annexe 15 : Maze NS experiment MazeEnvironment class
- the configuration file : Annexe 16 : Maze NS experiment configuration file
- `NoveltyArchive` class : Annexe 17 : Maze NS experiment NoveltyArchive class
- `TrialsArchive` class : Annexe 18 : Maze NS experiment TrialsArchive class
- general visualization methods : Annexe 19 : Maze NS experiment Visualization methods
- one-genome visualization methods : Annexe 20 : Maze NS experiment visualization of one particular genome methods

- a text file describing a medium maze : Annexe 11 : text file decribing a medium maze
- a text file describing a hard maze : Annexe 12 : text file describing a hard maze
- geometry methods : Annexe 9 : Maze NEAT experiment geometry methods
- utils methods : Annexe 10 : Maze NEAT experiment utility methods

With the medium maze

A new method implemented with the aid of the `TrialsArchive` class, gives us that, for 10 trials, the algorithm was able to find a successful solver 9 times, which is more than with NEAT, where we had to run the process 2 or 3 times between each successful run. Moreover, this method gives us that for these 9 trials that succeeded, the average of generations needed is 126, which is less than what we had with NEAT. Thus, Novelty Search is capable to have better results more often and more quickly than NEAT, and we are only talking about the medium maze, where local optimum were not too strong. This shows that even on regular problems, Novelty Search could be a substitute for NEAT, and there is no need to have a deceptive problem to implement this method.

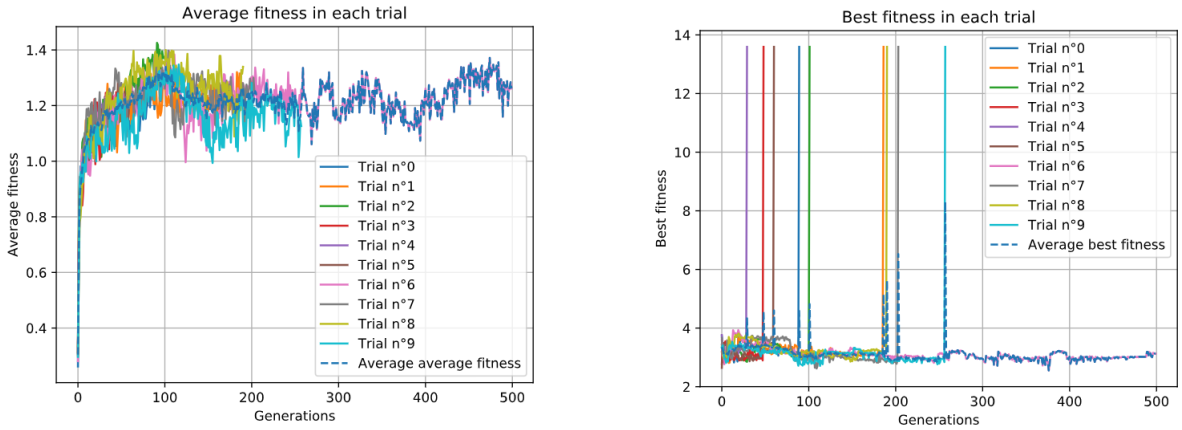

Figure 18: Average and best fitness in 10 trials of the maze experiment with Novelty Search

The peaks that we see on the right-hand side figure show where did the trial found their solution. In the code, we had to set an arbitrary fitness for when the agent found the exit point. For it to be high enough to be sure that it is not just a very novel point, we arbitrary put this threshold to 13.5. With these graphs, we can also see that the novelty score doesn't get lower, which means that during every generations, the algorithm explores new areas, until the area of the exit point is successfully found.

We are now going to go through the results of the first trial, that solved the maze in 84 generations, with a phenotype complexity of (2, 9), which is slightly better than what we had with NEAT, but also quicker.

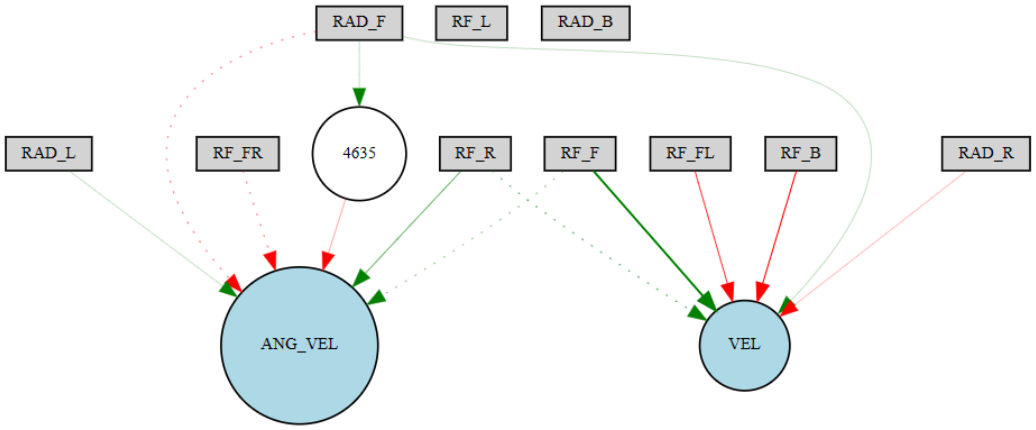

Figure 19: The phenotype found for the maze experiment via Novelty Search

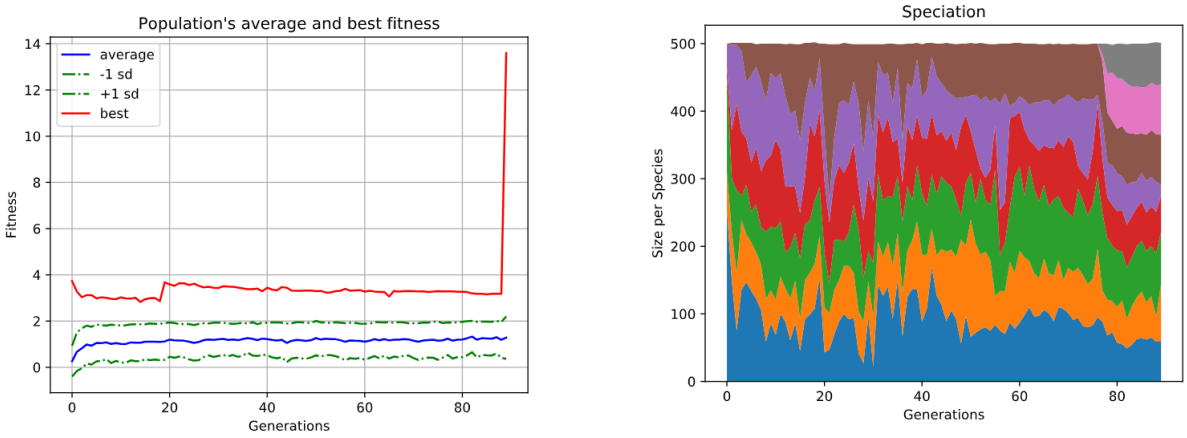

Figure 20: Evolution of fitness and species during this experiment via Novelty Search

We can also see in the next figure that the algorithm has searched every places in the maze, until it finally found the exit point. This algorithm allows the process to search in every direction, contrarily to the traditional NEAT algorithm that was guided, sometimes wrongly, by the fitness score.

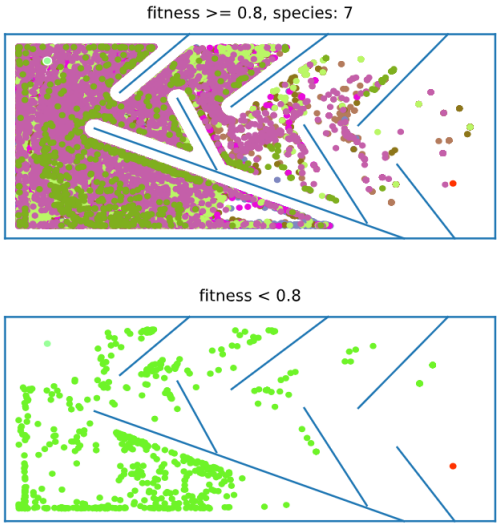

Figure 21: The record of every final positions the NS algorithm wen through

Also, it is important to note that seven of eight total species created during the evolutionary process demonstrate the highest goal-oriented fitness scores, that means they were all almost able to reach the maze exit.

Finally, the most exciting visualization allows us to look at the path of the successful maze solver agent :

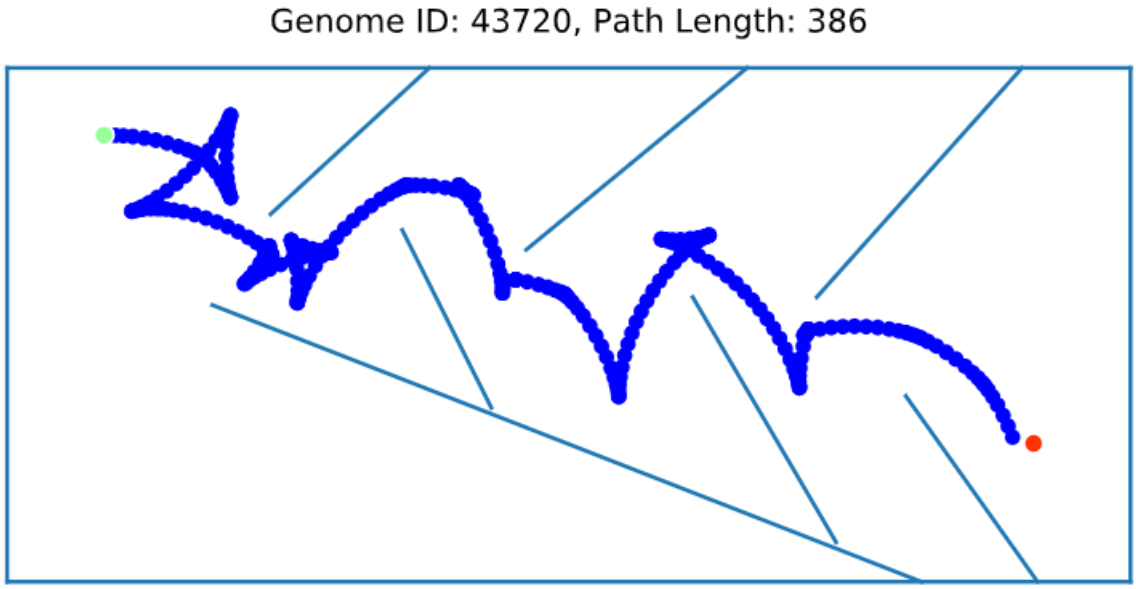

Figure 22: The path of the successful maze solver agent

Once again, our goal is not to find a quick and smart path, but only to find the first one to reach the exit in less than 400 steps. It is normal that the first solver is not a straight line and has some patterns that seem unintelligent. This can also be seen as the consequence of

starting with minimal phenotypes, as the development into bigger ones is a complex processus that takes many generations, and it is not what we seek.

With the hard maze

The Novelty Search algorithm was also not able to solve the hard maze. However, the results are still more interesting than the ones we had with NEAT algorithm.

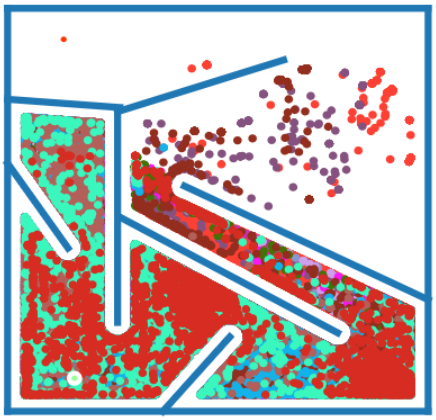

Figure 23: The Novelty Search exploration in the hard maze

When we compare this figure to the one that we obtained with NEAT algorithm (figure 17), we can see that the Novelty Search algorithm has explored more deeply the maze. There are even three points that are very close to the exit point. This observation makes us think that with more generations, it would be possible to find a successful maze solver agent with Novelty Search, although such thought would be irrational for the NEAT algorithm.

Overall, we can affirm that Novelty Search is more powerful than NEAT on the maze problem. With the medium maze, it had better results in less generations. In the hard maze, it was much closer to find a successful solution than the NEAT algorithm could never be. It is even possible that, by adjusting the hyperparameters, or allowing more generations, the Novelty Search approach could find a solution, without having any sense of closeness to the goal, but by being guided only by novelty and search for new places.

3 Part 3 : The orientation consensus problem

After having dealt with NEAT and Novelty Search algorithm implemented on a maze problem, Prof. Urbano wanted to take the research further, by continuing this approach of comparison between these both, on another problem. He also wanted to see what would be a minimal and effective way of communication between robots. For this purpose, we interested ourselves in the problem of orientation consensus. For this part, our main resource will be the paper of Rafael Sendra-Arranz and Alvaro Gutiérrez, Emergence of Communication Through Artificial Evolution in an Orientation Consensus Task in Swarm Robotics (3). The paper describes the experiment in all its details, with the mean of communication, inputs and outputs of the robots controller. In the following sections, we will reproduce this experiment with the algorithms seen in the previous sections 1.6 and 2.

3.1 Description of the orientation consensus problem

As it is explained in Chapter 4 of (3), orientation consensus refers to the task in which all the robots in a **swarm** have to point to the same direction. The positions of the robots are fixed, and their orientation must converge, only by means of rotation movements, either clockwise or counterclockwise, at an angular speed modulated by their corresponding neural controller. But robots don't have access to any absolute sensing reference, however they can communicate to gain information about relative headings of their neighbors.

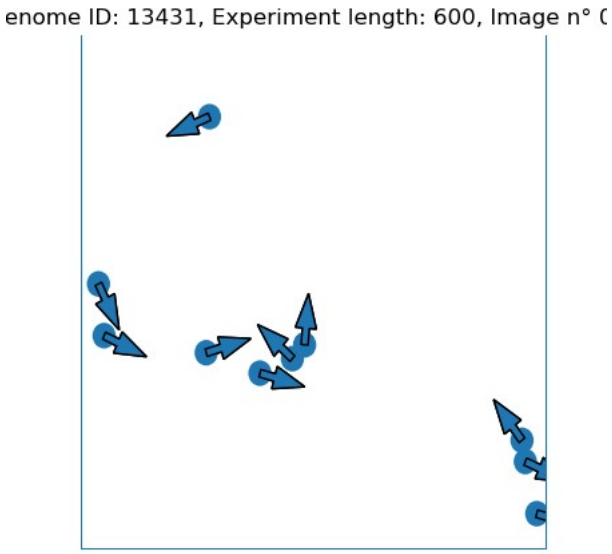

Figure 24: The random disposition of robots in an orientation consensus problem

3.2 The communication system

The communication is based on what is proposed in (6). It is a minimal IR-based minimal communication system with a local and constrained communication range of 80cm (the map

is a 100cm-square, so that it is not too constraining in our case). The robots can only perceive a single message at each time step of the simulation, from one of their sensors (see e-puck robot figure 25). The information received not only comprises the abstract message content, but also the relevant context information about the environment (which sensor did receive the message, and from which sensor of the emitter the message was sent). The robot's controller, fed by both the received message and its associated context, elaborates a new two-dimensional message to be broadcasted. This message is subject to a quantization mapping, that converts the raw message into one of the symbol in the set $\mathcal{C}$ defined as :

$$\mathcal{C} = \left\{0, \frac{1}{3}, \frac{2}{3}, 1\right\}^2$$

The communication of the robots can either be in SEND MODE, transmitting its own created message, or in RELAY MODE, by emitting a copy of the message received from other robots.

The model of robots on which our simulation will be based are **E-Pucks**. The placement of their sensors is described in the following figure :

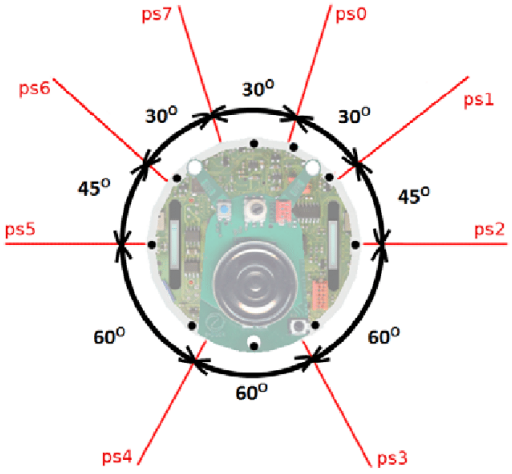

Figure 25: The sensors configuration of an E-Puck robot

The following figure illustrates the principle of relative heading between two robots, when one receives the signal of another.

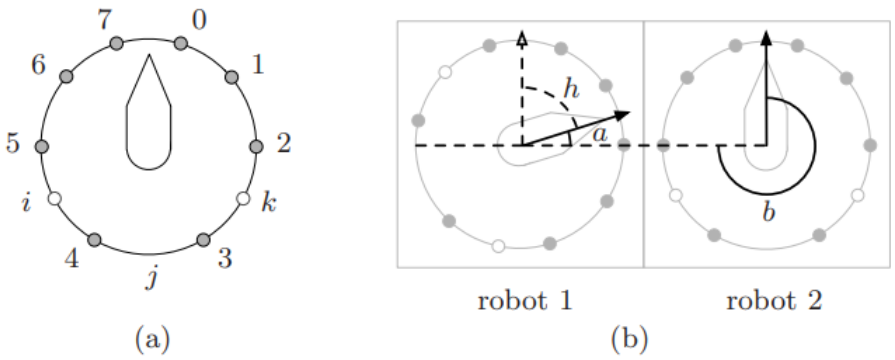

Figure 26: Relative heading between two robots during a communication between both

3.3 Running the experiment via NEAT algorithm

The fitness function

The fitness function given in Chapter 4.2 of (3) can be directly used for our NEAT algorithm. It is composed by two terms that are merged in a multiplicative way. The following equation gives the fitness score of a single agent r at time step t :

$$f(t,r) = \left(1 - \frac{\min\left\{2\pi - \left|\theta_r(t) - \bar{\theta}(t)\right|, \left|\theta_r(t) - \bar{\theta}(t)\right|\right\}}{\pi}\right) \cdot (1 - |a_{wr}(t)|)$$

The first term measures the orientation deviation or misalignment of the robot with respect to $\bar{\theta}(t)$ being the mean orientation of the swarm :

$$\bar{\theta}(t) = \arg\left(\sum_{r\in\mathcal{R}} e^{j\theta_r(t)}\right)$$

This term will increase as the orientation of the robot r tends to the mean orientation of the swarm. The second term rewards the robot for reducing its rotation velocity, a_{wr} being the signal that controls speed and sense of rotation.

To obtain the total fitness score of a genome, we compute the sum of all its robots individual fitness at the final time step :

$$F_{tot} = \frac{1}{R}\sum_{r\in\mathcal{R}} f(t_\infty, r)$$

Running the experiment

All the source codes necessary to run this experiment are given in the following annexes. We implemented this algorithm from scratch, but it was greatly inspired from the NEAT algorithm applied to the maze problem. The same classes, referred in 1.6 are used. We also had to add some geometry methods and many other to compute the fitness score. Also, the most prolonged part during the implementation was to create the mechanic of the simulation. Every steps that happens during each time steps are complex, and a perfect understanding of the NEAT algorithm is required.

- the main code : Annexe 21 : Orientation Consensus experiment
- `Agent` class : Annexe 22 : Orientation Consensus experiment Agent class
- `ConsensusEnvironment` class : Annexe 23 : Orientation Consensus experiment ConsensusEnvironment class
- the configuration file : Annexe 24 : Orientation Consensus experiment configuration file
- visualization methods : Annexe 25 : Orientation Consensus experiment visualization methods
- geometry methods : Annexe 26 : Orientation Consensus experiment geometry methods
- utils methods : Annexe 10 : Maze NEAT experiment utility methods

Without the message

We realized a first experiment where we omitted the two-dimensional message. The only information that was communicated between robots was their relative heading. Thus, their is only two inputs, the two relative angles, and one output, the angular velocity.

A solution was found after only 20 generations. The following figure shows the phenotype of the solver :

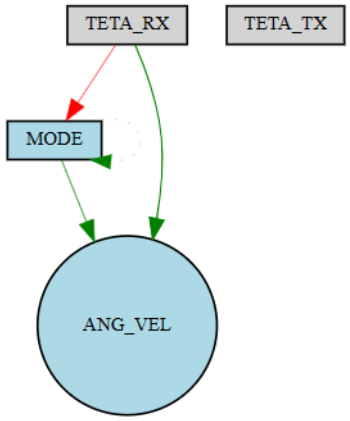

Figure 27: The phenotype of the solving ANN

The MODE neuron was an oversight and should not be here, however, we can see that it took the role of a hidden node. We can see that the phenotype is very minimalist, which is logical considering that the process was stopped at the 20^{th} generation. New nodes didn't have time to appear, but the ANN was capable of optimizing itself quickly as it is very low-dimensional.

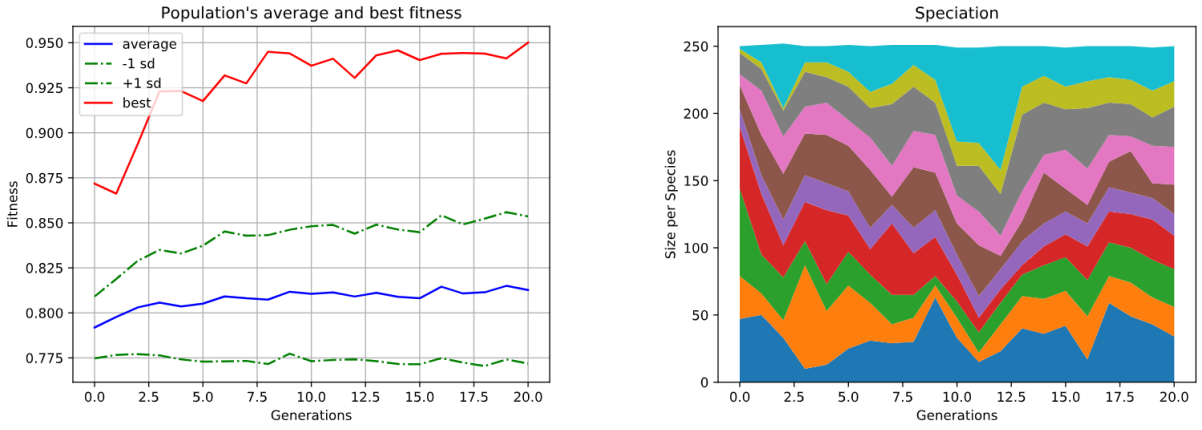

Figure 28: Fitness and species evolution during the orientation consensus without the message

Finally, we have the proof that the experiment was a success, through a visualization method that creates a video of the best genome's behaviour in this experiment. The following figures are two snapshots from the beginning and the end of the experiment :

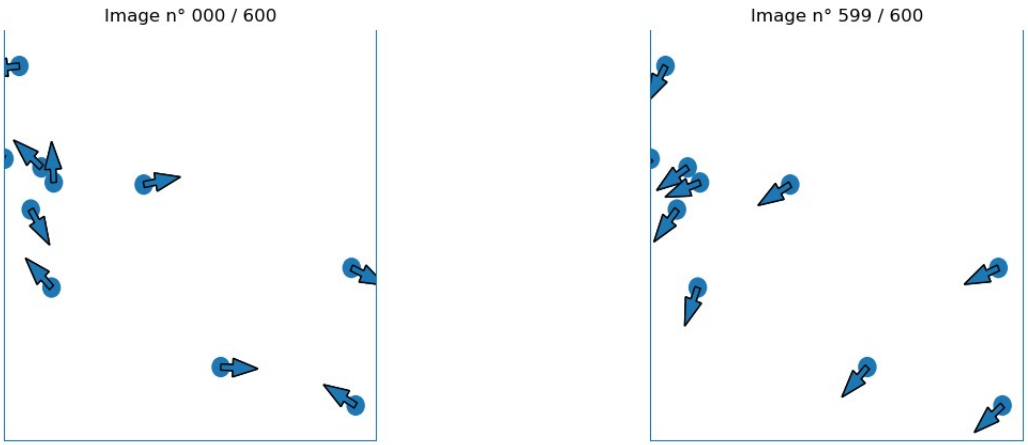

Figure 29: Initial and final orientations of the robots controlled by the best genome's ANN

We can see the initial position, where all robots are disposed randomly on the map, heading to random angles, and after 600 steps of running, all these robots are almost aligned. The alignment isn't perfect, and they still have an angular speed, however it was interesting in the video to see them align by making many rotation, and slightly reducing their speed in the last few steps.

At the time of this first introduction experiment, all visualization methods were not implemented, but we still have the video of the experiment, that shows every robots pivoting on themselves, and reaching more or less a consensus.

We also remarked afterward that there was a mistake in the fitness calculation. Fortunately, this mistake was minor and it didn't ruin the experiment, but it only explains why we would have such high fitness quickly. In the next experiment, to compensate with this mistake, the fitness threshold will be lowered from 0.95 to 0.80 (1.0 being a perfect consensus with every robot freezed).

With the message

With the message, the solver ANN wasnfound after only 6 generations, and is already much more complex (6, 13) :

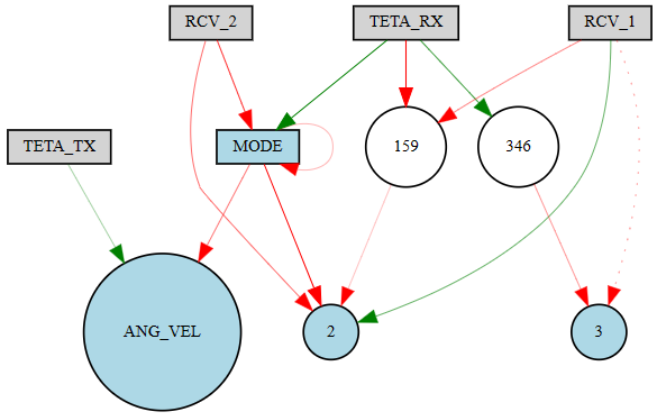

Figure 30: The phenotype of ANN solving the orientation consensus problem with a direct mean of communication

We can see that some nodes are link to themself, or to previous ones. This is because we authorized non-**feed-forward** links in the hyperparameters. This parameter deactivated allows us to have much more complex phenotypes, with fedbacks between the nodes.

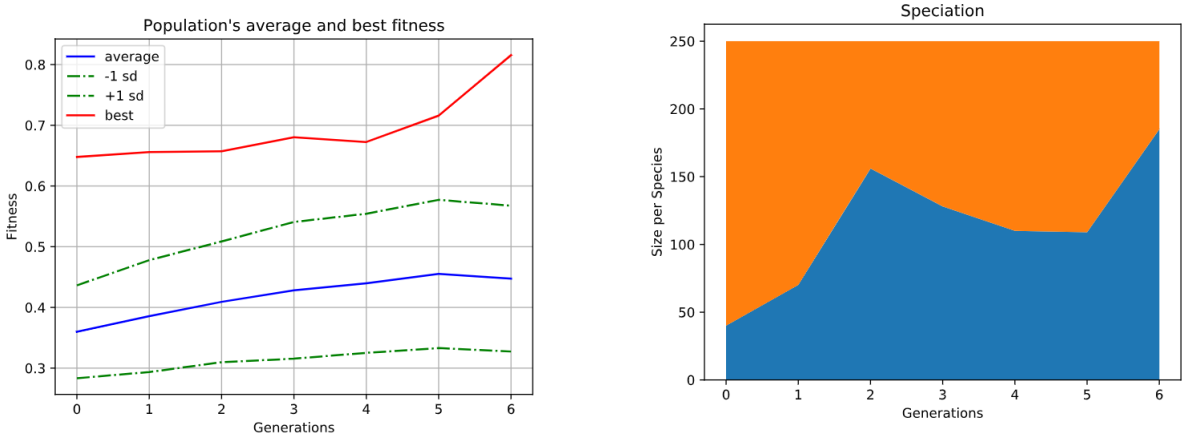

Figure 31: Fitness and species evolution during the orientation consensus with a direct mean of communication

Here, only two species created initially were able to solve the problem. Again, let's see the initial and final snapshots of the experiment :

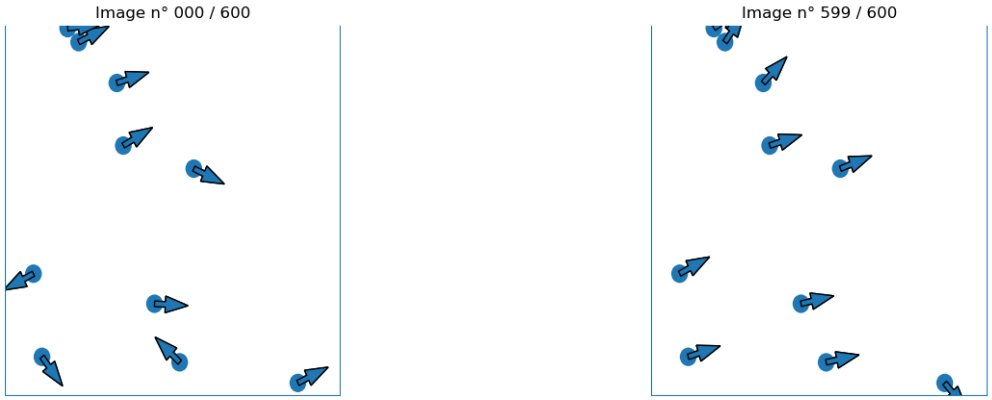

Figure 32: Initial and final orientations of the robots controlled by the best genome's ANN

Except for one robot, they are finally all heading towards the same direction. But what is interesting is that, when we look at the video, we can see that the behaviour is not the same. In the previous experiment, all robots were rotating, and slight changes of angular velocity made them adapt to one another. Here, some robots are completely still, then activate, and stop again. It seems like they are rotating until they have the message that their neighbour is heading towards the same direction. When they receive this signal, it's a brutal stop. Besides, in the end, every robots are motionless. It seems like they understood that fitness was calculated only in the last step, so they all reduce their speed when they are coming to the end.

After having experienced all these implementation complications, we didn't have enough time to implement every visualization methods that we wanted, considering that every run takes more than an hour, it takes a lot of time every time that we want to modify something. However, it is still interesting to look at what results they found in the original paper (3), where they use a different evolutionary algorithm. There, other visualization methods are implemented and allow us to take a deeper look into what happens in terms of communication.

In Chapter 5, the following graph shows that robots only use 2 out of 4 symbols available. More precisely, they seem to generate only `(0, 0)` or `(0.33, 0.33)`, which represent 2 out of 16 two-dimensional symbols available.

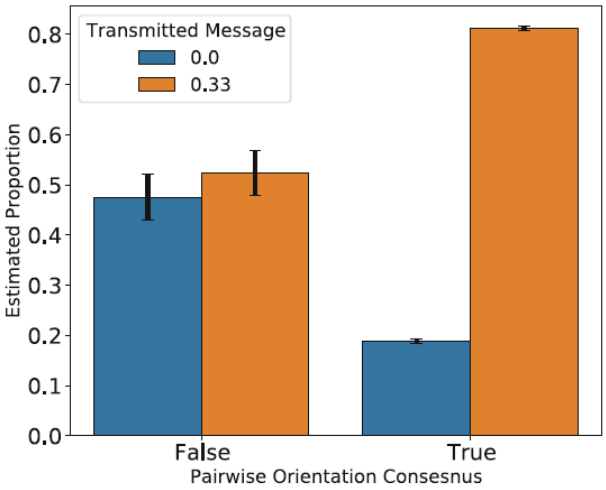

Figure 33: The proportion estimations of each symbol used in communications between robots (3)

The paper (3) tells us that when pairwise orientation consensus between two robots is reached, the symbol 0.33 is mostly generated. Otherwise, both symbols can be emitted. This last piece of information proves that the solver ANN effectively uses communication to reach a consensus, but this communication system is even more minimal than what the experimenter authorised. We can argue that by creating a more complex ANN, with more generations and more computation time, a better communication system would appear with a new species.

Error made and what to change

Unfortunately, the greatness of the task led us to many errors. The biggest one was in the fitness calculation process. We decided to only take into account the placement of the robots in the last time step, and only to verify at each time step if they reached the consensus. But this process led us to never reaching the consensus, and having an evaluation that didn't take into account every position. If we could have find a way to take every time steps of the experiment into account for the fitness calculation, the calculation would maybe take longer, but the result and progression of the evolution of our ANN would be better guided. That would have led us to a better final ANN, maybe capable of solving this problem entirely.

3.4 Running the experiment via Novelty Search algorithm

One of the main goal of this research was to find out if a Novelty Search approach would also be more effective on a problem like orientation consensus. However, after experiencing some problems with the NEAT algorithm approach, and having to code everything from scratch, it took more time that we thought. After that, we ran out of time and couldn't implement the Novelty Search method. There is no doubt that Prof. Urbano will continue his research on this subject in the next weeks.

Conclusion

Throughout my internship, I learned a lot of things concerning **neuroevolution** and more precisely **NEAT** and **Novelty Search** algorithms. I learned the mechanics behind these evolution processes, the **operators** involved in natural evolution that we want to virtually recreate, how to implement these types of algorithms with Python using the NEAT-Python library. Then I was able to manipulate many different experiments, using both of these algorithms, and see practical results thanks to some visualization methods, mostly implemented by myself. Finally, I was able to create such an experiment entirely by myself, and see how it is tricky to predict every difficulties, and how to manage every problems I encountered. Finally I was caught up by the time, and couldn't finish everything I wanted to do, even though I am still proud and happy of what I could do.

We saw in 2 that the **Novelty Search** algorithm, normally used in problems having a deceptive nature, can also work very well, even better than traditional NEAT, in common problems, here represented by the medium maze experiment 2.2. We saw with the hard maze experiment 2.2, that it is also effectively more adapted to deceptive problems, because of its exploratory nature.

Then, by implementing from scratch the **orientation consensus** experiment 3, we were able to manipulate a swarm of robots able to communicate between themselves. Even if the observation of this problem is far from complete in this paper, we still were able to see some interesting results, and draw conclusions about the minimal communication system used by maybe the most compact form of intelligence possible.

Finally, we were supposed to cross the parts between Novelty Search and communications, to see if a novelty-based evolution would change something of the communication used between robots, but after having experienced many difficulties in the implementation of the experiment with NEAT, we have been running late on the schedule. It is slightly frustrating to not have these results, but there is no doubt that Prof. Urbano will continue the research on this path.

I really enjoyed this internship, and most of all the autonomy that I was given. It was a source of motivation and I was driven during three months by wanting to have better results and adding new features to the experiments. For that, I would like once again to thank Prof. Urbano for his trust and his counselling throughout the internship.

Planning of the internship

May 15th → May 26th	learning with Hands-On NeuroEvolution with Python (1)+ use of NEAT algorithm on simulated robots in a maze problem
May 29th → June 30th	learning about Novelty Search and implementation of visualization methods for the maze problem
July 3rd → August 4th	learning about the orientation consensus problem with Emergence of Communication Through Artificial Evolution in an Orientation Consensus Task in Swarm Robotics (3) and implementation from scratch of a NEAT algorithm on this problem
August 7th → August 18th	Redaction of this report and last adjustments of the results of many experiments

Glossary

Deep-Learning : a type of Machine-Learning based on Artificial Neural Networks in which multiple layers of processing are used to extract progressively higher level features from data.

Neuroevolution : a machine learning technique that applies evolutionary algorithms to construct artificial neural networks, taking inspiration from the evolution of biological nervous systems in nature.

ANN : Artificial Neural Network

NEAT : NeuroEvolution of Augmenting Topologies. A genetic algorithm for the generation of evolving artificial neural networks

Fitness : a measure of the quality of a machine learning algorithm. It is typically calculated by comparing the algorithm's predictions to the actual outcomes of a dataset. The fitness score is a numerical value that indicates how well the algorithm performed on the dataset.

Genome : a simplified genetic representation that can be mapped to a neural network.

Swarm Robotics: an approach to the coordination of multiple robots as a system which consist of large numbers of mostly simple physical robots.

Bibliography

[1] Iaroslav Omelianenko, *Hands-On Neuroevolution with Python*, Packt Publishing Ltd, 2019, https://www.packtpub.com/product/hands-on-neuroevolution-with-python/9781838824914

[2] Joel Lehman, Kenneth O. Stanley, *Revising the Evolutionary Computation Abstraction: Minimal Criteria Novelty Search*, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 2010, https://dl.acm.org/doi/abs/10.1145/1830483.1830503

[3] Rafael Sendra-Arranz, Alvaro Gutiérrez, *Emergence of Communication Through Artificial Evolution in an Orientation Consensus Task in Swarm Robotics*, E.T.S. Ingenieros de Telecomunicación, Universidad Politécnica de Madrid, 2023, https://link.springer.com/chapter/10.1007/978-3-031-34107-6_41

[4] Alvaro Gutiérrez, Elio Tuci, Alexandre Campo, *Evolution of Neuro-Controllers for Robots' Alignement using Local Communication*, International Journal of Advanced Robotic Systems, Vol 6, No 1, 2009, https://journals.sagepub.com/doi/full/10.5772/6766

[5] Ramos RP, Oliveira SM, Vieira SM, Christensen AL, *Evolving flocking in embodied agents based on local and global application of Reynolds' rules*, Instituto Superior Técnico (IST), 2019, https://doi.org/10.1371/journal.pone.0224376

[6] Sendra-Arranz R, Gutiérrez A, *Evolution of Situated and Abstract Communication in Leader Selection and Borderline Indentification Swarm Robotics Problems*, E.T.S. Ingenieros de Telecomunicación, Universidad Politécnica de Madrid, 2021, https://doi.org/10.3390/app11083516

[7] Jozef Kelemen, Petr Sosík, *Avances in Artificial Life* 6th European Conference, ECAL 2001, https://books.google.fr/books?id=NRd6GIvgcQEC&lpg=PP1&hl=fr&pg=PP1#v=onepage&q&f=false

Annexe

Annexe 1 : XOR configuration file

```
#--- Hyper-parameters for the XOR experiment ---#

[NEAT]
fitness_criterion     = max
fitness_threshold     = 15.5
pop_size              = 150
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.0
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full_direct

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 2
num_outputs             = 1
```

```
52
53 # node response options
54 response_init_mean      = 1.0
55 response_init_stdev     = 0.0
56 response_max_value      = 30.0
57 response_min_value      = -30.0
58 response_mutate_power   = 0.0
59 response_mutate_rate    = 0.0
60 response_replace_rate   = 0.0
61
62 # connection weight options
63 weight_init_mean        = 0.0
64 weight_init_stdev       = 1.0
65 weight_max_value        = 30
66 weight_min_value        = -30
67 weight_mutate_power     = 0.5
68 weight_mutate_rate      = 0.8
69 weight_replace_rate     = 0.1
70
71 [DefaultSpeciesSet]
72 compatibility_threshold = 3.0
73
74 [DefaultStagnation]
75 species_fitness_func = max
76 max_stagnation       = 20
77 species_elitism      = 2
78
79 [DefaultReproduction]
80 elitism            = 2
81 survival_threshold = 0.2
82 min_species_size   = 2
```

Annexe 2 : XOR experiment source code

```
#
# This file provides source code of XOR experiment using on NEAT-Python
    library
#

# The Python standard library import
import os
import shutil
# The NEAT-Python library imports
import neat
# The helper used to visualize experiment results
import visualize

# The current working directory
local_dir = os.path.dirname(__file__)
# The directory to store outputs
out_dir = os.path.join(local_dir, 'out')

# The XOR inputs and expected corresponding outputs for fitness evaluation
xor_inputs  = [(0.0, 0.0), (0.0, 1.0), (1.0, 0.0), (1.0, 1.0)]
xor_outputs = [   (0.0,),     (1.0,),     (1.0,),     (0.0,)]

def eval_fitness(net):
    """
    Evaluates fitness of the genome that was used to generate
    provided net
    Arguments:
        net: The feed-forward neural network generated from genome
    Returns:
        The fitness score - the higher score the means the better
        fit organism. Maximal score: 16.0
    """
    error_sum = 0.0
    for xi, xo in zip(xor_inputs, xor_outputs):
        output = net.activate(xi)
        error_sum += abs(output[0] - xo[0])
    # Calculate amplified fitness
    fitness = (4 - error_sum) ** 2
    return fitness

def eval_genomes(genomes, config):
    """
    The function to evaluate the fitness of each genome in
    the genomes list.
    The provided configuration is used to create feed-forward
    neural network from each genome and after that created
    the neural network evaluated in its ability to solve
    XOR problem. As a result of this function execution, the
    the fitness score of each genome updated to the newly
    evaluated one.
    Arguments:
        genomes: The list of genomes from population in the
                current generation
        config: The configuration settings with algorithm
```

```
                hyper-parameters
    """
    for genome_id, genome in genomes:
        genome.fitness = 4.0
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        genome.fitness = eval_fitness(net)

def run_experiment(config_file):
    """
    The function to run XOR experiment against hyper-parameters
    defined in the provided configuration file.
    The winner genome will be rendered as a graph as well as the
    important statistics of neuroevolution process execution.
    Arguments:
        config_file: the path to the file with experiment
                     configuration
    """
    # Load configuration.
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         config_file)

    # Create the population, which is the top-level object for a NEAT run.
    p = neat.Population(config)

    # Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    p.add_reporter(neat.Checkpointer(5, filename_prefix='out/neat-checkpoint
    -'))

    # Run for up to 300 generations.
    best_genome = p.run(eval_genomes, 300)

    # Display the best genome among generations.
    print('\nBest genome:\n{!s}'.format(best_genome))

    # Show output of the most fit genome against training data.
    print('\nOutput:')
    net = neat.nn.FeedForwardNetwork.create(best_genome, config)
    for xi, xo in zip(xor_inputs, xor_outputs):
        output = net.activate(xi)
        print("input {!r}, expected output {!r}, got {!r}".format(xi, xo,
    output))

    # Check if the best genome is an adequate XOR solver
    best_genome_fitness = eval_fitness(net)
    if best_genome_fitness > config.fitness_threshold:
        print("\n\nSUCCESS: The XOR problem solver found!!!")
    else:
        print("\n\nFAILURE: Failed to find XOR problem solver!!!")

    # Visualize the experiment results
    node_names = {-1:'A', -2: 'B', 0:'A XOR B'}
    visualize.draw_net(config, best_genome, True, node_names=node_names,
    directory=out_dir)
    visualize.plot_stats(stats, ylog=False, view=True, filename=os.path.join
    (out_dir, 'avg_fitness.svg'))
```

```
109     visualize.plot_species(stats, view=True, filename=os.path.join(out_dir,
        'speciation.svg'))
110
111 def clean_output():
112     if os.path.isdir(out_dir):
113         # remove files from previous run
114         shutil.rmtree(out_dir)
115
116     # create the output directory
117     os.makedirs(out_dir, exist_ok=False)
118
119
120 if __name__ == '__main__':
121     # Determine path to configuration file. This path manipulation is
122     # here so that the script will run successfully regardless of the
123     # current working directory.
124     config_path = os.path.join(local_dir, 'xor_config.ini')
125
126     # Clean results of previous run if any or init the ouput directory
127     clean_output()
128
129     # Run the experiment
130     run_experiment(config_path)
```

Annexe 3 : XOR visualization codes

```
#Copyright (c) 2007-2011, cesar.gomes and mirrorballu2
#Copyright (c) 2015-2017, CodeReclaimers, LLC
#
#Redistribution and use in source and binary forms, with or without modification, are permitted provided that the
#following conditions are met:
#
#1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following
#disclaimer.
#
#2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following
#disclaimer in the documentation and/or other materials provided with the distribution.
#
#3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products
#derived from this software without specific prior written permission.
#
#THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
#INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
#DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
#SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
#LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
#CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
#SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
from __future__ import print_function

import copy
import warnings

import graphviz
import matplotlib.pyplot as plt
import numpy as np


def plot_stats(statistics, ylog=False, view=False, filename='avg_fitness.svg'):
    """ Plots the population's average and best fitness. """
    if plt is None:
        warnings.warn("This display is not available due to a missing optional dependency (matplotlib)")
        return

    generation = range(len(statistics.most_fit_genomes))
    best_fitness = [c.fitness for c in statistics.most_fit_genomes]
    avg_fitness = np.array(statistics.get_fitness_mean())
```

```
    stdev_fitness = np.array(statistics.get_fitness_stdev())

    plt.plot(generation, avg_fitness, 'b-', label="average")
    plt.plot(generation, avg_fitness - stdev_fitness, 'g-.', label="-1 sd")
    plt.plot(generation, avg_fitness + stdev_fitness, 'g-.', label="+1 sd")
    plt.plot(generation, best_fitness, 'r-', label="best")

    plt.title("Population's average and best fitness")
    plt.xlabel("Generations")
    plt.ylabel("Fitness")
    plt.grid()
    plt.legend(loc="best")
    if ylog:
        plt.gca().set_yscale('symlog')

    plt.savefig(filename)
    if view:
        plt.show()

    plt.close()

def plot_species(statistics, view=False, filename='speciation.svg'):
    """ Visualizes speciation throughout evolution. """
    if plt is None:
        warnings.warn("This display is not available due to a missing optional dependency (matplotlib)")
        return

    species_sizes = statistics.get_species_sizes()
    num_generations = len(species_sizes)
    curves = np.array(species_sizes).T

    fig, ax = plt.subplots()
    ax.stackplot(range(num_generations), *curves)

    plt.title("Speciation")
    plt.ylabel("Size per Species")
    plt.xlabel("Generations")

    plt.savefig(filename)

    if view:
        plt.show()

    plt.close()


def draw_net(config, genome, view=False, filename=None, directory=None, node_names=None, show_disabled=True, prune_unused=False,
             node_colors=None, fmt='svg'):
    """ Receives a genome and draws a neural network with arbitrary topology. """
    # Attributes for network nodes.
    if graphviz is None:
        warnings.warn("This display is not available due to a missing optional dependency (graphviz)")
        return

    if node_names is None:
```

```
        node_names = {}

    assert type(node_names) is dict

    if node_colors is None:
        node_colors = {}

    assert type(node_colors) is dict

    node_attrs = {
        'shape': 'circle',
        'fontsize': '9',
        'height': '0.2',
        'width': '0.2'}

    dot = graphviz.Digraph(format=fmt, node_attr=node_attrs)

    inputs = set()
    for k in config.genome_config.input_keys:
        inputs.add(k)
        name = node_names.get(k, str(k))
        input_attrs = {'style': 'filled', 'shape': 'box', 'fillcolor':
node_colors.get(k, 'lightgray')}
        dot.node(name, _attributes=input_attrs)

    outputs = set()
    for k in config.genome_config.output_keys:
        outputs.add(k)
        name = node_names.get(k, str(k))
        node_attrs = {'style': 'filled', 'fillcolor': node_colors.get(k, '
lightblue')}

        dot.node(name, _attributes=node_attrs)

    if prune_unused:
        connections = set()
        for cg in genome.connections.values():
            if cg.enabled or show_disabled:
                connections.add((cg.in_node_id, cg.out_node_id))

        used_nodes = copy.copy(outputs)
        pending = copy.copy(outputs)
        while pending:
            new_pending = set()
            for a, b in connections:
                if b in pending and a not in used_nodes:
                    new_pending.add(a)
                    used_nodes.add(a)
            pending = new_pending
    else:
        used_nodes = set(genome.nodes.keys())

    for n in used_nodes:
        if n in inputs or n in outputs:
            continue

        attrs = {'style': 'filled',
                 'fillcolor': node_colors.get(n, 'white')}
        dot.node(str(n), _attributes=attrs)
```

```
154
155     for cg in genome.connections.values():
156         if cg.enabled or show_disabled:
157             #if cg.input not in used_nodes or cg.output not in used_nodes:
158             #    continue
159             input, output = cg.key
160             a = node_names.get(input, str(input))
161             b = node_names.get(output, str(output))
162             style = 'solid' if cg.enabled else 'dotted'
163             color = 'green' if cg.weight > 0 else 'red'
164             width = str(0.1 + abs(cg.weight / 5.0))
165             dot.edge(a, b, _attributes={'style': style, 'color': color, '
    penwidth': width})
166
167     dot.render(filename, directory, view=view)
168
169     return dot
```

Annexe 4 : Maze NEAT experiment Agent class

```
#
# This is the definition of a maze navigating agent.
#
import pickle

class Agent:
    """
    This is the maze navigating agent
    """
    def __init__(self, location, heading=0, speed=0, angular_vel=0, radius=8.0, range_finder_range=100.0):
        """
        Creates new Agent with specified parameters.
        Arguments:
            location:               The agent initial position within maze
            heading:                The heading direction in degrees.
            speed:                  The linear velocity of the agent.
            angular_vel:            The angular velocity of the agent.
            radius:                 The agent's body radius.
            range_finder_range:     The maximal detection range for range finder sensors.
        """
        self.heading = heading
        self.speed = speed
        self.angular_vel = angular_vel
        self.radius = radius
        self.range_finder_range = range_finder_range
        self.location = location

        # defining the range finder sensors
        self.range_finder_angles = [-90.0, -45.0, 0.0, 45.0, 90.0, -180.0]

        # defining the radar sensors
        self.radar_angles = [(315.0, 405.0), (45.0, 135.0), (135.0, 225.0), (225.0, 315.0)]

        # the list to hold range finders activations
        self.range_finders = [None] * len(self.range_finder_angles)
        # the list to hold pie-slice radar activations
        self.radar = [None] * len(self.radar_angles)

class AgentRecord:
    """
    The class to hold results of maze navigation simulation for specific
    solver agent. It provides all statistics about the agent at the end
    of navigation run.
    """
    def __init__(self, generation, agent_id):
        """
        Creates new record for specific agent at the specific generation
        of the evolutionary process.
        """
        self.generation = generation
        self.agent_id = agent_id
```

```
52          # initialize agent's properties
53          self.x = -1
54          self.y = -1
55          self.fitness = -1
56          # The flag to indicate whether this agent was able to find maze exit
57          self.hit_exit = False
58          # The ID of species this agent belongs to
59          self.species_id = -1
60          # The age of agent's species at the time of recording
61          self.species_age = -1
62
63  class AgentRecordStore:
64      """
65      The class to control agents record store.
66      """
67      def __init__(self):
68          """
69          Creates new instance.
70          """
71          self.records = []
72
73      def add_record(self, record):
74          """
75          The function to add specified record to this store.
76          Arguments:
77              record: The record to be added.
78          """
79          self.records.append(record)
80
81      def load(self, file):
82          """
83          The function to load records list from the specied file into this
        class.
84          Arguments:
85              file: The path to the file to read agents records from.
86          """
87          with open(file, 'rb') as dump_file:
88              self.records = pickle.load(dump_file)
89
90      def dump(self, file):
91          """
92          The function to dump records list to the specified file from this
        class.
93          Arguments:
94              file: The path to the file to hold data dump.
95          """
96          with open(file, 'wb') as dump_file:
97              pickle.dump(self.records, dump_file)
```

Annexe 5 : Maze NEAT experiment MazeEnvironment class

```
#
# This is a definition of a maze environment simulation engine. It provides
# routines to read maze configuration and build related simulation
   environment
# from it. Also it provides method to simulate the behavior of the
   navigating agent
# and interaction with his sensors.
#
import math

import agent
import geometry

# The maximal allowed speed for the maze solver agent
MAX_AGENT_SPEED = 3.0

class MazeEnvironment:
    """
    This class encapsulates the maze simulation environment.
    """
    def __init__(self, agent, walls, exit_point, exit_range=10.0):
        """
        Creates new maze environment with specified walls and exit point.
        Arguments:
            agent:      The maze navigating agent
            walls:      The maze walls
            exit_point: The maze exit point
            exit_range: The range arround exit point marking exit area
        """
        self.walls = walls
        self.exit_point = exit_point
        self.exit_range = exit_range
        # The maze navigating agent
        self.agent = agent
        # The flag to indicate if exit was found
        self.exit_found = False
        # The initial distance of agent from exit
        self.initial_distance = self.agent_distance_to_exit()

        # Update sensors
        self.update_rangefinder_sensors()
        self.update_radars()

    def agent_distance_to_exit(self):
        """
        The function to estimate distance from maze solver agent to the maze
    exit.
        Returns:
            The distance from maze solver agent to the maze exit.
        """
        return self.agent.location.distance(self.exit_point)

    def test_wall_collision(self, loc):
        """
```

```
52          The function to test if agent at specified location collides with
    any
53          of the maze walls.
54          Arguments:
55              loc: The new agent location to test for collision.
56          Returns:
57              The True if agent at new location will collide with any of the
    maze walls.
58          """
59          for w in self.walls:
60              if w.distance(loc) < self.agent.radius:
61                  return True
62
63          return False
64
65      def create_net_inputs(self):
66          """
67          The function to create the ANN input values from the simulation
    environment.
68          Returns:
69              The list of ANN inputs consist of values get from solver agent
    sensors.
70          """
71          inputs = []
72          # The range finders
73          for ri in self.agent.range_finders:
74              inputs.append(ri)
75
76          # The radar sensors
77          for rs in self.agent.radar:
78              inputs.append(rs)
79
80          return inputs
81
82      def apply_control_signals(self, control_signals):
83          """
84          The function to apply control signals received from control ANN to
    the
85          maze solver agent.
86          Arguments:
87              control_signals: The control signals received from the control
    ANN
88          """
89          self.agent.angular_vel  += (control_signals[0] - 0.5)
90          self.agent.speed        += (control_signals[1] - 0.5)
91
92          # constrain the speed & angular velocity
93          if self.agent.speed > MAX_AGENT_SPEED:
94              self.agent.speed = MAX_AGENT_SPEED
95
96          if self.agent.speed < -MAX_AGENT_SPEED:
97              self.agent.speed = -MAX_AGENT_SPEED
98
99          if self.agent.angular_vel > MAX_AGENT_SPEED:
100             self.agent.angular_vel = MAX_AGENT_SPEED
101
102         if self.agent.angular_vel < -MAX_AGENT_SPEED:
103             self.agent.angular_vel = -MAX_AGENT_SPEED
104
```

```
105     def update_rangefinder_sensors(self):
106         """
107         The function to update the agent range finder sensors.
108         """
109         for i, angle in enumerate(self.agent.range_finder_angles):
110             rad = geometry.deg_to_rad(angle)
111             # project a point from agent location outwards
112             projection_point = geometry.Point(
113                 x = self.agent.location.x + math.cos(rad) * self.agent.
    range_finder_range,
114                 y = self.agent.location.y + math.sin(rad) * self.agent.
    range_finder_range
115             )
116             # rotate the projection point by the agent's heading angle to
117             # align it with heading direction
118             projection_point.rotate(self.agent.heading, self.agent.location)
119             # create the line segment from the agent location to the
    projected point
120             projection_line = geometry.Line(
121                 a = self.agent.location,
122                 b = projection_point
123             )
124             # set range to maximum detection range
125             min_range = self.agent.range_finder_range
126 
127             # now test against maze walls to see if projection line hits any
     wall
128             # and find the closest hit
129             for wall in self.walls:
130                 found, intersection = wall.intersection(projection_line)
131                 if found:
132                     found_range = intersection.distance(self.agent.location)
133                     # we are interested in the closest hit
134                     if found_range < min_range:
135                         min_range = found_range
136 
137             # Update sensor value
138             self.agent.range_finders[i] = min_range
139 
140     def update_radars(self):
141         """
142         The function to update the agent radar sensors.
143         """
144         target = geometry.Point(self.exit_point.x, self.exit_point.y)
145         # rotate target with respect to the agent's heading to align it with
     heading direction
146         target.rotate(self.agent.heading, self.agent.location)
147         # translate with respect to the agent's location
148         target.x -= self.agent.location.x
149         target.y -= self.agent.location.y
150         # the angle between maze exit point and the agent's heading
    direction
151         angle = target.angle()
152         # find the appropriate radar sensor to be fired
153         for i, r_angles in enumerate(self.agent.radar_angles):
154             self.agent.radar[i] = 0.0 # reset specific radar
155 
156             if (angle >= r_angles[0] and angle < r_angles[1]) or (angle +
    360 >= r_angles[0] and angle + 360 < r_angles[1]):
```

```
157                 self.agent.radar[i] = 1.0 # fire the radar
158
159     def update(self, control_signals):
160         """
161         The function to update solver agent position within maze. After
    agent position
162         updated it will be checked to find out if maze exit was reached
    afetr that.
163         Arguments:
164             control_signals: The control signals received from the control
    ANN
165         Returns:
166             The True if maze exit was found after update or maze exit was
    already
167             found in previous simulation cycles.
168         """
169         if self.exit_found:
170             # Maze exit already found
171             return True
172
173         # Apply control signals
174         self.apply_control_signals(control_signals)
175
176         # get X and Y velocity components
177         vx = math.cos(geometry.deg_to_rad(self.agent.heading)) * self.agent.
    speed
178         vy = math.sin(geometry.deg_to_rad(self.agent.heading)) * self.agent.
    speed
179
180         # Update current Agent's heading (we consider the simulation time
    step size equal to 1s
181         # and the angular velocity as degrees per second)
182         self.agent.heading += self.agent.angular_vel
183
184         # Enforce angular velocity bounds by wrapping
185         if self.agent.heading > 360:
186             self.agent.heading -= 360
187         elif self.agent.heading < 0:
188             self.agent.heading += 360
189
190         # find the next location of the agent
191         new_loc = geometry.Point(
192             x = self.agent.location.x + vx,
193             y = self.agent.location.y + vy
194         )
195
196         if not self.test_wall_collision(new_loc):
197             self.agent.location = new_loc
198
199         # update agent's sensors
200         self.update_rangefinder_sensors()
201         self.update_radars()
202
203         # check if agent reached exit point
204         distance = self.agent_distance_to_exit()
205         self.exit_found = (distance < self.exit_range)
206         return self.exit_found
207
208     def __str__(self):
```

```
209         """
210         Returns the nicely formatted string representation of this
    environment.
211         """
212         str = "MAZE\nAgent at: (%.1f, %.1f)" % (self.agent.location.x, self.
    agent.location.y)
213         str += "\nExit  at: (%.1f, %.1f), exit range: %.1f" % (self.
    exit_point.x, self.exit_point.y, self.exit_range)
214         str += "\nWalls [%d]" % len(self.walls)
215         for w in self.walls:
216             str += "\n\t%s" % w
217 
218         return str
219 
220 def read_environment(file_path):
221     """
222     The function to read maze environment configuration from provided
223     file.
224     Arguments:
225         file_path: The path to the file to read maze configuration from.
226     Returns:
227         The initialized maze environment.
228     """
229     num_lines, index = -1, 0
230     walls = []
231     maze_agent, maze_exit = None, None
232     with open(file_path, 'r') as file:
233         for line in file.readlines():
234             line = line.strip()
235             if len(line) == 0:
236                 # skip empty lines
237                 continue
238 
239             if index == 0:
240                 # read the number of line segments
241                 num_lines = int(line)
242             elif index == 1:
243                 # read the agent's position
244                 loc = geometry.read_point(line)
245                 maze_agent = agent.Agent(location=loc)
246             elif index == 2:
247                 # read the agent's initial heading
248                 maze_agent.heading = float(line)
249             elif index == 3:
250                 # read the maze exit location
251                 maze_exit = geometry.read_point(line)
252             else:
253                 # read the walls
254                 wall = geometry.read_line(line)
255                 walls.append(wall)
256 
257             # increment cursor
258             index += 1
259 
260     assert len(walls) == num_lines
261 
262     print("Maze environment configured successfully from the file: %s" %
    file_path)
263     # create and return the maze environment
```

```
264     return MazeEnvironment(agent=maze_agent, walls=walls, exit_point=
        maze_exit)
265
266 def maze_simulation_evaluate(env, net, time_steps):
267     """
268     The function to evaluate maze simulation for specific environment
269     and controll ANN provided. The results will be saved into provided
270     agent record holder.
271     Arguments:
272         env: The maze configuration environment.
273         net: The maze solver agent's control ANN.
274         time_steps: The number of time steps for maze simulation.
275     """
276     for i in range(time_steps):
277         if maze_simulation_step(env, net):
278             print("Maze solved in %d steps" % (i + 1))
279             return 1.0
280
281     # Calculate the fitness score based on distance from exit
282     fitness = env.agent_distance_to_exit()
283     # Normalize fitness score to range (0,1]
284     fitness = (env.initial_distance - fitness) / env.initial_distance
285     if fitness <= 0.01:
286         fitness = 0.01
287
288     return fitness
289
290
291 def maze_simulation_step(env, net):
292     """
293     The function to perform one step of maze simulation.
294     Arguments:
295         env: The maze configuration environment.
296         net: The maze solver agent's control ANN
297     Returns:
298         The True if maze agent solved the maze.
299     """
300     # create inputs from the current state of the environment
301     inputs = env.create_net_inputs()
302     # load inputs into controll ANN and get results
303     output = net.activate(inputs)
304     # apply control signal to the environment and update
305     return env.update(output)
```

Annexe 6 : Maze NEAT experiment

```
#
# The script to run maze navigation experiment for both medium and hard
# maze configurations.
#

# The Python standard library import
import os
import shutil
import math
import random
import time
import copy
import argparse

# The NEAT-Python library imports
import neat
# The helper used to visualize experiment results
import visualize
import utils

# The maze environment
import maze_environment as maze
import agent

# The current working directory
local_dir = os.path.dirname(__file__)
# The directory to store outputs
out_dir = os.path.join(local_dir, 'out')
out_dir = os.path.join(out_dir, 'maze_objective')

class MazeSimulationTrial:
    """
    The class to hold maze simulator execution parameters and results.
    """
    def __init__(self, maze_env, population):
        """
        Creates new instance and initialize fileds.
        Arguments:
            maze_env:   The maze environment as loaded from configuration file.
            population: The population for this trial run
        """
        # The initial maze simulation environment
        self.orig_maze_environment = maze_env
        # The record store for evaluated maze solver agents
        self.record_store = agent.AgentRecordStore()
        # The NEAT population object
        self.population = population

# The simulation results holder for a one trial.
# It must be initialized before start of each trial.
trialSim = None

def eval_fitness(genome_id, genome, config, time_steps=400):
```

```
54      """
55      Evaluates fitness of the provided genome.
56      Arguments:
57          genome_id:  The ID of genome.
58          genome:     The genome to evaluate.
59          config:     The NEAT configuration holder.
60          time_steps: The number of time steps to execute for maze solver
    simulation.
61      Returns:
62          The phenotype fitness score in range (0, 1]
63      """
64      # run the simulation
65      maze_env = copy.deepcopy(trialSim.orig_maze_environment)
66      control_net = neat.nn.FeedForwardNetwork.create(genome, config)
67      fitness = maze.maze_simulation_evaluate(
68                                          env=maze_env,
69                                          net=control_net,
70                                          time_steps=time_steps)
71
72      # Store simulation results into the agent record
73      record = agent.AgentRecord(
74          generation=trialSim.population.generation,
75          agent_id=genome_id)
76      record.fitness = fitness
77      record.x = maze_env.agent.location.x
78      record.y = maze_env.agent.location.y
79      record.hit_exit = maze_env.exit_found
80      record.species_id = trialSim.population.species.get_species_id(genome_id
    )
81      record.species_age = record.generation - trialSim.population.species.
    get_species(genome_id).created
82      # add record to the store
83      trialSim.record_store.add_record(record)
84
85      return fitness
86
87  def eval_genomes(genomes, config):
88      """
89      The function to evaluate the fitness of each genome in
90      the genomes list.
91      Arguments:
92          genomes: The list of genomes from population in the
93                   current generation
94          config:  The configuration settings with algorithm
95                   hyper-parameters
96      """
97      for genome_id, genome in genomes:
98          genome.fitness = eval_fitness(genome_id, genome, config)
99
100 def run_experiment(config_file, maze_env, trial_out_dir, args=None,
    n_generations=100, silent=False):
101     """
102     The function to run the experiment against hyper-parameters
103     defined in the provided configuration file.
104     The winner genome will be rendered as a graph as well as the
105     important statistics of neuroevolution process execution.
106     Arguments:
107         config_file:    The path to the file with experiment configuration
108         maze_env:       The maze environment to use in simulation.
```

```
        trial_out_dir:  The directory to store outputs for this trial
        n_generations:  The number of generations to execute.
        silent:         If True than no intermediary outputs will be
                        presented until solution is found.
        args:           The command line arguments holder.
    Returns:
        True if experiment finished with successful solver found.
    """
    # set random seed
    seed = int(time.time())
    random.seed(seed)

    # Load configuration.
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         config_file)

    # Create the population, which is the top-level object for a NEAT run.
    p = neat.Population(config)

    # Create the trial simulation
    global trialSim
    trialSim = MazeSimulationTrial(maze_env=maze_env, population=p)

    # Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    p.add_reporter(neat.Checkpointer(5, filename_prefix='%s/maze-neat-
checkpoint-' % trial_out_dir))

    # Run for up to N generations.
    start_time = time.time()
    best_genome = p.run(eval_genomes, n=n_generations)

    elapsed_time = time.time() - start_time

    # Display the best genome among generations.
    print('\nBest genome:\n%s' % (best_genome))

    solution_found = (best_genome.fitness >= config.fitness_threshold)
    if solution_found:
        print("SUCCESS: The stable maze solver controller was found!!!")
    else:
        print("FAILURE: Failed to find the stable maze solver controller!!!"
)

    # write the record store data
    rs_file = os.path.join(trial_out_dir, "data.pickle")
    trialSim.record_store.dump(rs_file)

    print("Record store file: %s" % rs_file)
    print("Random seed:", seed)
    print("Trial elapsed time: %.3f sec" % (elapsed_time))

    # Visualize the experiment results
    if not silent or solution_found:
        node_names =   {-1:'RF_R', -2:'RF_FR', -3:'RF_F', -4:'RF_FL', -5:'
RF_L', -6: 'RF_B',
```

```
                          -7:'RAD_F', -8:'RAD_L', -9:'RAD_B', -10:'RAD_R',
                          0:'ANG_VEL', 1:'VEL'}
        visualize.draw_net(config, best_genome, True, node_names=node_names,
    directory=trial_out_dir, fmt='svg')
        if args is None:
            visualize.draw_maze_records(maze_env, trialSim.record_store.
    records, view=True)
        else:
            visualize.draw_maze_records(maze_env, trialSim.record_store.
    records,
                                        view=True,
                                        width=args.width,
                                        height=args.height,
                                        filename=os.path.join(trial_out_dir,
    'maze_records.svg'))
        visualize.plot_stats(stats, ylog=False, view=True, filename=os.path.
    join(trial_out_dir, 'avg_fitness.svg'))
        visualize.plot_species(stats, view=True, filename=os.path.join(
    trial_out_dir, 'speciation.svg'))

    return solution_found

if __name__ == '__main__':
    # read command line parameters
    parser = argparse.ArgumentParser(description="The maze experiment runner
    .")
    parser.add_argument('-m', '--maze', default='medium',
                        help='The maze configuration to use.')
    parser.add_argument('-g', '--generations', default=500, type=int,
                        help='The number of generations for the evolutionary
    process.')
    parser.add_argument('--width', type=int, default=400, help='The width of
    the records subplot')
    parser.add_argument('--height', type=int, default=400, help='The height
    of the records subplot')
    args = parser.parse_args()

    if not (args.maze == 'medium' or args.maze == 'hard'):
        print('Unsupported maze configuration: %s' % args.maze)
        exit(1)

    # Determine path to configuration file.
    config_path = os.path.join(local_dir, 'maze_config.ini')

    trial_out_dir = os.path.join(out_dir, args.maze)

    # Clean results of previous run if any or init the ouput directory
    utils.clear_output(trial_out_dir)

    # Run the experiment
    maze_env_config = os.path.join(local_dir, '%s_maze.txt' % args.maze)
    maze_env = maze.read_environment(maze_env_config)

    # visualize.draw_maze_records(maze_env, None, view=True)

    print("Starting the %s maze experiment" % args.maze)
    run_experiment( config_file=config_path,
                    maze_env=maze_env,
                    trial_out_dir=trial_out_dir,
```

```
214                         n_generations=args.generations,
215                         args=args)
```

Annexe 7 : Maze NEAT experiment configuration file

```
#--- Hyper-parameters for the Single-Pole balancing experiment ---#

[NEAT]
fitness_criterion     = max
fitness_threshold     = 1.0
pop_size              = 250
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.0
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.1
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = False
initial_connection      = partial_direct 0.5

# node add/remove rates
node_add_prob           = 0.1
node_delete_prob        = 0.1

# network parameters
num_hidden              = 1
num_inputs              = 10
num_outputs             = 2

# node response options
response_init_mean      = 1.0
```

```
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 1

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.1
min_species_size   = 2
```

Annexe 8 : Maze NEAT experiment visualization methods

```
#Copyright (c) 2007-2011, cesar.gomes and mirrorballu2
#Copyright (c) 2015-2017, CodeReclaimers, LLC
#
#Redistribution and use in source and binary forms, with or without modification, are permitted provided that the
#following conditions are met:
#
#1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following
#disclaimer.
#
#2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following
#disclaimer in the documentation and/or other materials provided with the distribution.
#
#3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products
#derived from this software without specific prior written permission.
#
#THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
#INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
#DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
#SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
#LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
#CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
#SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
from __future__ import print_function

import copy
import warnings
import random
import argparse
import os

import graphviz
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
import numpy as np

import geometry
import agent
import maze_environment as maze

def plot_stats(statistics, ylog=False, view=False, filename='avg_fitness.svg'):
    """ Plots the population's average and best fitness. """
```

```
43      if plt is None:
44          warnings.warn("This display is not available due to a missing
    optional dependency (matplotlib)")
45          return
46
47      generation = range(len(statistics.most_fit_genomes))
48      best_fitness = [c.fitness for c in statistics.most_fit_genomes]
49      avg_fitness = np.array(statistics.get_fitness_mean())
50      stdev_fitness = np.array(statistics.get_fitness_stdev())
51
52      plt.plot(generation, avg_fitness, 'b-', label="average")
53      plt.plot(generation, avg_fitness - stdev_fitness, 'g-.', label="-1 sd")
54      plt.plot(generation, avg_fitness + stdev_fitness, 'g-.', label="+1 sd")
55      plt.plot(generation, best_fitness, 'r-', label="best")
56
57      plt.title("Population's average and best fitness")
58      plt.xlabel("Generations")
59      plt.ylabel("Fitness")
60      plt.grid()
61      plt.legend(loc="best")
62      if ylog:
63          plt.gca().set_yscale('symlog')
64
65      plt.savefig(filename)
66      if view:
67          plt.show()
68
69      plt.close()
70
71  def plot_species(statistics, view=False, filename='speciation.svg'):
72      """ Visualizes speciation throughout evolution. """
73      if plt is None:
74          warnings.warn("This display is not available due to a missing
    optional dependency (matplotlib)")
75          return
76
77      species_sizes = statistics.get_species_sizes()
78      num_generations = len(species_sizes)
79      curves = np.array(species_sizes).T
80
81      fig, ax = plt.subplots()
82      ax.stackplot(range(num_generations), *curves)
83
84      plt.title("Speciation")
85      plt.ylabel("Size per Species")
86      plt.xlabel("Generations")
87
88      plt.savefig(filename)
89
90      if view:
91          plt.show()
92
93      plt.close()
94
95
96  def draw_net(config, genome, view=False, filename=None, directory=None,
    node_names=None, show_disabled=True, prune_unused=False,
97               node_colors=None, fmt='svg'):
98      """ Receives a genome and draws a neural network with arbitrary topology
```

```
    . """
99      # Attributes for network nodes.
100     if graphviz is None:
101         warnings.warn("This display is not available due to a missing
    optional dependency (graphviz)")
102         return
103 
104     if node_names is None:
105         node_names = {}
106 
107     assert type(node_names) is dict
108 
109     if node_colors is None:
110         node_colors = {}
111 
112     assert type(node_colors) is dict
113 
114     node_attrs = {
115         'shape': 'circle',
116         'fontsize': '9',
117         'height': '0.2',
118         'width': '0.2'}
119 
120     dot = graphviz.Digraph(format=fmt, node_attr=node_attrs)
121 
122     inputs = set()
123     for k in config.genome_config.input_keys:
124         inputs.add(k)
125         name = node_names.get(k, str(k))
126         input_attrs = {'style': 'filled', 'shape': 'box', 'fillcolor':
    node_colors.get(k, 'lightgray')}
127         dot.node(name, _attributes=input_attrs)
128 
129     outputs = set()
130     for k in config.genome_config.output_keys:
131         outputs.add(k)
132         name = node_names.get(k, str(k))
133         node_attrs = {'style': 'filled', 'fillcolor': node_colors.get(k, '
    lightblue')}
134 
135         dot.node(name, _attributes=node_attrs)
136 
137     if prune_unused:
138         connections = set()
139         for cg in genome.connections.values():
140             if cg.enabled or show_disabled:
141                 connections.add((cg.in_node_id, cg.out_node_id))
142 
143         used_nodes = copy.copy(outputs)
144         pending = copy.copy(outputs)
145         while pending:
146             new_pending = set()
147             for a, b in connections:
148                 if b in pending and a not in used_nodes:
149                     new_pending.add(a)
150                     used_nodes.add(a)
151             pending = new_pending
152     else:
153         used_nodes = set(genome.nodes.keys())
```

```
154
155     for n in used_nodes:
156         if n in inputs or n in outputs:
157             continue
158
159         attrs = {'style': 'filled',
160                  'fillcolor': node_colors.get(n, 'white')}
161         dot.node(str(n), _attributes=attrs)
162
163     for cg in genome.connections.values():
164         if cg.enabled or show_disabled:
165             #if cg.input not in used_nodes or cg.output not in used_nodes:
166             #    continue
167             input, output = cg.key
168             a = node_names.get(input, str(input))
169             b = node_names.get(output, str(output))
170             style = 'solid' if cg.enabled else 'dotted'
171             color = 'green' if cg.weight > 0 else 'red'
172             width = str(0.1 + abs(cg.weight / 5.0))
173             dot.edge(a, b, _attributes={'style': style, 'color': color, '
    penwidth': width})
174
175     dot.render(filename, directory, view=view)
176
177     return dot
178
179 def draw_maze_records(maze_env, records, best_threshold=0.8, filename=None,
    view=False, show_axes=False, width=400, height=400, fig_height=7):
180     """
181     The function to draw maze with recorded agents positions.
182     Arguments:
183         maze_env:       The maze environment configuration.
184         records:        The records of solver agents collected during NEAT
    execution.
185         best_threshold: The minimal fitness of maze solving agent's species
    to be included into the best ones.
186         view:           The flag to indicate whether to view plot.
187         width:          The width of drawing in pixels
188         height:         The height of drawing in pixels
189         fig_height:      The plot figure height in inches
190     """
191     # find the distance threshold for the best species
192     dist_threshold = maze_env.agent_distance_to_exit() * (1.0 -
    best_threshold)
193     # generate color palette and find the best species IDS
194     max_sid = 0
195     for r in records:
196         if r.species_id > max_sid:
197             max_sid = r.species_id
198     colors = [None] * (max_sid + 1)
199     sp_idx = [False] * (max_sid + 1)
200     best_sp_idx = [0] * (max_sid + 1)
201     for r in records:
202         if not sp_idx[r.species_id]:
203             sp_idx[r.species_id] = True
204             rgb = (random.random(), random.random(), random.random())
205             colors[r.species_id] = rgb
206         if maze_env.exit_point.distance(geometry.Point(r.x, r.y)) <=
    dist_threshold:
```

```
            best_sp_idx[r.species_id] += 1

    # initialize plotting
    fig = plt.figure()
    fig.set_dpi(100)
    fig_width = fig_height * (float(width)/float(2.0 * height )) - 0.2
    print("Plot figure width: %.1f, height: %.1f" % (fig_width, fig_height))
    fig.set_size_inches(fig_width, fig_height)
    ax1, ax2 = fig.subplots(2, 1, sharex=True)
    ax1.set_xlim(0, width)
    ax1.set_ylim(0, height)
    ax2.set_xlim(0, width)
    ax2.set_ylim(0, height)

    # draw species
    n_best_species = 0
    for i, v in enumerate(best_sp_idx):
        if v > 0:
            n_best_species += 1
            _draw_species_(records=records, sid=i, colors=colors, ax=ax1)
        else:
            _draw_species_(records=records, sid=i, colors=colors, ax=ax2)

    ax1.set_title('fitness >= %.1f, species: %d' % (best_threshold,
   n_best_species))
    ax2.set_title('fitness < %.1f' % best_threshold)

    # draw maze
    _draw_maze_(maze_env, ax1)
    _draw_maze_(maze_env, ax2)

    # turn off axis rendering
    if not show_axes:
        ax1.axis('off')
        ax2.axis('off')
    # Invert Y axis to have coordinates origin at the top left
    ax1.invert_yaxis()
    ax2.invert_yaxis()

    # Save figure to file
    if filename is not None:
        plt.savefig(filename)

    if view:
        plt.show()

    plt.close()

def _draw_species_(records, sid, colors, ax):
    """
    The function to draw specific species from the records with
    particular color.
    Arguments:
        records:    The records of solver agents collected during NEAT
   execution.
        sid:        The species ID
        colors:     The colors table by species ID
        ax:         The figure axis instance
    """
```

```
264     for r in records:
265         if r.species_id == sid:
266             circle = plt.Circle((r.x, r.y), 2.0, facecolor=colors[r.
    species_id])
267             ax.add_patch(circle)
268 
269 def _draw_maze_(maze_env, ax):
270     """
271     The function to draw maze environment
272     Arguments:
273         maze_env:   The maze environment configuration.
274         ax:         The figure axis instance
275     """
276     # draw maze walls
277     for wall in maze_env.walls:
278         line = plt.Line2D((wall.a.x, wall.b.x), (wall.a.y, wall.b.y), lw
    =1.5)
279         ax.add_line(line)
280 
281     # draw start point
282     start_circle = plt.Circle((maze_env.agent.location.x, maze_env.agent.
    location.y),
283                               radius=2.5, facecolor=(0.6, 1.0, 0.6),
    edgecolor='w')
284     ax.add_patch(start_circle)
285 
286     # draw exit point
287     exit_circle = plt.Circle((maze_env.exit_point.x, maze_env.exit_point.y),
288                              radius=2.5, facecolor=(1.0, 0.2, 0.0),
    edgecolor='w')
289     ax.add_patch(exit_circle)
290 
291 if __name__ == '__main__':
292     # read command line parameters
293     parser = argparse.ArgumentParser(description="The maze experiment
    visualizer.")
294     parser.add_argument('-m', '--maze', default='medium', help='The maze
    configuration to use.')
295     parser.add_argument('-r', '--records', help='The records file.')
296     parser.add_argument('-o', '--output', help='The file to store the plot.'
    )
297     parser.add_argument('--width', type=int, default=400, help='The width of
     the subplot')
298     parser.add_argument('--height', type=int, default=400, help='The height
    of the subplot')
299     parser.add_argument('--fig_height', type=float, default=7, help='The
    height of the plot figure')
300     parser.add_argument('--show_axes', type=bool, default=False, help='The
    flag to indicate whether to show plot axes.')
301     args = parser.parse_args()
302 
303     local_dir = os.path.dirname(__file__)
304     if not (args.maze == 'medium' or args.maze == 'hard'):
305         print('Unsupported maze configuration: %s' % args.maze)
306         exit(1)
307 
308     # read maze environment
309     maze_env_config = os.path.join(local_dir, '%s_maze.txt' % args.maze)
310     maze_env = maze.read_environment(maze_env_config)
```

```
311
312     # read agents records
313     rs = agent.AgentRecordStore()
314     rs.load(args.records)
315
316     # render visualization
317     random.seed(42)
318     draw_maze_records(maze_env,
319                       rs.records,
320                       width=args.width,
321                       height=args.height,
322                       fig_height=args.fig_height,
323                       view=True,
324                       show_axes=args.show_axes,
325                       filename=args.output)
```

Annexe 9 : Maze NEAT experiment geometry methods

```
#
# Here we define common geometric primitives along with utilities allowing to find distance from point to the line, to find intersection point
# of two lines, and to find the length of the line in two dimensional Euclidean
# space.
#

import math

def deg_to_rad(degrees):
    """
    The function to convert degrees to radians.
    Arguments:
        degrees: The angle in degrees to be converted.
    Returns:
        The degrees converted to radians.
    """
    return degrees / 180.0 * math.pi

def read_point(str):
    """
    The function to read Point from specified string. The point
    coordinates are in order (x, y) and delimited by space.
    Arguments:
        str: The string encoding Point coorinates.
    Returns:
        The Point with coordinates parsed from provided string.
    """
    coords = str.split(' ')
    assert len(coords) == 2
    return Point(float(coords[0]), float(coords[1]))

def read_line(str):
    """
    The function to read line segment from provided string. The coordinates
    of line end points are in order: x1, y1, x2, y2 and delimited by spaces.
    Arguments:
        str: The string to read line coordinates from.
    Returns:
        The parsed line segment.
    """
    coords = str.split(' ')
    assert len(coords) == 4
    a = Point(float(coords[0]), float(coords[1]))
    b = Point(float(coords[2]), float(coords[3]))
    return Line(a, b)

class Point:
    """
    The basic class describing point in the two dimensional Cartesian coordinate
    system.
```

```
    """
    def __init__(self, x, y):
        """
        Creates new point at specified coordinates
        """
        self.x = x
        self.y = y

    def angle(self):
        """
        The function to determine angle in degrees of vector drawn from the
        center of coordinates to this point. The angle values is in range
        from 0 to 360 degrees in anticlockwise direction.
        """
        ang = math.atan2(self.y, self.x) / math.pi * 180.0
        if (ang < 0.0):
            # the lower quadrants (3 or 4)
            return ang + 360
        return ang

    def rotate(self, angle, point):
        """
        The function to rotate this point around another point with given
        angle in degrees.
        Arguments:
            angle: The rotation angle (degrees)
            point: The point - center of rotation
        """
        rad = deg_to_rad(angle)
        # translate to have another point at the center of coordinates
        self.x -= point.x
        self.y -= point.y
        # rotate
        ox, oy = self.x, self.y
        self.x = math.cos(rad) * ox - math.sin(rad) * oy
        self.y = math.sin(rad) * ox - math.cos(rad) * oy
        # restore
        self.x += point.x
        self.y += point.y

    def distance(self, point):
        """
        The function to caclulate Euclidean distance between this and given
    point.
        Arguments:
            point: The another point
        Returns:
            The Euclidean distance between this and given point.
        """
        dx = self.x - point.x
        dy = self.y - point.y

        return math.sqrt(dx*dx + dy*dy)

    def __str__(self):
        """
        Returns the nicely formatted string representation of this point.
        """
        return "Point (%.1f, %.1f)" % (self.x, self.y)
```

```
110
111 class Line:
112     """
113     The simple line segment between two points. Used to represent maze wals.
114     """
115     def __init__(self, a, b):
116         """
117         Creates new line segment between two points.
118         Arguments:
119             a, b: The end points of the line
120         """
121         self.a = a
122         self.b = b
123
124     def midpoint(self):
125         """
126         The function to find midpoint of this line segment.
127         Returns:
128             The midpoint of this line segment.
129         """
130         x = (self.a.x + self.b.x) / 2.0
131         y = (self.a.y + self.b.y) / 2.0
132
133         return Point(x, y)
134
135     def intersection(self, line):
136         """
137         The function to find intersection between this line and the given
    one.
138         Arguments:
139             line: The line to test intersection against.
140         Returns:
141             The tuple with the first value indicating if intersection was
    found (True/False)
142             and the second value holding the intersection Point or None
143         """
144         A, B, C, D = self.a, self.b, line.a, line.b
145
146         rTop = (A.y - C.y) * (D.x - C.x) - (A.x - C.x) * (D.y - C.y)
147         rBot = (B.x - A.x) * (D.y - C.y) - (B.y - A.y) * (D.x - C.x)
148
149         sTop = (A.y - C.y) * (B.x - A.x) - (A.x - C.x) * (B.y - A.y)
150         sBot = (B.x - A.x) * (D.y - C.y) - (B.y - A.y) * (D.x - C.x)
151
152         if rBot == 0 or sBot == 0:
153             # lines are parallel
154             return False, None
155
156         r = rTop / rBot
157         s = sTop / sBot
158         if r > 0 and r < 1 and s > 0 and s < 1:
159             x = A.x + r * (B.x - A.x)
160             y = A.y + r * (B.y - A.y)
161             return True, Point(x, y)
162
163         return False, None
164
165     def distance(self, p):
166         """
```

```
        The function to estimate distance to the given point from this line.
        Arguments:
            p: The point to find distance to.
        Returns:
            The distance between given point and this line.
        """
        utop = (p.x - self.a.x) * (self.b.x - self.a.x) + (p.y - self.a.y) *
    (self.b.y - self.a.y)
        ubot = self.a.distance(self.b)
        ubot *= ubot
        if ubot == 0.0:
            return 0.0

        u = utop / ubot
        if u < 0 or u > 1:
            d1 = self.a.distance(p)
            d2 = self.b.distance(p)
            if d1 < d2:
                return d1
            return d2

        x = self.a.x + u * (self.b.x - self.a.x)
        y = self.a.y + u * (self.b.y - self.a.y)
        point = Point(x, y)
        return point.distance(p)

    def length(self):
        """
        The function to calculate the length of this line segment.
        Returns:
            The length of this line segment as distance between its
   endpoints.
        """
        return self.a.distance(self.b)

    def __str__(self):
        """
        Returns the nicely formatted string representation of this line.
        """
        return "Line (%.1f, %.1f) -> (%.1f, %.1f)" % (self.a.x, self.a.y,
   self.b.x, self.b.y)
```

Annexe 10 : Maze NEAT experiment utility methods

```
#
# The collection of utilities
#
import os
import shutil

def clear_output(out_dir):
    """
    Function to clear output directory.
    Arguments:
        out_dir: The directory to be cleared
    """
    if os.path.isdir(out_dir):
        # remove files from previous run
        shutil.rmtree(out_dir)
```

```
16
17      # create the output directory
18      os.makedirs(out_dir, exist_ok=False)
```

Annexe 11 : text file decribing a medium maze

```
11
30 22
0
270 100
5 5 295 5
295 5 295 135
295 135 5 135
5 135 5 5
241 135 58 65
114 5 73 42
130 91 107 46
196 5 139 51
219 125 182 63
267 5 214 63
271 135 237 88
```

Annexe 12 : text file describing a hard maze

```
11
36 184
0
31 20

5 5 5 200
5 200 200 200
200 200 200 5
200 5 5 5

5 49 57 53
56 54 56 157
57 106 158 162
77 200 108 164
5 80 33 121
200 146 87 91
56 55 133 30
```

Annexe 13 : Maze NS experiment

```
#
# The script to run maze navigation experiment for both medium and hard
# maze configurations.
#

# The Python standard library import
import os
import shutil
import math
import random
import time
import copy
import argparse

# The NEAT-Python library imports
import neat
# The helper used to visualize experiment results
import visualize
import utils
import trials_archive as trial

# The maze environment
import maze_environment as maze
import agent
import novelty_archive as archive

# The number of maze solving simulator steps
SOLVER_TIME_STEPS = 400
# The minimal goal fitness criterion
MCNS = 0.01

class MazeSimulationTrial:
    """
    The class to hold maze simulator execution parameters and results.
    """
    def __init__(self, maze_env, population, archive):
        """
        Creates new instance and initialize fileds.
        Arguments:
            maze_env:   The maze environment as loaded from configuration
   file.
            population: The population for this trial run
            archive:    The archive to hold NoveltyItems
        """
        # The initial maze simulation environment
        self.orig_maze_environment = maze_env
        # The record store for evaluated maze solver agents
        self.record_store = agent.AgentRecordStore()
        # The NEAT population object
        self.population = population
        # The NoveltyItem archive
        self.archive = archive

# The simulation results holder for a one trial.
```

```
# It must be initialized before start of each trial.
trial_sim = None

def eval_individual(genome_id, genome, genomes, n_items_map, config):
    """
    Evaluates the individual represented by genome.
    Arguments:
        genome_id:      The ID of genome.
        genome:         The genome to evaluate.
        genomes:        The genomes population for current generation.
        n_items_map:    The map to hold novelty items for current generation
    .
        config:         The NEAT configuration holder.
    Return:
        The True if successful solver found.
    """
    # create NoveltyItem for genome and store it into map
    n_item = archive.NoveltyItem(generation=trial_sim.population.generation,
                                 genomeId=genome_id)
    n_items_map[genome_id] = n_item
    # run the simulation
    maze_env = copy.deepcopy(trial_sim.orig_maze_environment)
    control_net = neat.nn.FeedForwardNetwork.create(genome, config)
    goal_fitness = maze.maze_simulation_evaluate(
                                        env=maze_env,
                                        net=control_net,
                                        time_steps=SOLVER_TIME_STEPS,
                                        n_item=n_item,
                                        mcns=MCNS)

    if goal_fitness == -1:
        # The individual doesn't meet the minimal fitness criterion
        print("Individ with ID: %d marked for extiction, MCNS: %f"
                % (genome_id, MCNS))
        return False

    # Store simulation results into the agent record
    record = agent.AgentRecord(
        generation=trial_sim.population.generation,
        agent_id=genome_id, genome=genome)
    record.fitness = goal_fitness
    record.x = maze_env.agent.location.x
    record.y = maze_env.agent.location.y
    record.hit_exit = maze_env.exit_found
    record.species_id = trial_sim.population.species \
        .get_species_id(genome_id)
    record.species_age = record.generation - \
        trial_sim.population.species.get_species(genome_id).created
    # add record to the store
    trial_sim.record_store.add_record(record)

    # Evaluate the novelty of a genome and add the novelty item to the
   archive of Novelty items if appropriate
    if not maze_env.exit_found:
        # evaluate genome novelty and add it to the archive if appropriate
        record.novelty = trial_sim.archive \
            .evaluate_individual_novelty(genome=genome, genomes=genomes,
                                        n_items_map=n_items_map)

```

```
111     # update fittest organisms list
112     trial_sim.archive.update_fittest_with_genome(genome=genome,
113                                                  n_items_map=n_items_map)
114 
115     return maze_env.exit_found
116 
117 def eval_genomes(genomes, config):
118     """
119     The function to evaluate the fitness of each genome in
120     the genomes list.
121     Arguments:
122         genomes: The list of genomes from population in the
123                  current generation
124         config:  The configuration settings with algorithm
125                  hyper-parameters
126     """
127     n_items_map = {} # The map to hold the novelty items for current
    generation
128     solver_genome = None
129     for genome_id, genome in genomes:
130         found = eval_individual(genome_id=genome_id,
131                                 genome=genome,
132                                 genomes=genomes,
133                                 n_items_map=n_items_map,
134                                 config=config)
135         if found:
136             solver_genome = genome
137 
138     # now adjust the archive settings and evaluate population
139     trial_sim.archive.end_of_generation()
140     for genome_id, genome in genomes:
141         # set fitness value as a logarithm of a novelty score of a genome in
     the population
142         fitness = trial_sim.archive.evaluate_individual_novelty(genome=
    genome,
143                                                                 genomes=
    genomes,
144                                                                 n_items_map=
    n_items_map,
145                                                                 only_fitness
    =True)
146         # To avoid negative genome fitness scores we just set to zero all
    obtained
147         # fitness scores that is less than 1 (note we use the natural
    logarithm)
148         if fitness > 1:
149             fitness = math.log(fitness)
150         else:
151             fitness = 0
152         # assign the adjusted fitness score to the genome
153         genome.fitness = fitness
154 
155     # if successful maze solver was found then adjust its fitness
156     # to signal the finish evolution
157     if solver_genome is not None:
158         solver_genome.fitness = math.log(800000) # ~=13.59
159 
160 
161 def run_experiment(config_file, maze_env, novelty_archive, trial_archive,
```

```
    trial_out_dir, args=None, n_generations=100,
162                   save_results=False, silent=False):
163     """
164     The function to run the experiment against hyper-parameters
165     defined in the provided configuration file.
166     The winner genome will be rendered as a graph as well as the
167     important statistics of neuroevolution process execution.
168     Arguments:
169         config_file:        The path to the file with experiment
    configuration
170         maze_env:           The maze environment to use in simulation.
171         novelty_archive:    The archive to work with NoveltyItems.
172         trial_out_dir:      The directory to store outputs for this trial
173         n_generations:      The number of generations to execute.
174         save_results:       The flag to control if intermdiate results will
    be saved.
175         silent:             If True than no intermediary outputs will be
176                             presented until solution is found.
177         args:               The command line arguments holder.
178     Returns:
179         True if experiment finished with successful solver found.
180     """
181     # set random seed
182     seed = int(time.time()) #1687106299
183     random.seed(seed)
184     print("Selected random seed:", seed)
185 
186     # Load configuration.
187     config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
188                          neat.DefaultSpeciesSet, neat.DefaultStagnation,
189                          config_file)
190 
191     # Create the population, which is the top-level object for a NEAT run.
192     p = neat.Population(config)
193 
194     # Create the trial simulation
195     global trial_sim
196     trial_sim = MazeSimulationTrial(maze_env=maze_env,
197                                     population=p,
198                                     archive=novelty_archive)
199 
200     # Add a stdout reporter to show progress in the terminal.
201     p.add_reporter(neat.StdOutReporter(True))
202     stats = neat.StatisticsReporter()
203     p.add_reporter(stats)
204 
205     # Run for up to N generations.
206     start_time = time.time()
207     best_genome = p.run(eval_genomes, n=n_generations)
208 
209     elapsed_time = time.time() - start_time
210 
211     # Display the best genome among generations.
212     print('\nBest genome:\n%s' % (best_genome))
213 
214     solution_found = \
215         (best_genome.fitness >= config.fitness_threshold)
216     if solution_found:
217         print("SUCCESS: The stable maze solver controller was found!!!")
```

```
218     else:
219         print("FAILURE: Failed to find the stable maze solver controller!!!"
    )
220 
221     #Store data of the trial in the archive
222     trial_archive.avg_fitness = stats.get_fitness_mean()
223     trial_archive.best_fitness = [c.fitness for c in stats.most_fit_genomes]
224     trial_archive.seed = seed
225     trial_archive.gen_max = n_generations
226     if solution_found:
227         trial_archive.success_gen = len(stats.most_fit_genomes)
228         trial_archive.best_genome_complexity = best_genome.size()
229 
230     archive_store.add_archive(trial_archive)
231     print("Data stored in archive")
232 
233     # write the record store data
234     rs_file = os.path.join(trial_out_dir, "data.pickle")
235     trial_sim.record_store.dump(rs_file)
236 
237     print("Record store file: %s" % rs_file)
238     print("Random seed:", seed)
239     print("Trial elapsed time: %.3f sec" % (elapsed_time))
240 
241     # Visualize the experiment results
242     show_results = not silent #or solution_found
243     if save_results or show_results:
244         node_names =   {-1:'RF_R', -2:'RF_FR', -3:'RF_F', -4:'RF_FL', -5:'
    RF_L', -6: 'RF_B',
245                         -7:'RAD_F', -8:'RAD_L', -9:'RAD_B', -10:'RAD_R',
246                         0:'ANG_VEL', 1:'VEL'}
247         visualize.draw_net(config, best_genome, view=show_results,
    node_names=node_names, directory=trial_out_dir, fmt='svg')
248         if args is None:
249             visualize.draw_maze_records(maze_env, trial_sim.record_store.
    records, view=show_results)
250         else:
251             visualize.draw_maze_records(maze_env, trial_sim.record_store.
    records,
252                                         view=show_results, width=args.width,
     height=args.height,
253                                         filename=os.path.join(trial_out_dir,
     'maze_records.svg'))
254         visualize.plot_stats(stats, ylog=False, view=show_results, filename=
    os.path.join(trial_out_dir, 'avg_fitness.svg'))
255         visualize.plot_species(stats, view=show_results, filename=os.path.
    join(trial_out_dir, 'speciation.svg'))
256 
257         # store NoveltyItems archive data
258         trial_sim.archive.write_fittest_to_file(path=os.path.join(
    trial_out_dir, 'ns_items_fittest.txt'))
259         trial_sim.archive.write_to_file(path=os.path.join(trial_out_dir, '
    ns_items_all.txt'))
260 
261         # create the best genome simulation path and render
262         maze_env = copy.deepcopy(trial_sim.orig_maze_environment)
263         control_net = neat.nn.FeedForwardNetwork.create(best_genome, config)
264         path_points = []
265         evaluate_fitness = maze.maze_simulation_evaluate(
```

```
                                    env=maze_env,
                                    net=control_net,
                                    time_steps=SOLVER_TIME_STEPS,
                                    path_points=path_points)
        print("Evaluated fitness of best agent: %f" % evaluate_fitness)
        visualize.draw_agent_path(trial_sim.orig_maze_environment, path_points, best_genome,
                                    view=show_results,
                                    width=args.width,
                                    height=args.height,
                                    filename=os.path.join(trial_out_dir, 'best_solver_path.svg'))

        visualize.animate_agent_path(trial_sim.orig_maze_environment, path_points, best_genome, trial_out_dir,
                                    width=args.width,
                                    height=args.height)


    return solution_found

if __name__ == '__main__':
    # read command line parameters
    parser = argparse.ArgumentParser(description="The maze experiment runner (Novelty Search).")
    parser.add_argument('-m', '--maze', default='medium',
                        help='The maze configuration to use.')
    parser.add_argument('-g', '--generations', default=500, type=int,
                        help='The number of generations for the evolutionary process.')
    parser.add_argument('-t', '--trials', type=int, default=1, help='The number of trials to run')
    parser.add_argument('-n', '--ns_threshold', type=float, default=6.0,
                        help="The novelty threshold value for the archive of NoveltyItems.")
    parser.add_argument('-r', '--location_sample_rate', type=int, default=4000,
                        help="The sample rate of agent position points saving during simulation steps.")
    parser.add_argument('--width', type=int, default=400, help='The width of the records subplot')
    parser.add_argument('--height', type=int, default=400, help='The height of the records subplot')
    args = parser.parse_args()

    if not (args.maze == 'medium' or args.maze == 'hard'):
        print('Unsupported maze configuration: %s' % args.maze)
        exit(1)

    # The current working directory
    local_dir = os.path.dirname(__file__)
    # The directory to store outputs
    out_dir = os.path.join(local_dir, 'out')
    out_dir = os.path.join(out_dir, 'maze_ns')

    # Determine path to configuration file.
    config_path = os.path.join(local_dir, 'maze_config.ini')

    # Clean results of previous run if any or init the ouput directory
```

```
    out_dir = os.path.join(out_dir, args.maze)
    utils.clear_output(out_dir)

    # Read the maze environment configuration
    maze_env_config = os.path.join(local_dir, '%s_maze.txt' % args.maze)
    maze_env = maze.read_environment(maze_env_config)
    maze_env.location_sample_rate = args.location_sample_rate

    #Create archives store for trials
    global archive_store
    archive_store = trial.TrialArchiveStore()
    print("Archive store created")

    # Run the maze experiment trials
    print("Starting the %s maze experiment (Novelty Search), for %d trials"
   % (args.maze, args.trials))
    for t in range(args.trials):
        print("\n\n----- Starting Trial: %d ------" % (t))
        # Create novelty archive and trial archive
        novelty_archive = archive.NoveltyArchive(threshold=args.ns_threshold
   ,
                                                 metric=maze.
   maze_novelty_metric)
        trial_archive = trial.TrialArchive(t)
        print("Archive created for trial %d" % t)

        trial_out_dir = os.path.join(out_dir, str(t))
        os.makedirs(trial_out_dir, exist_ok=True)
        silent = args.trials>1
        #seed = 1687106299 + 6*t
        solution_found = run_experiment( config_file=config_path,
                                         maze_env=maze_env,
                                         novelty_archive=novelty_archive,
                                         trial_archive=trial_archive,
                                         trial_out_dir=trial_out_dir,
                                         n_generations=args.generations,
                                         args=args,
                                         save_results=True,
                                         silent=silent)

        print("\n------ Trial %d complete, solution found: %s ------\n" % (t
   , solution_found))

    print("GLOBAL DATA :")
    archive_store.visualize_global_data(out_dir=out_dir,
                                        view=True)
```

Annexe 14 : Maze NS experiment Agent class

```
#
# This is the definition of a maze navigating agent.
#
import pickle

class Agent:
    """
    This is the maze navigating agent
    """
    def __init__(self, location, heading=0, speed=0, angular_vel=0, radius=8.0, range_finder_range=100.0):
        """
        Creates new Agent with specified parameters.
        Arguments:
            location:               The agent initial position within maze
            heading:                The heading direction in degrees.
            speed:                  The linear velocity of the agent.
            angular_vel:            The angular velocity of the agent.
            radius:                 The agent's body radius.
            range_finder_range:     The maximal detection range for range finder sensors.
        """
        self.heading = heading
        self.speed = speed
        self.angular_vel = angular_vel
        self.radius = radius
        self.range_finder_range = range_finder_range
        self.location = location

        # defining the range finder sensors
        self.range_finder_angles = [-90.0, -45.0, 0.0, 45.0, 90.0, -180.0]

        # defining the radar sensors
        self.radar_angles = [(315.0, 405.0), (45.0, 135.0), (135.0, 225.0), (225.0, 315.0)]

        # the list to hold range finders activations
        self.range_finders = [None] * len(self.range_finder_angles)
        # the list to hold pie-slice radar activations
        self.radar = [None] * len(self.radar_angles)

class AgentRecord:
    """
    The class to hold results of maze navigation simulation for specific
    solver agent. It provides all statistics about the agent at the end
    of navigation run.
    """
    def __init__(self, generation, agent_id, genome):
        """
        Creates new record for specific agent at the specific generation
        of the evolutionary process.
        """
        self.generation = generation
        self.agent_id = agent_id
```

```
52          self.genome = genome
53          # initialize agent's properties
54          self.x = -1
55          self.y = -1
56          self.fitness = -1
57          self.novelty = -1
58          # The flag to indicate whether this agent was able to find maze exit
59          self.hit_exit = False
60          # The ID of species this agent belongs to
61          self.species_id = -1
62          # The age of agent's species at the time of recording
63          self.species_age = -1
64
65  class AgentRecordStore:
66      """
67      The class to control agents record store.
68      """
69      def __init__(self):
70          """
71          Creates new instance.
72          """
73          self.records = []
74
75      def add_record(self, record):
76          """
77          The function to add specified record to this store.
78          Arguments:
79              record: The record to be added.
80          """
81          self.records.append(record)
82
83      def load(self, file):
84          """
85          The function to load records list from the specied file into this
    class.
86          Arguments:
87              file: The path to the file to read agents records from.
88          """
89          with open(file, 'rb') as dump_file:
90              self.records = pickle.load(dump_file)
91
92      def dump(self, file):
93          """
94          The function to dump records list to the specified file from this
    class.
95          Arguments:
96              file: The path to the file to hold data dump.
97          """
98          with open(file, 'wb') as dump_file:
99              pickle.dump(self.records, dump_file)
```

Annexe 15 : Maze NS experiment MazeEnvironment class

```
#
# This is a definition of a maze environment simulation engine. It provides
# routines to read maze configuration and build related simulation environment
# from it. Also it provides method to simulate the behavior of the navigating agent
# and interaction with his sensors.
#
import math

import agent
import geometry

from novelty_archive import NoveltyItem

# The maximal allowed speed for the maze solver agent
MAX_AGENT_SPEED = 3.0

def maze_novelty_metric(first_item, second_item):
    """
    The function to calculate the novelty metric score as a distance between two
    data vectors in provided NoveltyItems
    Arguments:
        first_item:     The first NoveltyItem
        second_item:    The second NoveltyItem
    Returns:
        The novelty metric as a distance between two
        data vectors in provided NoveltyItems
    """
    if not (hasattr(first_item, "data") or hasattr(second_item, "data")):
        return NotImplemented

    if len(first_item.data) != len(second_item.data):
        # can not be compared
        return 0.0

    diff_accum = 0.0
    size = len(first_item.data)
    for i in range(size):
        diff = abs(first_item.data[i] - second_item.data[i])
        diff_accum += diff

    return diff_accum / float(size)

def maze_novelty_metric_euclidean(first_item, second_item):
    """
    The function to calculate the novelty metric score as a distance between two
    data vectors in provided NoveltyItems
    Arguments:
        first_item:     The first NoveltyItem
        second_item:    The second NoveltyItem
    Returns:
```

```
        The novelty metric as a distance between two
        data vectors in provided NoveltyItems
    """
    if not (hasattr(first_item, "data") or hasattr(second_item, "data")):
        return NotImplemented

    if len(first_item.data) != len(second_item.data):
        # can not be compared
        return 0.0

    diff_accum = 0.0
    size = len(first_item.data)
    for i in range(size):
        diff = (first_item.data[i] - second_item.data[i])
        diff_accum += (diff * diff)

    return math.sqrt(diff_accum)

class MazeEnvironment:
    """
    This class encapsulates the maze simulation environment.
    """
    def __init__(self, agent, walls, exit_point, exit_range=5.0):
        """
        Creates new maze environment with specified walls and exit point.
        Arguments:
            agent:      The maze navigating agent
            walls:      The maze walls
            exit_point: The maze exit point
            exit_range: The range arround exit point marking exit area
        """
        self.walls = walls
        self.exit_point = exit_point
        self.exit_range = exit_range
        # The maze navigating agent
        self.agent = agent
        # The flag to indicate if exit was found
        self.exit_found = False
        # The initial distance of agent from exit
        self.initial_distance = self.agent_distance_to_exit()

        # The sample rate of agent position points saving during simulation
    steps.
        self.location_sample_rate = -1

        # Update sensors
        self.update_rangefinder_sensors()
        self.update_radars()

    def agent_distance_to_exit(self):
        """
        The function to estimate distance from maze solver agent to the maze
     exit.
        Returns:
            The distance from maze solver agent to the maze exit.
        """
        return self.agent.location.distance(self.exit_point)

    def test_wall_collision(self, loc):
```

```
108         """
109         The function to test if agent at specified location collides with
    any
110         of the maze walls.
111         Arguments:
112             loc: The new agent location to test for collision.
113         Returns:
114             The True if agent at new location will collide with any of the
    maze walls.
115         """
116         for w in self.walls:
117             if w.distance(loc) < self.agent.radius:
118                 return True
119 
120         return False
121 
122     def create_net_inputs(self):
123         """
124         The function to create the ANN input values from the simulation
    environment.
125         Returns:
126             The list of ANN inputs consist of values get from solver agent
    sensors.
127         """
128         inputs = []
129         # The range finders
130         for ri in self.agent.range_finders:
131             inputs.append(ri)
132 
133         # The radar sensors
134         for rs in self.agent.radar:
135             inputs.append(rs)
136 
137         return inputs
138 
139     def apply_control_signals(self, control_signals):
140         """
141         The function to apply control signals received from control ANN to
    the
142         maze solver agent.
143         Arguments:
144             control_signals: The control signals received from the control
    ANN
145         """
146         self.agent.angular_vel  += (control_signals[0] - 0.5)
147         self.agent.speed        += (control_signals[1] - 0.5)
148 
149         # constrain the speed & angular velocity
150         if self.agent.speed > MAX_AGENT_SPEED:
151             self.agent.speed = MAX_AGENT_SPEED
152 
153         if self.agent.speed < -MAX_AGENT_SPEED:
154             self.agent.speed = -MAX_AGENT_SPEED
155 
156         if self.agent.angular_vel > MAX_AGENT_SPEED:
157             self.agent.angular_vel = MAX_AGENT_SPEED
158 
159         if self.agent.angular_vel < -MAX_AGENT_SPEED:
160             self.agent.angular_vel = -MAX_AGENT_SPEED
```

```
161
162     def update_rangefinder_sensors(self):
163         """
164         The function to update the agent range finder sensors.
165         """
166         for i, angle in enumerate(self.agent.range_finder_angles):
167             rad = geometry.deg_to_rad(angle)
168             # project a point from agent location outwards
169             projection_point = geometry.Point(
170                 x = self.agent.location.x + math.cos(rad) * self.agent.
    range_finder_range,
171                 y = self.agent.location.y + math.sin(rad) * self.agent.
    range_finder_range
172             )
173             # rotate the projection point by the agent's heading angle to
174             # align it with heading direction
175             projection_point.rotate(self.agent.heading, self.agent.location)
176             # create the line segment from the agent location to the
    projected point
177             projection_line = geometry.Line(
178                 a = self.agent.location,
179                 b = projection_point
180             )
181             # set range to maximum detection range
182             min_range = self.agent.range_finder_range
183
184             # now test against maze walls to see if projection line hits any
     wall
185             # and find the closest hit
186             for wall in self.walls:
187                 found, intersection = wall.intersection(projection_line)
188                 if found:
189                     found_range = intersection.distance(self.agent.location)
190                     # we are interested in the closest hit
191                     if found_range < min_range:
192                         min_range = found_range
193
194             # Update sensor value
195             self.agent.range_finders[i] = min_range
196
197     def update_radars(self):
198         """
199         The function to update the agent radar sensors.
200         """
201         target = geometry.Point(self.exit_point.x, self.exit_point.y)
202         # rotate target with respect to the agent's heading to align it with
     heading direction
203         target.rotate(self.agent.heading, self.agent.location)
204         # translate with respect to the agent's location
205         target.x -= self.agent.location.x
206         target.y -= self.agent.location.y
207         # the angle between maze exit point and the agent's heading
    direction
208         angle = target.angle()
209         # find the appropriate radar sensor to be fired
210         for i, r_angles in enumerate(self.agent.radar_angles):
211             self.agent.radar[i] = 0.0 # reset specific radar
212
213             if (angle >= r_angles[0] and angle < r_angles[1]) or (angle +
```

```
    360 >= r_angles[0] and angle + 360 < r_angles[1]):
214                 self.agent.radar[i] = 1.0 # fire the radar
215
216     def update(self, control_signals):
217         """
218         The function to update solver agent position within maze. After
    agent position
219         updated it will be checked to find out if maze exit was reached
    afetr that.
220         Arguments:
221             control_signals: The control signals received from the control
    ANN
222         Returns:
223             The True if maze exit was found after update or maze exit was
    already
224             found in previous simulation cycles.
225         """
226         if self.exit_found:
227             # Maze exit already found
228             return True
229
230         # Apply control signals
231         self.apply_control_signals(control_signals)
232
233         # get X and Y velocity components
234         vx = math.cos(geometry.deg_to_rad(self.agent.heading)) * self.agent.
    speed
235         vy = math.sin(geometry.deg_to_rad(self.agent.heading)) * self.agent.
    speed
236
237         # Update current Agent's heading (we consider the simulation time
    step size equal to 1s
238         # and the angular velocity as degrees per second)
239         self.agent.heading += self.agent.angular_vel
240
241         # Enforce angular velocity bounds by wrapping
242         if self.agent.heading > 360:
243             self.agent.heading -= 360
244         elif self.agent.heading < 0:
245             self.agent.heading += 360
246
247         # find the next location of the agent
248         new_loc = geometry.Point(
249             x = self.agent.location.x + vx,
250             y = self.agent.location.y + vy
251         )
252
253         if not self.test_wall_collision(new_loc):
254             self.agent.location = new_loc
255
256         # update agent's sensors
257         self.update_rangefinder_sensors()
258         self.update_radars()
259
260         # check if agent reached exit point
261         distance = self.agent_distance_to_exit()
262         self.exit_found = (distance < self.exit_range)
263         return self.exit_found
264
```

```
265     def __str__(self):
266         """
267         Returns the nicely formatted string representation of this
    environment.
268         """
269         str = "MAZE\nAgent at: (%.1f, %.1f)" % (self.agent.location.x, self.
    agent.location.y)
270         str += "\nExit  at: (%.1f, %.1f), exit range: %.1f" % (self.
    exit_point.x, self.exit_point.y, self.exit_range)
271         str += "\nWalls [%d]" % len(self.walls)
272         for w in self.walls:
273             str += "\n\t%s" % w
274 
275         return str
276 
277 def read_environment(file_path):
278     """
279     The function to read maze environment configuration from provided
280     file.
281     Arguments:
282         file_path: The path to the file to read maze configuration from.
283     Returns:
284         The initialized maze environment.
285     """
286     num_lines, index = -1, 0
287     walls = []
288     maze_agent, maze_exit = None, None
289     with open(file_path, 'r') as file:
290         for line in file.readlines():
291             line = line.strip()
292             if len(line) == 0:
293                 # skip empty lines
294                 continue
295 
296             if index == 0:
297                 # read the number of line segments
298                 num_lines = int(line)
299             elif index == 1:
300                 # read the agent's position
301                 loc = geometry.read_point(line)
302                 maze_agent = agent.Agent(location=loc)
303             elif index == 2:
304                 # read the agent's initial heading
305                 maze_agent.heading = float(line)
306             elif index == 3:
307                 # read the maze exit location
308                 maze_exit = geometry.read_point(line)
309             else:
310                 # read the walls
311                 wall = geometry.read_line(line)
312                 walls.append(wall)
313 
314             # increment cursor
315             index += 1
316 
317     assert len(walls) == num_lines
318 
319     print("Maze environment configured successfully from the file: %s" %
    file_path)
```

```
320     # create and return the maze environment
321     return MazeEnvironment(agent=maze_agent, walls=walls, exit_point=
    maze_exit)
322
323 def maze_simulation_evaluate(env, net, time_steps, mcns=0.0, n_item=None,
    path_points=None):
324     """
325     The function to evaluate maze simulation for specific environment
326     and controll ANN provided. The results will be saved into provided
327     agent record holder.
328     Arguments:
329         env:            The maze configuration environment.
330         net:            The maze solver agent's control ANN.
331         time_steps:     The number of time steps for maze simulation.
332         mcns:           The minimal criteria fitness value.
333         n_item:         The NoveltyItem to store evaluation results.
334         path_points:    The holder for path points collected during
    simulation. If
335                         provided None then nothing will be collected.
336     Returns:
337         The goal-oriented fitness value, i.e., how close is agent to the
    exit at
338         the end of simulation.
339     """
340     exit_found = False
341     for i in range(time_steps):
342         if maze_simulation_step(env, net):
343             print("Maze solved in %d steps" % (i + 1))
344             exit_found = True
345             break
346
347         if path_points is not None:
348             # collect current position
349             path_points.append(geometry.Point(env.agent.location.x, env.
    agent.location.y))
350
351         # store agent path points at a given sample size rate
352         if (time_steps - i) % env.location_sample_rate == 0 and n_item is
    not None:
353             n_item.data.append(env.agent.location.x)
354             n_item.data.append(env.agent.location.y)
355
356     # store final agent coordinates as genome's novelty characteristics
357     if n_item is not None:
358         n_item.data.append(env.agent.location.x)
359         n_item.data.append(env.agent.location.y)
360
361     # Calculate the fitness score based on distance from exit
362     fitness = 0.0
363     if exit_found:
364         fitness = 1.0
365     else:
366         # Normalize distance to range (0,1]
367         distance = env.agent_distance_to_exit()
368         fitness = (env.initial_distance - distance) / env.initial_distance
369         if fitness <= 0:
370             fitness = 0.01
371
372     # Use minimal criteria fitness value to signal if genome should be
```

```
    included into population
    if fitness < mcns:
        fitness = -1 # mark genome to be excluded

    if n_item is not None:
        n_item.fitness = fitness

    return fitness


def maze_simulation_step(env, net):
    """
    The function to perform one step of maze simulation.
    Arguments:
        env: The maze configuration environment.
        net: The maze solver agent's control ANN
    Returns:
        The True if maze agent solved the maze.
    """
    # create inputs from the current state of the environment
    inputs = env.create_net_inputs()
    # load inputs into controll ANN and get results
    output = net.activate(inputs)
    # apply control signal to the environment and update
    return env.update(output)
```

Annexe 16 : Maze NS experiment configuration file

```
#--- Hyper-parameters for the Single-Pole balancing experiment ---#

[NEAT]
fitness_criterion     = max
fitness_threshold     = 13.5
pop_size              = 500
reset_on_extinction   = True

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.0
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.1
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.1

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = False
initial_connection      = partial_direct 0.5

# node add/remove rates
node_add_prob           = 0.1
node_delete_prob        = 0.1

# network parameters
num_hidden              = 1
num_inputs              = 10
num_outputs             = 2

# node response options
response_init_mean      = 1.0
```

```
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 100
species_elitism      = 1

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.1
min_species_size   = 2
```

Annexe 17 : Maze NS experiment NoveltyArchive class

```
#
# The script providing implementation of structures and functions used in
# the Novelty Search method.
#
from functools import total_ordering

# how many nearest neighbors to consider for calculating novelty score?
KNNNoveltyScore = 15
# The maximal allowed size for fittest items list
FittestAllowedSize = 5
# The minimal number of items to include in the archive unconditionaly
ArchiveSeedAmount = 1

@total_ordering
class NoveltyItem:
    """
    The class to encapsulate information about particular item that
    holds information about novelty score associated with specific
    genome along with auxiliary information. It is used in combination
    with NoveltyArchive
    """
    def __init__(self, generation=-1, genomeId=-1, fitness=-1, novelty=-1):
        """
        Creates new item with specified parameters.
        Arguments:
            generation: The evolution generation when this item was created
            genomeId:   The ID of genome associated with it
            fitness:    The goal-oriented fitness score of genome associated
    with this item
            novelty:    The novelty score of genome
        """
        self.generation = generation
        self.genomeId = genomeId
        self.fitness = fitness
        self.novelty = novelty
        # Indicates whether this item was already added to the archive
        self.in_archive = False
        # The list holding data points associated with this item that will
    be used
        # to calculate distance between this item and any other item. This
    distance
        # will be used to estimate the novelty score associated with the
    item.
        self.data = []

    def __str__(self):
        """
        The function to create string representation
        """
        return "%s: id: %d, at generation: %d, fitness: %f, novelty: %f\
    tdata: %s" % \
            (self.__class__.__name__, self.genomeId, self.generation, self.
    fitness, self.novelty, self.data)

```

```
49      def _is_valid_operand(self, other):
50          return (hasattr(other, "fitness") and
51                  hasattr(other, "novelty"))
52
53      def __lt__(self, other):
54          """
55          Compare if this item is less than supplied other item by
56          goal-oriented fitness value.
57          """
58          if not self._is_valid_operand(other):
59              return NotImplemented
60
61          if self.fitness < other.fitness:
62              return True
63          elif self.fitness == other.fitness:
64              # less novel is less
65              return self.novelty < other.novelty
66          return False
67
68  @total_ordering
69  class ItemsDistance:
70      """
71      Holds information about distance between the two NoveltyItem objects
        based
72      on the nearest neighbour metric.
73      """
74      def __init__(self, first_item, second_item, distance):
75          """
76          Creates new instance for two NoveltyItem objects
77          Arguments:
78              first_item:     The item from which distance is measured
79              second_item:    The item to which distance is measured
80              distance:       The distance value
81          """
82          self.first_item = first_item
83          self.second_item = second_item
84          self.distance = distance
85
86      def _is_valid_operand(self, other):
87          return hasattr(other, "distance")
88
89      def __lt__(self, other):
90          """
91          Compare if the distance in this object is less that in other.
92          """
93          if not self._is_valid_operand(other):
94              return NotImplemented
95
96          return self.distance < other.distance
97
98  class NoveltyArchive:
99      """
100     The novelty archive contains all of the novel items we have encountered
        thus far.
101     """
102     def __init__(self, threshold, metric):
103         """
104         Creates new instance with specified novelty threshold and function
105         defined novelty metric.
```

```
106         Arguments:
107             threshold:  The minimal novelty score of the item to be included
     into this archive.
108             metric:     The function to calculate the novelty score of
    specific genome.
109         """
110         self.novelty_metric = metric
111         self.novelty_threshold = threshold
112 
113         # the minimal possible value of novelty threshold
114         self.novelty_floor = 0.25
115         # the novel items added during current generation
116         self.items_added_in_generation = 0
117         # the counter to keep track of how many generations passed
118         # since we've added to the archive
119         self.time_out = 0
120         # the parameter specifying how many neighbors to look at for the K-
    nearest
121         # neighbor distance estimation to be used in novelty score
122         self.neighbors = KNNNoveltyScore
123         # the current evolutionary generation
124         self.generation = 0
125 
126         # list with all novel items found so far
127         self.novel_items = []
128         # list with all novel items found that is related to the fittest
129         # genomes (using the goal-oriented fitness score)
130         self.fittest_items = []
131 
132     def evaluate_individual_novelty(self, genome, genomes, n_items_map,
    only_fitness=False):
133         """
134         The function to evaluate the novelty score of a single genome within
135         population and update its fitness if appropriate (only_fitness=True)
136         Arguments:
137             genome:         The genome to evaluate
138             genomes:        The current population of genomes
139             n_items_map:    The map of novelty items for the current
    population by genome ID
140             only_fitness:   The flag to indicate if only fitness should be
    calculated and assigned to genome
141                             using the novelty score. Otherwise novelty score
     will be used to accept
142                             genome into novelty items archive.
143         Returns:
144             The calculated novelty score for individual genome.
145         """
146         if genome.key not in n_items_map:
147             print("WARNING! Found Genome without novelty point associated: %
    s" +
148                   "\nNovelty evaluation will be skipped for it. Probably
    winner found!" % genome.key)
149             return
150 
151         item = n_items_map[genome.key]
152         # Check if individual was marked for extinction due to failure to
    meet minimal fitness criterion
153         if item.fitness == -1.0:
154             return -1.0
```

```
155
156         result = 0.0
157         if only_fitness:
158             # assign genome fitness according to the average novelty within
    archive and population
159             result = self._novelty_avg_knn(item=item, genomes=genomes,
    n_items_map=n_items_map)
160         else:
161             # consider adding a NoveltyItem to the archive based on the
    distance to a closest neighbor
162             result = self._novelty_avg_knn(item=item, neighbors=1,
    n_items_map=n_items_map)
163             if result > self.novelty_threshold or len(self.novel_items) <
    ArchiveSeedAmount:
164                 self._add_novelty_item(item)
165
166         # store found values to the novelty item
167         item.novelty = result
168         item.generation = self.generation
169
170         return result
171
172     def update_fittest_with_genome(self, genome, n_items_map):
173         """
174         The function to update list of NovelItems for the genomes with the
    higher
175         fitness scores achieved so far during the evolution.
176         Arguments:
177             genome:          The genome to evaluate
178             n_items_map:     The map of novelty items for the current
    population by genome ID
179         """
180         assert genome.key in n_items_map
181         item = n_items_map[genome.key]
182
183         if len(self.fittest_items) < FittestAllowedSize:
184             # store novelty item into fittest
185             self.fittest_items.append(item)
186             # sort in descending order by fitness
187             self.fittest_items.sort(reverse=True)
188         else:
189             last_item = self.fittest_items[-1]
190             if item.fitness > last_item.fitness:
191                 # store novelty item into fittest
192                 self.fittest_items.append(item)
193                 # sort in descending order by fitness
194                 self.fittest_items.sort(reverse=True)
195                 # remove the less fit item
196                 del self.fittest_items[-1]
197
198     def end_of_generation(self):
199         """
200         The function to update archive state at the end of the generation.
201         """
202         self.generation += 1
203         self._adjust_archive_settings()
204
205     def write_to_file(self, path):
206         """
```

```
        The function to write all NoveltyItems stored in this archive.
        Arguments:
            path: The path to the file where to store NoveltyItems
        """
        with open(path, 'w') as file:
            for ni in self.novel_items:
                file.write("%s\n" % ni)

    def write_fittest_to_file(self, path):
        """
        The function to write the list of NoveltyItems of fittests genomes
        that was collected during the evolution.
        Arguments:
            path: The path to the file where to store NoveltyItems
        """
        with open(path, 'w') as file:
            for ni in self.fittest_items:
                file.write("%s\n" % ni)

    def _add_novelty_item(self, item):
        """
        The function to add specified NoveltyItem to this archive.
        Arguments:
            item: The NoveltyItem to be added
        """
        # add item
        item.in_archive = True
        item.generation = self.generation
        self.novel_items.append(item)
        self.items_added_in_generation += 1

    def _adjust_archive_settings(self):
        """
        The function to adjust the dynamic novelty threshold depending
        on how many have NoveltyItem objects have been added to the archive
    recently
        """
        if self.items_added_in_generation == 0:
            self.time_out += 1
        else:
            self.time_out = 0

        # if no items have been added for the last 10 generations lower the
    threshold
        if self.time_out >= 10:
            self.novelty_threshold *= 0.95
            if self.novelty_threshold < self.novelty_floor:
                self.novelty_threshold = self.novelty_floor
            self.time_out = 0

        # if more than four individuals added in last generation then raise
    threshold
        if self.items_added_in_generation >= 4:
            self.novelty_threshold *= 1.2

        # reset counters
        self.items_added_in_generation = 0

    def _map_novelty(self, item):
```

```
263         """
264         The function to map the novelty metric across the archive against
    provided item
265         Arguments:
266             item: The NoveltyItem to be used for archive mapping.
267         Returns:
268             The list with distances (novelty scores) of provided item from
    items stored in this archive.
269         """
270         distances = [None] * len(self.novel_items)
271         for i, n in enumerate(self.novel_items):
272             distances[i] = ItemsDistance(
273                 first_item = n,
274                 second_item = item,
275                 distance = self.novelty_metric(n, item))
276 
277         return distances
278 
279     def _map_novelty_in_population(self, item, genomes, n_items_map):
280         """
281         The function to map the novelty metric across the archive and the
    current population
282         against the provided item.
283         Arguments:
284             item:        The NoveltyItem to be used for archive mapping.
285             genomes:     The list of genomes from current population.
286             n_items_map: The map of novelty items for the current population
     by genome ID.
287         Returns:
288             The list with distances (novelty scores) of provided item from
    items stored in this archive
289             and from the novelty items associated with genomes in current
    population.
290         """
291         # first, map item against the archive
292         distances = self._map_novelty(item)
293 
294         # second, map item against the population
295         for genome_id, _ in genomes:
296             if genome_id in n_items_map:
297                 gen_item = n_items_map[genome_id]
298                 distance = ItemsDistance(
299                     first_item = gen_item,
300                     second_item = item,
301                     distance = self.novelty_metric(gen_item, item)
302                 )
303                 distances.append(distance)
304 
305         return distances
306 
307     def _novelty_avg_knn(self, item, n_items_map, genomes=None, neighbors=
    None):
308         """
309         The function to calculate the novelty score of a given item within
    the provided population if any
310         using a K-nearest neighbor algorithm.
311         Argumnets:
312             item:        The NoveltyItem to calculate the score
313             n_items_map: The map of novelty items for the current population
```

```
    by genome ID
            genomes:     The list of genomes from population or None
            neighbors:   The number of neighbors to use for calculation (None - to use archive settings)
        Returns:
            The density within the vicinity of the provided NoveltyItem calculated using the K-nearest neighbor
            algorithm. This density can be used either as a novelty score value or as a fitness value.
        """
        distances = None
        if genomes is not None:
            distances = self._map_novelty_in_population(item=item, genomes=genomes, n_items_map=n_items_map)
        else:
            distances = self._map_novelty(item=item)

        # sort by distance (novelty) in ascending order - the minimal first
        distances.sort()
        # if neighbors size not set - use value from archive parameters
        if neighbors is None:
            neighbors = self.neighbors

        density, weight, distance_sum = 0.0, 0.0, 0.0
        length = len(distances)
        if length >= ArchiveSeedAmount:
            length = neighbors
            if len(distances) < length:
                # the number of mapped distances is less than number of neighbors
                length = len(distances)
            i = 0
            while weight < float(neighbors) and i < length:
                distance_sum += distances[i].distance
                weight += 1.0
                i += 1

            # finding the average
            if weight > 0:
                density = distance_sum / weight

        return density
```

Annexe 18 : Maze NS experiment TrialsArchive class

```
#
# The script providing implementation of structures and functions used in
    analysis of data about several trials
#

import numpy as np
import matplotlib.pyplot as plt
import os

class TrialArchive:
    """
    The class to hold record of the performances of the algorithm in one
    trial
    """
    def __init__(self, trial_id):
        """
        Creates the record of one trial
        """
        self.trial_id = trial_id
        self.success_gen = -1
        self.avg_fitness = []
        self.best_fitness = []
        self.best_genome_complexity = -1
        self.seed = -1
        self.gen_max = -1

class TrialArchiveStore:
    """
    The class to hold records of all the trials of one experiment
    """
    def __init__(self):
        """
        Create the new store
        """
        self.archives = []

    def add_archive(self, archive):
        """
        The function to add a new archive to the store
        """
        self.archives.append(archive)

    def visualize_global_data(self, out_dir, view=True):
        """
        The function to show every stats about the experiment
        """
        print("\nGLOBAL STATISTICS :\n")
        self.show_gen()
        self.show_genome_complexity()
        self.plot_avg_fitness(out_dir=out_dir,
                              view=view)
        self.plot_best_fitness(out_dir=out_dir,
                               view=view)

```

```
53      def show_gen(self):
54          """
55          Function that prints for each trials the seed, the number of
    generation before one successful
56          Also prints the average number of generations
57          """
58          success_gen_list =[]
59          sol_exists = False
60
61          for trial in self.archives:
62
63              if trial.success_gen != -1:
64                  success_gen_list.append(trial.success_gen)
65                  sol_exists = True
66                  print("Trial n %d (seed : %d) finds a successful solution
    at generation n %d" % (trial.trial_id, trial.seed, trial.success_gen))
67              else:
68                  print("Trial n %d doesn't fin a successful solution after %
    d generations" % (trial.trial_id, trial.gen_max))
69
70          if sol_exists:
71              avg_number_gen = np.mean(success_gen_list)
72              print("The average number of generation before a successful
    solution is %d" % avg_number_gen)
73          else:
74              print("No trial managed to find a solution in less than %d
    generations" % trial.gen_max)
75
76      def show_genome_complexity(self):
77          """
78          Function that prints genome complexity of the best genome in each
    trial, then the average of all the trials
79          """
80          genome_complexity_list = np.empty([len(self.archives),2])
81          sol_exists = False
82
83          for trial in self.archives:
84              if trial.success_gen != -1:
85                  genome_complexity_list[trial.trial_id] = trial.
    best_genome_complexity
86                  sol_exists = True
87                  print("The best genome of trial n %d has a complexity of (%
    d, %d)" % (trial.trial_id, trial.best_genome_complexity[0], trial.
    best_genome_complexity[1]))
88              else:
89                  print("Trial n %d doesn't fin a successful solution after %
    d generations" % (trial.trial_id, trial.gen_max))
90
91          if sol_exists:
92              avg_complexity = np.mean(genome_complexity_list, axis=0)
93              print("The average complexity of the best genome in each trial
    is (%.02f, %.02f)" % (avg_complexity[0], avg_complexity[1]))
94          else:
95              print("No trial managed to find a solution in less than %d
    generations" % trial.gen_max)
96
97      def plot_avg_fitness(self, out_dir, view=True):
98          """
99          Function that plots the fitness average of every trials, and the
```

```
    average of every trials
        """
        max_gen = -1

        for trial in self.archives:
            if trial.success_gen > max_gen:
                max_gen= trial.success_gen

        if max_gen != -1:
            for trial in self.archives:
                generation = range(trial.gen_max)
                plt.plot(generation[:len(trial.avg_fitness)], trial.
    avg_fitness, label='Trial  n  %d' % trial.trial_id)

        else:
            print("No trial managed to find a solution in less than %d
    generations" % trial.gen_max)
            return

        avg_fitness_list = []
        for i in generation:
            avg_fitness_i = []
            i_is_empty = True

            for trial in self.archives:
                if len(trial.avg_fitness) > i:
                    i_is_empty = False
                    avg_fitness_i.append(trial.avg_fitness[i])

            if i_is_empty:
                break
            else:
                avg_fitness_list.append(np.mean(avg_fitness_i))

        plt.plot(generation[:len(avg_fitness_list)], avg_fitness_list, '--',
     label='Average average fitness')

        plt.title('Average fitness in each trial')
        plt.xlabel('Generations')
        plt.ylabel('Average fitness')
        plt.grid()
        plt.legend()

        plt.savefig(os.path.join(out_dir, 'avg_fitness.svg'))

        if view:
            plt.show()

        plt.close()

    def plot_best_fitness(self, out_dir, view=True):
        """
        Function that plots the fitness average of every trials, and the
    average of every trials
        """
        max_gen = -1

        for trial in self.archives:
            if trial.success_gen > max_gen:
```

```
154                 max_gen= trial.success_gen
155
156         if max_gen != -1:
157             for trial in self.archives:
158                 generation = range(trial.gen_max)
159                 plt.plot(generation[:len(trial.best_fitness)], trial.
    best_fitness, label='Trial  n  %d' % trial.trial_id)
160
161         else:
162             print("No trial managed to find a solution in less than %d
    generations" % trial.gen_max)
163             return
164
165         best_fitness_list = []
166         for i in generation:
167             best_fitness_i = []
168             i_is_empty = True
169
170             for trial in self.archives:
171                 if len(trial.best_fitness) > i:
172                     i_is_empty = False
173                     best_fitness_i.append(trial.best_fitness[i])
174
175             if i_is_empty:
176                 break
177             else:
178                 best_fitness_list.append(np.mean(best_fitness_i))
179
180         plt.plot(generation[:len(best_fitness_list)], best_fitness_list, '--
    ', label='Average best fitness')
181
182         plt.title('Best fitness in each trial')
183         plt.xlabel('Generations')
184         plt.ylabel('Best fitness')
185         plt.grid()
186         plt.legend()
187
188         plt.savefig(os.path.join(out_dir, 'best_fitness.svg'))
189
190         if view:
191             plt.show()
192
193         plt.close()
```

Annexe 19 : Maze NS experiment Visualization methods

```
#Copyright (c) 2007-2011, cesar.gomes and mirrorballu2
#Copyright (c) 2015-2017, CodeReclaimers, LLC
#
#Redistribution and use in source and binary forms, with or without modification, are permitted provided that the
#following conditions are met:
#
#1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following
#disclaimer.
#
#2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following
#disclaimer in the documentation and/or other materials provided with the distribution.
#
#3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products
#derived from this software without specific prior written permission.
#
#THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
#INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
#DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
#SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
#LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
#CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
#SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
from __future__ import print_function

import copy
import warnings
import random
import argparse
import os

import graphviz
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
import numpy as np

import imageio

import geometry
import agent
import maze_environment as maze

```

```
43 def plot_stats(statistics, ylog=False, view=False, filename='avg_fitness.svg
   '):
44     """ Plots the population's average and best fitness. """
45     if plt is None:
46         warnings.warn("This display is not available due to a missing
   optional dependency (matplotlib)")
47         return
48 
49     generation = range(len(statistics.most_fit_genomes))
50     best_fitness = [c.fitness for c in statistics.most_fit_genomes]
51     avg_fitness = np.array(statistics.get_fitness_mean())
52     stdev_fitness = np.array(statistics.get_fitness_stdev())
53 
54     plt.plot(generation, avg_fitness, 'b-', label="average")
55     plt.plot(generation, avg_fitness - stdev_fitness, 'g-.', label="-1 sd")
56     plt.plot(generation, avg_fitness + stdev_fitness, 'g-.', label="+1 sd")
57     plt.plot(generation, best_fitness, 'r-', label="best")
58 
59     plt.title("Population's average and best fitness")
60     plt.xlabel("Generations")
61     plt.ylabel("Fitness")
62     plt.grid()
63     plt.legend(loc="best")
64     if ylog:
65         plt.gca().set_yscale('symlog')
66 
67     plt.savefig(filename)
68     if view:
69         plt.show()
70 
71     plt.close()
72 
73 def plot_species(statistics, view=False, filename='speciation.svg'):
74     """ Visualizes speciation throughout evolution. """
75     if plt is None:
76         warnings.warn("This display is not available due to a missing
   optional dependency (matplotlib)")
77         return
78 
79     species_sizes = statistics.get_species_sizes()
80     num_generations = len(species_sizes)
81     curves = np.array(species_sizes).T
82 
83     fig, ax = plt.subplots()
84     ax.stackplot(range(num_generations), *curves)
85 
86     plt.title("Speciation")
87     plt.ylabel("Size per Species")
88     plt.xlabel("Generations")
89 
90     plt.savefig(filename)
91 
92     if view:
93         plt.show()
94 
95     plt.close()
96 
97 
98 def draw_net(config, genome, view=False, filename=None, directory=None,
```

```
    node_names=None, show_disabled=True, prune_unused=False,
99              node_colors=None, fmt='svg'):
100     """ Receives a genome and draws a neural network with arbitrary topology
    . """
101     # Attributes for network nodes.
102     if graphviz is None:
103         warnings.warn("This display is not available due to a missing
    optional dependency (graphviz)")
104         return
105 
106     if node_names is None:
107         node_names = {}
108 
109     assert type(node_names) is dict
110 
111     if node_colors is None:
112         node_colors = {}
113 
114     assert type(node_colors) is dict
115 
116     node_attrs = {
117         'shape': 'circle',
118         'fontsize': '9',
119         'height': '0.2',
120         'width': '0.2'}
121 
122     dot = graphviz.Digraph(format=fmt, node_attr=node_attrs)
123 
124     inputs = set()
125     for k in config.genome_config.input_keys:
126         inputs.add(k)
127         name = node_names.get(k, str(k))
128         input_attrs = {'style': 'filled', 'shape': 'box', 'fillcolor':
    node_colors.get(k, 'lightgray')}
129         dot.node(name, _attributes=input_attrs)
130 
131     outputs = set()
132     for k in config.genome_config.output_keys:
133         outputs.add(k)
134         name = node_names.get(k, str(k))
135         node_attrs = {'style': 'filled', 'fillcolor': node_colors.get(k, '
    lightblue')}
136 
137         dot.node(name, _attributes=node_attrs)
138 
139     if prune_unused:
140         connections = set()
141         for cg in genome.connections.values():
142             if cg.enabled or show_disabled:
143                 connections.add((cg.in_node_id, cg.out_node_id))
144 
145         used_nodes = copy.copy(outputs)
146         pending = copy.copy(outputs)
147         while pending:
148             new_pending = set()
149             for a, b in connections:
150                 if b in pending and a not in used_nodes:
151                     new_pending.add(a)
152                     used_nodes.add(a)
```

```
            pending = new_pending
    else:
        used_nodes = set(genome.nodes.keys())

    for n in used_nodes:
        if n in inputs or n in outputs:
            continue

        attrs = {'style': 'filled',
                 'fillcolor': node_colors.get(n, 'white')}
        dot.node(str(n), _attributes=attrs)

    for cg in genome.connections.values():
        if cg.enabled or show_disabled:
            #if cg.input not in used_nodes or cg.output not in used_nodes:
            #    continue
            input, output = cg.key
            a = node_names.get(input, str(input))
            b = node_names.get(output, str(output))
            style = 'solid' if cg.enabled else 'dotted'
            color = 'green' if cg.weight > 0 else 'red'
            width = str(0.1 + abs(cg.weight / 5.0))
            dot.edge(a, b, _attributes={'style': style, 'color': color, '
    penwidth': width})

    dot.render(filename, directory, view=view)

    return dot

def draw_agent_path(maze_env, path_points, genome, filename=None, view=False
    , show_axes=False, width=400, height=400, fig_height=4):
    """
    The function to draw path of the maze solver agent through the maze.
    Arguments:
        maze_env:       The maze environment configuration.
        path_points:    The list of agent positions during simulation.
        genome:         The genome of solver agent.
        filename:       The name of file to store plot.
        view:           The flag to indicate whether to view plot.
        width:          The width of drawing in pixels
        height:         The height of drawing in pixels
        fig_height:      The plot figure height in inches
    """
    # initialize plotting
    fig, ax = plt.subplots()
    fig.set_dpi(100)
    fig_width = fig_height * (float(width)/float(height )) - 0.2
    #print("Plot figure width: %.1f, height: %.1f" % (fig_width, fig_height)
    )
    fig.set_size_inches(fig_width, fig_height)
    ax.set_xlim(0, width)
    ax.set_ylim(0, height)

    ax.set_title('Genome ID: %s, Path Length: %d' % (genome.key, len(
    path_points)))
    # draw path
    for p in path_points:
        circle = plt.Circle((p.x, p.y), 2.0, facecolor='b')
        ax.add_patch(circle)
```

```
208
209     # draw maze
210     _draw_maze_(maze_env, ax)
211
212     # turn off axis rendering
213     if not show_axes:
214         ax.axis('off')
215
216     # Invert Y axis to have coordinates origin at the top left
217     ax.invert_yaxis()
218
219     # Save figure to file
220     if filename is not None:
221         plt.savefig(filename)
222
223     if view:
224         plt.show()
225
226     plt.close()
227
228 def animate_agent_path(maze_env, path_points, genome, dirname, show_axes=
    False, width=400, height=400, fig_height=4):
229     """
230     The function to draw path of the maze solver agent through the maze.
231     Arguments:
232         maze_env:       The maze environment configuration.
233         path_points:    The list of agent positions during simulation.
234         genome:         The genome of solver agent.
235         filename:       The name of file to store plot.
236         width:          The width of drawing in pixels
237         height:         The height of drawing in pixels
238         fig_height:      The plot figure height in inches
239     """
240
241
242     # animate path
243     for i,p in enumerate(path_points):
244
245         # initialize plotting
246         fig, ax = plt.subplots()
247         fig.set_dpi(100)
248         fig_width = fig_height * (float(width)/float(height )) - 0.2
249         #print("Plot figure width: %.1f, height: %.1f" % (fig_width,
    fig_height))
250         fig.set_size_inches(fig_width, fig_height)
251         ax.set_xlim(0, width)
252         ax.set_ylim(0, height)
253
254         ax.set_title('Genome ID: %s, Path Length: %d, Image  n  : %03d' % (
    genome.key, len(path_points), i))
255
256         #draw image
257         circle = plt.Circle((p.x, p.y), 2.0, facecolor='b')
258         ax.add_patch(circle)
259
260         # draw maze
261         _draw_maze_(maze_env, ax)
262
263         # turn off axis rendering
```

```
264         if not show_axes:
265             ax.axis('off')
266 
267         # Invert Y axis to have coordinates origin at the top left
268         ax.invert_yaxis()
269 
270         #save image
271         dir_images = os.path.join(dirname, 'images_gif')
272         os.makedirs(dir_images, exist_ok=True)
273         filename_image = os.path.join(dir_images, 'image %03d' % i)
274         plt.savefig(filename_image)
275 
276         plt.close()
277 
278     #create gif
279     path_gif = os.path.join(dirname, 'genome_%d_animation.gif' % genome.key)
280     with imageio.get_writer(path_gif, mode='I') as writer:
281         for filename in os.listdir(dir_images):
282             path_image = os.path.join(dir_images, filename)
283             image = imageio.imread(path_image)
284             writer.append_data(image)
285 
286             os.remove(path_image)
287 
288     os.removedirs(dir_images)
289 
290 
291 
292 
293 def draw_maze_records(maze_env, records, best_threshold=0.8, filename=None,
    view=False, show_axes=False, width=400, height=400, fig_height=7):
294     """
295     The function to draw maze with recorded agents positions.
296     Arguments:
297         maze_env:       The maze environment configuration.
298         records:        The records of solver agents collected during NEAT
    execution.
299         best_threshold: The minimal fitness of maze solving agent's species
    to be included into the best ones.
300         filename:       The name of file to store plot.
301         view:           The flag to indicate whether to view plot.
302         width:          The width of drawing in pixels
303         height:         The height of drawing in pixels
304         fig_height:      The plot figure height in inches
305     """
306     # find the distance threshold for the best species
307     dist_threshold = maze_env.agent_distance_to_exit() * (1.0 -
    best_threshold)
308     # generate color palette and find the best species IDS
309     max_sid = 0
310     for r in records:
311         if r.species_id > max_sid:
312             max_sid = r.species_id
313     colors = [None] * (max_sid + 1)
314     sp_idx = [False] * (max_sid + 1)
315     best_sp_idx = [0] * (max_sid + 1)
316     for r in records:
317         if not sp_idx[r.species_id]:
318             sp_idx[r.species_id] = True
```

```
            rgb = (random.random(), random.random(), random.random())
            colors[r.species_id] = rgb
        if maze_env.exit_point.distance(geometry.Point(r.x, r.y)) <=
    dist_threshold:
            best_sp_idx[r.species_id] += 1

    # initialize plotting
    fig = plt.figure()
    fig.set_dpi(100)
    fig_width = fig_height * (float(width)/float(2.0 * height )) - 0.2
    print("Plot figure width: %.1f, height: %.1f" % (fig_width, fig_height))
    fig.set_size_inches(fig_width, fig_height)
    ax1, ax2 = fig.subplots(2, 1, sharex=True)
    ax1.set_xlim(0, width)
    ax1.set_ylim(0, height)
    ax2.set_xlim(0, width)
    ax2.set_ylim(0, height)

    # draw species
    n_best_species = 0
    for i, v in enumerate(best_sp_idx):
        if v > 0:
            n_best_species += 1
            _draw_species_(records=records, sid=i, colors=colors, ax=ax1)
        else:
            _draw_species_(records=records, sid=i, colors=colors, ax=ax2)

    ax1.set_title('fitness >= %.1f, species: %d' % (best_threshold,
    n_best_species))
    ax2.set_title('fitness < %.1f' % best_threshold)

    # draw maze
    _draw_maze_(maze_env, ax1)
    _draw_maze_(maze_env, ax2)

    # turn off axis rendering
    if not show_axes:
        ax1.axis('off')
        ax2.axis('off')
    # Invert Y axis to have coordinates origin at the top left
    ax1.invert_yaxis()
    ax2.invert_yaxis()

    # Save figure to file
    if filename is not None:
        plt.savefig(filename)

    if view:
        plt.show()

    plt.close()

def _draw_species_(records, sid, colors, ax):
    """
    The function to draw specific species from the records with
    particular color.
    Arguments:
        records:    The records of solver agents collected during NEAT
    execution.
```

```
375         sid:        The species ID
376         colors:     The colors table by species ID
377         ax:         The figure axis instance
378     """
379     for r in records:
380         if r.species_id == sid:
381             circle = plt.Circle((r.x, r.y), 2.0, facecolor=colors[r.
    species_id])
382             ax.add_patch(circle)
383
384
385 def _draw_maze_(maze_env, ax):
386     """
387     The function to draw maze environment
388     Arguments:
389         maze_env:   The maze environment configuration.
390         ax:         The figure axis instance
391     """
392     # draw maze walls
393     for wall in maze_env.walls:
394         line = plt.Line2D((wall.a.x, wall.b.x), (wall.a.y, wall.b.y), lw
    =1.5)
395         ax.add_line(line)
396
397     # draw start point
398     start_circle = plt.Circle((maze_env.agent.location.x, maze_env.agent.
    location.y),
399                               radius=2.5, facecolor=(0.6, 1.0, 0.6),
    edgecolor='w')
400     ax.add_patch(start_circle)
401
402     # draw exit point
403     exit_circle = plt.Circle((maze_env.exit_point.x, maze_env.exit_point.y),
404                              radius=2.5, facecolor=(1.0, 0.2, 0.0),
    edgecolor='w')
405     ax.add_patch(exit_circle)
406
407
408
409
410
411
412 if __name__ == '__main__':
413     # read command line parameters
414     parser = argparse.ArgumentParser(description="The maze experiment
    visualizer.")
415     parser.add_argument('-m', '--maze', default='medium', help='The maze
    configuration to use.')
416     parser.add_argument('-r', '--records', help='The records file.')
417     parser.add_argument('-o', '--output', help='The file to store the plot.'
    )
418     parser.add_argument('--width', type=int, default=400, help='The width of
     the subplot')
419     parser.add_argument('--height', type=int, default=400, help='The height
    of the subplot')
420     parser.add_argument('--fig_height', type=float, default=7, help='The
    height of the plot figure')
421     parser.add_argument('--show_axes', type=bool, default=False, help='The
    flag to indicate whether to show plot axes.')
```

```
422     args = parser.parse_args()
423 
424     local_dir = os.path.dirname(__file__)
425     if not (args.maze == 'medium' or args.maze == 'hard'):
426         print('Unsupported maze configuration: %s' % args.maze)
427         exit(1)
428 
429     # read maze environment
430     maze_env_config = os.path.join(local_dir, '%s_maze.txt' % args.maze)
431     maze_env = maze.read_environment(maze_env_config)
432 
433     # read agents records
434     rs = agent.AgentRecordStore()
435     rs.load(args.records)
436 
437     # render visualization
438     random.seed(42)
439     draw_maze_records(maze_env,
440                       rs.records,
441                       width=args.width,
442                       height=args.height,
443                       fig_height=args.fig_height,
444                       view=True,
445                       show_axes=args.show_axes,
446                       filename=args.output)
```

Annexe 20 : Maze NS experiment visualization of one particular genome methods

```
#
# The script to visualize one specific genome of a previous experience
#

import argparse
import os

import visualize
import neat
import agent
import maze_environment as maze



def visualize_genome(genome_id, records, maze_env, trial_out_dir, config,
    solver_time_steps=400, width=400, height=400):
    """
    The function to visualise fitness, species_id, and path of a certain
    genome represented by its id
    Arguments:
        genome_id: The id of the genome
        record: the record of the genome
        maze_env: The maze environment configuration
        trial_out_dir:
        config:
        solver_time_steps:
    """
    #find corresponding record (and genome)
    for r in records:
        if r.agent_id == genome_id:
            record = r
            genome = r.genome

    #show species id, age and current generation
    species_id = record.species_id
    species_age = record.species_age
    genome_gen = record.generation

    print("Genome %d is of species %d, aged %d, and runs during generation %
    d" % (genome.key, species_id, species_age, genome_gen))

    #compute and show fitness (and path)
    control_net = neat.nn.FeedForwardNetwork.create(genome, config)
    path_points = []
    evaluate_fitness = maze.maze_simulation_evaluate(env=maze_env,
                                                     net=control_net,
                                                     time_steps=
    solver_time_steps,
                                                     path_points=path_points
    )

```

```
48      print("Evaluated fitness of genome %d: %f" % (genome.key,
    evaluate_fitness))
49
50      #draw net
51      node_names =   {-1:'RF_R', -2:'RF_FR', -3:'RF_F', -4:'RF_FL', -5:'RF_L',
     -6: 'RF_B',
52                      -7:'RAD_F', -8:'RAD_L', -9:'RAD_B', -10:'RAD_R',
53                      0:'ANG_VEL', 1:'VEL'}
54      visualize.draw_net(config, genome,
55              view=True,
56              filename=os.path.join(trial_out_dir, 'Digraph_genome_%d.gv' %
    genome.key),
57              node_names=None,
58              fmt='svg')
59
60      #draw agent path's figure and animation
61      visualize.draw_agent_path(maze_env, path_points, genome,
62                                      view=True,
63                                      width=width,
64                                      height=height,
65                                      filename=os.path.join(trial_out_dir, '
    Genome_%d_path.svg' % genome.key))
66
67      visualize.animate_agent_path(maze_env, path_points, genome,
    trial_out_dir,
68                                      width=width,
69                                      height=height)
70
71  if __name__ == '__main__':
72      # read command line parameters
73      parser = argparse.ArgumentParser(description="The visualizer for a
    specific genome in a previous experiment.")
74      parser.add_argument('-g', '--genome_id', type=int, help='The ID of the
    genome to visualize.')
75      parser.add_argument('-m', '--maze', default='medium', help='The maze
    configuration to use.')
76      parser.add_argument('-r', '--records', help='The records file.')
77      parser.add_argument('-o', '--output', help='The file to store the plot.'
    )
78      parser.add_argument('--width', type=int, default=400, help='The width of
     the subplot')
79      parser.add_argument('--height', type=int, default=400, help='The height
    of the subplot')
80      parser.add_argument('--fig_height', type=float, default=7, help='The
    height of the plot figure')
81      parser.add_argument('--show_axes', type=bool, default=False, help='The
    flag to indicate whether to show plot axes.')
82      parser.add_argument('--solver_time_steps', type=int, default=400, help='
    Maximum number of steps before reaching the end of the maze')
83      args = parser.parse_args()
84
85      local_dir = os.path.dirname(__file__)
86      if not (args.maze == 'medium' or args.maze == 'hard'):
87          print('Unsupported maze configuration: %s' % args.maze)
88          exit(1)
89
90      # read maze environment
91      maze_env_config = os.path.join(local_dir, '%s_maze.txt' % args.maze)
92      maze_env = maze.read_environment(maze_env_config)
```

```
93
94      # find and load config file
95      config_path = os.path.join(local_dir, 'maze_config.ini')
96      config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
97                           neat.DefaultSpeciesSet, neat.DefaultStagnation,
98                           config_path)
99
100     # read agents records
101     rs = agent.AgentRecordStore()
102     rs.load(args.records)
103
104     #run function
105     print('\n')
106     visualize_genome(args.genome_id, rs.records, maze_env, args.output,
    config,
107                      solver_time_steps=args.solver_time_steps,
108                      width=args.width,
109                      height=args.height)
```

Annexe 21 : Orientation Consensus experiment

```
#
# The script to run the orientation consensus experiment
#

import random
import time
import copy
import os
import argparse
import numpy as np

import consensus_environment as env
import agent
import consensus_visualize as visualize
import utils

import neat

# The current working directory
local_dir = os.path.dirname(__file__)
# The directory to store outputs
out_dir = os.path.join(local_dir, 'out')
out_dir = os.path.join(out_dir, 'consensus_objective')

class ConsensusSimulationTrial:
    """
    The class to hold consensus orientation simulator execution parameters and results.
    """
    def __init__(self, consensus_env, population):
        """
        Creates new instance and initialize fileds.
        Arguments:
            consensus_env:   The environment as loaded (randomly placed robots)
            population:      The population for this trial run
        """
        # The initial simulation environment
        self.orig_consensus_environment = consensus_env

        # The record store for evaluated solver agents
        # self.record_store = agent.AgentRecordStore()

        # The NEAT population object
        self.population = population

def eval_genomes(genomes, config):
    """
    The function to evaluate the fitness of each genome in
    the genomes list.
    Arguments:
        genomes: The list of genomes from population in the
                 current generation
        config:  The configuration settings with algorithm
```

```
                hyper-parameters
    """
    for genome_id, genome in genomes:
        genome.fitness = eval_fitness(genome_id, genome, config)

def eval_fitness(genome_id, genome, config, time_steps=600):
    """
    Evaluates fitness of the provided genome.
    Arguments:
        genome_id:  The ID of genome.
        genome:     The genome to evaluate.
        config:     The NEAT configuration holder.
        time_steps: The number of time steps to execute for consensus solver simulation.
    Returns:
        The phenotype fitness score in range (0, 1]
    """
    # run the simulation
    maze_env = copy.deepcopy(trialSim.orig_consensus_environment)

    # create the net with feed-forward neat class or Recurent Network
    if config.genome_config.feed_forward:
        control_net = neat.nn.FeedForwardNetwork.create(genome, config)
    else:
        control_net = neat.nn.RecurrentNetwork.create(genome, config)

    epochs_fitness = []
    for i in range(evaluate_epochs):
        epochs_fitness.append(env.consensus_simulation_evaluate(
                                        env=maze_env,
                                        net=control_net,
                                        time_steps=time_steps))
    fitness_array=np.array(epochs_fitness)
    fitness = fitness_array.mean()

    # Store simulation results into the agent record
    #record = agent.AgentRecord(
    #    generation=trialSim.population.generation,
    #    agent_id=genome_id)
    #record.fitness = fitness
    #record.x = maze_env.agent.location.x
    #record.y = maze_env.agent.location.y
    #record.hit_exit = maze_env.exit_found
    #record.species_id = trialSim.population.species.get_species_id(genome_id)
    #record.species_age = record.generation - trialSim.population.species.get_species(genome_id).created
    # add record to the store
    #trialSim.record_store.add_record(record)

    return fitness

def run_experiment(config_file, consensus_env, trial_out_dir, n_generations=100, silent=False):
    """
    The function to run the experiment against hyper-parameters
    defined in the provided configuration file.
    The winner genome will be rendered as a graph as well as the
    important statistics of neuroevolution process execution.
```

```
    Arguments:
        config_file:    The path to the file with experiment configuration
        consensus_env:  The environment to use in simulation.
        trial_out_dir:  The directory to store outputs for this trial
        n_generations:  The number of generations to execute.
        silent:         If True than no intermediary outputs will be
                        presented until solution is found.
        args:           The command line arguments holder.
    Returns:
        True if experiment finished with successful solver found.
    """

    # set random seed
    seed = int(time.time())
    random.seed(seed)

    # Load configuration.
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         config_file)

    # Create the population, which is the top-level object for a NEAT run.
    p = neat.Population(config)

    # Create the trial simulationf
    global trialSim
    trialSim = ConsensusSimulationTrial(consensus_env=consensus_env,
    population=p)

    # Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    p.add_reporter(neat.Checkpointer(5, filename_prefix='%s/consensus-neat-
    checkpoint-' % trial_out_dir))

    # Run for up to N generations.
    start_time = time.time()
    best_genome = p.run(eval_genomes, n=n_generations)

    elapsed_time = time.time() - start_time

    # Display the best genome among generations.
    print('\nBest genome:\n%s' % (best_genome))

    solution_found = (best_genome.fitness >= config.fitness_threshold)
    if solution_found:
        print("SUCCESS: The orientation consensus solver was found !!!")
    else:
        print("FAILURE: Failed to find the orientation consensus solver !!!"
    )

    # write the record store data
    # rs_file = os.path.join(trial_out_dir, "data.pickle")
    # trialSim.record_store.dump(rs_file)

    # print("Record store file: %s" % rs_file)
    print("Random seed:", seed)
    print("Trial elapsed time: %.3f sec" % (elapsed_time))
```

```
164
165     # Visualize the experiment results
166     if not silent or solution_found:
167         node_names =   {-1:'MODE', -2:'TETA_TX', -3:'TETA_RX', -4:'RCV_1',
    -5:'RCV_2',
168                        0:'MODE', 1:'ANG_VEL', 2:'SEN_1', 3:'SEN_2'}
169         visualize.draw_net(config, best_genome, True, node_names=node_names,
     directory=trial_out_dir, fmt='svg')
170         #if args is None:
171         #    visualize.draw_maze_records(maze_env, trialSim.record_store.
    records, view=True)
172         #else:
173         #    visualize.draw_maze_records(maze_env, trialSim.record_store.
    records,
174         #                                view=True,
175         #                                width=args.width,
176         #                                height=args.height,
177         #                                filename=os.path.join(trial_out_dir
    , 'maze_records.svg'))
178         visualize.plot_stats(stats, ylog=False, view=False, filename=os.path
    .join(trial_out_dir, 'avg_fitness.svg'))
179         visualize.plot_species(stats, view=False, filename=os.path.join(
    trial_out_dir, 'speciation.svg'))
180
181     # create the best genome simulation path and visualize it
182     consensus_env = copy.deepcopy(trialSim.orig_consensus_environment)
183
184     # create the best genome net with feed-forward or recurrent neat class
185     if config.genome_config.feed_forward:
186         control_net = neat.nn.FeedForwardNetwork.create(best_genome, config)
187     else:
188         control_net = neat.nn.RecurrentNetwork.create(best_genome, config)
189
190     #best_fitness = 0
191     #for i in range(evaluate_epochs):
192     #    robot_orientation_list = [[] for i in range(len(consensus_env.
    agent_list))]
193     #    evaluate_fitness = env.consensus_simulation_evaluate(consensus_env,
     control_net,
194     #
    robot_orientation_list=robot_orientation_list)
195     #    if evaluate_fitness > best_fitness:
196     #        best_fitness = evaluate_fitness
197     #        best_robot_orientation_list = robot_orientation_list
198
199     #print("Evaluated fitness of best agent: %f" % best_fitness)
200     #visualize.animate_experiment(consensus_env, best_robot_orientation_list
    , best_genome, trial_out_dir)
201
202     # try the experiment with the best genome until one successful run is
    found
203     robot_orientation_list = [[] for i in range(len(consensus_env.agent_list
    ))]
204     tries = 0
205     #loop until a succesful run is found
206     if solution_found:
207         fitness = env.consensus_simulation_evaluate(consensus_env,
    control_net,
208
```

```
    robot_orientation_list=robot_orientation_list)
209         while tries<evaluate_epochs and fitness < config.fitness_threshold:
210
211             print("Run  n  %d unsuccessful. Fitness : %f" % (tries, fitness))
212             robot_orientation_list = [[] for i in range(len(consensus_env.
    agent_list))]
213             fitness = env.consensus_simulation_evaluate(consensus_env,
    control_net,
214
    robot_orientation_list=robot_orientation_list)
215             tries +=1
216
217         if tries<evaluate_epochs:
218             print("Successful run found after %d tries. Fitness = %f" % (
    tries, fitness))
219         else:
220             print("No successful run was found in %d tries with the best
    genome. Visualization of the last try." % evaluate_epochs)
221
222         visualize.animate_experiment(consensus_env, robot_orientation_list,
    best_genome, trial_out_dir)
223         visualize.plot_headings(robot_orientation_list, best_genome, dirname
    =trial_out_dir, view=True)
224
225     return solution_found
226
227
228 if __name__ == '__main__':
229     # read command line parameters
230     parser = argparse.ArgumentParser(description="The maze experiment runner
    .")
231     parser.add_argument('-g', '--generations', default=500, type=int,
232                         help='The number of generations for the evolutionary
     process.')
233     parser.add_argument('-e', '--epochs', default=5, type=int,
234                         help='The number of epochs used to evaluate the
    fitness of one genome.')
235     #parser.add_argument('--width', type=int, default=100, help='The width
    of the records subplot')
236     #parser.add_argument('--height', type=int, default=100, help='The height
     of the records subplot')
237     args = parser.parse_args()
238
239     # create variable evaluate_epoch and set it global
240     global evaluate_epochs
241     evaluate_epochs=args.epochs
242
243     # Determine path to configuration file.
244     config_path = os.path.join(local_dir, 'consensus_config.ini')
245
246     # Clean results of previous run if any or init the ouput directory
247     utils.clear_output(out_dir)
248
249     # Run the experiment
250     # maze_env_config = os.path.join(local_dir, '%s_maze.txt' % args.maze)
251     # maze_env = env.read_environment(maze_env_config)
252
253     consensus_env = env.Environment()
254
```

```
255     # visualize.draw_maze_records(maze_env, None, view=True)
256
257     print("Starting the experiment")
258     run_experiment( config_file=config_path,
259                     consensus_env=consensus_env,
260                     trial_out_dir=out_dir,
261                     n_generations=args.generations)
```

Annexe 22 : Orientation Consensus experiment Agent class

```
#
# This is the definition of an orientation consensus agent.
# It  provides initialization and methods to simulate the behavior of the agents through the interactions with its radars
#

import math
import geometry
import numpy as np

MAX_ANGULAR_VEL = 3.0

class Agent:
    """
    The instance that holds every information of one robot in the consensus map and methods of mobility of the agent
    """

    def __init__(self, location, agent_id, heading=0.0, angular_vel=0.0, radius=8.0, mode=1, radar_range=80):
        """
        Creates new Agent with specified parameters.
        Arguments:
            location:               The agent initial position within maze
            heading:                The heading direction in degrees.
            angular_vel:            The angular velocity of the agent.
            radius:                 The agent's body radius.
            range_finder_range:     The maximal detection range for range finder sensors.
        """
        self.agent_id = agent_id
        self.heading = heading
        self.angular_vel = angular_vel
        self.radius = radius
        self.location = location
        self.mode = 1 # 0=transmission, 1=emission
        self.radar_range=radar_range
        self.msg_rcv_1=None
        self.msg_rcv_2=None
        self.msg_sen_1=0
        self.msg_sen_2=0

        # This variable contains the angle of radars (here is an EPUCK robot)
        self.radar_angle = [[345.0, 375.0], [15.0, 45.0], [45.0, 90.0], [90.0, 150.0], [150.0, 210.0], [210.0, 270.0], [270.0, 315.0], [315.0, 345.0]]

        #This variable contains the angular position of radars (here is an EPUCk robot)
        self.radar_position = [0.0]
        for i in range(1,len(self.radar_angle)):
            self.radar_position.append((self.radar_angle[i][0]+self.radar_angle[i][1])/2)
```

```
46
47          # This variable contains the message received by each radar
48          # Later, only one element of the list can be other than None,
    because the robots must receive only one message at each time step
49          # For now, the information is the angle with which the sender sent
    the message to this agent
50          self.radar = [None] * len(self.radar_angle)
51
52
53      def create_net_inputs(self):
54          """
55          The function to return the ANN input values from the agent.
56          """
57
58          for i,msg in enumerate(self.radar):
59              if msg != None:
60                  angle_of_reception = self.radar_position[i]
61                  angle_of_emission = msg
62                  break
63
64          self.msg_rcv_1
65
66          inputs = [self.mode, angle_of_reception/360.0, angle_of_emission
    /360.0, self.msg_rcv_1, self.msg_rcv_2]
67          return inputs
68
69
70      def apply_outputs(self, outputs):
71          """
72          The function to change behaviour of agent according to the outputs
    of the ANN
73          """
74
75          # Change mode of the agent
76          if outputs[0] < 0.5:
77              self.mode = 0
78          else:
79              self.mode = 1
80
81          # Change angular velocity of the agent
82          new_ang_vel = outputs[1]-0.5
83          if new_ang_vel >= MAX_ANGULAR_VEL:
84              self.angular_vel = MAX_ANGULAR_VEL
85          elif new_ang_vel < -MAX_ANGULAR_VEL:
86              self.angular_vel = -MAX_ANGULAR_VEL
87          else:
88              self.angular_vel = new_ang_vel
89
90          # Change heading of the agent
91          self.heading += 10* self.angular_vel # an angular velocity of 1
    corresponds to 10 degrees per step
92          if self.heading < 0:
93              self.heading += 360
94          elif self.heading >= 360:
95              self.heading -= 360
96
97          if outputs[2]<1/6:
98              self.msg_sen_1=0
99          elif outputs[2]<3/6:
```

```
100             self.msg_sen_1=1/3
101         elif outputs[2]<5/6:
102             self.msg_sen_1=2/3
103         else:
104             self.msg_sen_1=1
105 
106         if outputs[3]<1/6:
107             self.msg_sen_2=0
108         elif outputs[3]<3/6:
109             self.msg_sen_2=1/3
110         elif outputs[3]<5/6:
111             self.msg_sen_2=2/3
112         else:
113             self.msg_sen_2=1
114 
115     def update_radar(self, sender):
116         """
117         Updates the list radar, by initializing it and putting the msg in
    the right element of the list
118         """
119         # Initializing radar list
120         for radar in self.radar:
121             radar = None
122 
123         # Calculating geometric angle from sender to receiver
124         vect = geometry.Point(self.location.x-sender.location.x, self.
    location.y-sender.location.y)
125         angle = vect.angle()
126 
127         rel_angle = self.heading-angle
128 
129         self.radar[self.find_radar_index(rel_angle)] = sender.
    calculate_msg_to(self)
130 
131         # Transmission of the two-digits message
132         if sender.mode == 0:
133             self.msg_rcv_1 = sender.msg_rcv_1
134             self.msg_rcv_2 = sender.msg_rcv_2
135         elif sender.mode == 1:
136             self.msg_rcv_1 = sender.msg_sen_1
137             self.msg_rcv_2 = sender.msg_sen_2
138         else:
139             print("ERROR: mode not equal 1 or 0")
140 
141 
142 
143     def calculate_msg_to(self, aimed_robot):
144         """
145         Calculates the position of the radar used by sender to send a
    message to an aimed robot
146         """
147 
148         # Calculating geometric angle from sender to receiver
149         vect = geometry.Point(aimed_robot.location.x-self.location.x,
    aimed_robot.location.y-self.location.y)
150         angle = vect.angle()
151 
152         # Substract heading of sender to have the relative angle
153         rel_angle = angle - self.heading
```

```
154         if rel_angle <0:
155             rel_angle += 360
156         elif rel_angle >360:
157             rel_angle -= 360
158 
159         return self.radar_position[self.find_radar_index(rel_angle)]
160 
161     def find_radar_index(self, angle):
162         """
163         Gives the index of the radar covering this angle
164         """
165 
166         for i,span in enumerate(self.radar_angle):
167             if angle > span[0] and angle < span[1]:
168                 return i
169         else:
170             return 0
171 
172         print("ERROR : find_radar_index")
173 
174 
175     def individual_fitness(self, avg_heading):
176         """
177         Returns the individual fitness of the agent, in an environment with
    a specific avg_heading
178         Argument : avg_heading, average angle of the heading in the
    evironment (in degrees)
179         """
180 
181         tetaR = self.heading
182 
183         # Computing absolute value term
184         abs_term = abs(geometry.deg_to_rad(tetaR)-geometry.deg_to_rad(
    avg_heading))
185 
186         # Computing min term
187         min_term = min(2*math.pi - abs_term, abs_term)
188 
189         # Computing left term
190         left_term = 1-min_term/math.pi
191 
192         # Computing right term
193         right_term = 1-abs(self.angular_vel)
194 
195         return left_term * right_term
```

Annexe 23 : Orientation Consensus experiment ConsensusEnvironment class

```
#
# This is a definition of the orientation consensus environment simulation engine.
# It provides the initialization of a random environment and methods of communication between the different agents.
#

import random
import geometry
import math

import agent

import neat

class Environment:
    """
    The instance holding every agents and methods of communication
    """

    def __init__(self, length=100, height=100, N=10):
        """
        Creates a new random environment of length and height given, with N agents placed with random locations and headings, and angular velocity of zero
        """

        self.length=length
        self.height=height

        self.agent_list=[]

        for i in range(N):

            x = random.random() * length
            y = random.random() * height
            location = geometry.Point(x,y)

            heading = random.random() * 360

            robot = agent.Agent(location, i, heading, angular_vel=0)
            self.agent_list.append(robot)

    def communication(self):
        """
        Gives each robot on the map a new message in one of their radar
        """
        for robot in self.agent_list:

            # The list of all robots in range
            robots_in_range = []
            for other_robot in self.agent_list:
```

```
                if other_robot.agent_id != robot.agent_id and robot.location
.distance(other_robot.location) <= 80:
                    robots_in_range.append(other_robot)

            # Randomly selects one robot from which the message will be
received
            random_id = random.randint(0, len(robots_in_range)-1)
            selected_robot = robots_in_range[random_id]

            robot.update_radar(selected_robot)

    def consensus_verified(self):
        """
        Returns : True if all robots in the environment are heading the same
 way, with an error tolerated of 5  .
        """
        min_heading = self.agent_list[0].heading
        max_heading = self.agent_list[0].heading

        for robot in self.agent_list:


            if robot.heading < min_heading:
                min_heading = robot.heading

            elif robot.heading > max_heading:
                max_heading = robot.heading

        return max_heading-min_heading < 5    # doesn't consider the fact
that it can be around 360

    def avg_heading(self):
        """
        Returns the average angle of heading of every robots in the
environment
        """
        r0 = self.agent_list[0]
        sum_point = geometry.Point(math.cos(geometry.deg_to_rad(r0.heading))
, math.sin(geometry.deg_to_rad(r0.heading)))
        for r in self.agent_list[1:]:
            new_point = geometry.Point(math.cos(geometry.deg_to_rad(r.
heading)), math.sin(geometry.deg_to_rad(r.heading)))
            sum_point = geometry.sum_points(sum_point, new_point)

        return sum_point.angle()


    def fitness(self):
        """
        Return the fitness score of the environment
        """
        R = len(self.agent_list)

        sum_r = 0
        avg_heading = self.avg_heading()

        for r in self.agent_list:
            sum_r += r.individual_fitness(avg_heading)

```

```
        return (1/R)*sum_r


def consensus_simulation_evaluate(env, net, time_steps=600,
    robot_orientation_list = None):
    """
    The function to evaluate simulation for specific environment
    and controll ANN provided. The results will be saved into provided
    agent record holder.
    Arguments:
        env: The configuration environment.
        net: The agent's control ANN.
        time_steps: The number of time steps for maze simulation.
    """
    for i in range(time_steps):
        if consensus_simulation_step(env, net, robot_orientation_list):
            print("Consensus reached in %d steps" % (i + 1))
            return 1.0

    # Calculate the fitness score based on distance from exit
    fitness = env.fitness()
    if fitness <= 0.01:
        fitness = 0.01

    return fitness

def consensus_simulation_step(env, net, robot_orientation_list):
    """
    The function to perform one step of consensus orientation simulation.
    Arguments:
    env: The maze configuration environment.
       net: The maze solver agent's control ANN
    Returns:
        The True if every robots are heading the same way, with a 5   error
    tolerated
    """

    # Activate/update communication for this step
    env.communication()

    for i, robot in enumerate(env.agent_list):
        # create inputs from the current state of the robot in environment
        inputs = robot.create_net_inputs()
        # load inputs into controll ANN and get results
        output = net.activate(inputs)
        # apply control signal to the environment and update
        robot.apply_outputs(output)

        if robot_orientation_list != None:
            robot_orientation_list[i].append(robot.heading)

    return env.consensus_verified()
```

Annexe 24 : Orientation Consensus experiment configuration file

```
#--- Hyper-parameters for the Single-Pole balancing experiment ---#

[NEAT]
fitness_criterion     = max
fitness_threshold     = 0.80
pop_size              = 250
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.0
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.1
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = False
initial_connection      = partial_direct 0.5

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 1
num_inputs              = 5
num_outputs             = 4

# node response options
response_init_mean      = 1.0
```

```
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 1

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.1
min_species_size   = 2
```

Annexe 25 : Orientation Consensus experiment visualization methods

```
#Copyright (c) 2007-2011, cesar.gomes and mirrorballu2
#Copyright (c) 2015-2017, CodeReclaimers, LLC
#
#Redistribution and use in source and binary forms, with or without modification, are permitted provided that the
#following conditions are met:
#
#1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following
#disclaimer.
#
#2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following
#disclaimer in the documentation and/or other materials provided with the distribution.
#
#3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products
#derived from this software without specific prior written permission.
#
#THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
#INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
#DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
#SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
#LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
#CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
#SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
from __future__ import print_function

import copy
import warnings
import random
import argparse
import os
import math
import imageio

import graphviz
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
import numpy as np

import geometry
# import agent
# import consensus_environment as env

```

```
43 def plot_stats(statistics, ylog=False, view=False, filename='avg_fitness.svg
   '):
44     """ Plots the population's average and best fitness. """
45     if plt is None:
46         warnings.warn("This display is not available due to a missing
   optional dependency (matplotlib)")
47         return
48 
49     generation = range(len(statistics.most_fit_genomes))
50     best_fitness = [c.fitness for c in statistics.most_fit_genomes]
51     avg_fitness = np.array(statistics.get_fitness_mean())
52     stdev_fitness = np.array(statistics.get_fitness_stdev())
53 
54     plt.plot(generation, avg_fitness, 'b-', label="average")
55     plt.plot(generation, avg_fitness - stdev_fitness, 'g-.', label="-1 sd")
56     plt.plot(generation, avg_fitness + stdev_fitness, 'g-.', label="+1 sd")
57     plt.plot(generation, best_fitness, 'r-', label="best")
58 
59     plt.title("Population's average and best fitness")
60     plt.xlabel("Generations")
61     plt.ylabel("Fitness")
62     plt.grid()
63     plt.legend(loc="best")
64     if ylog:
65         plt.gca().set_yscale('symlog')
66 
67     plt.savefig(filename)
68     if view:
69         plt.show()
70 
71     plt.close()
72 
73 def plot_species(statistics, view=False, filename='speciation.svg'):
74     """ Visualizes speciation throughout evolution. """
75     if plt is None:
76         warnings.warn("This display is not available due to a missing
   optional dependency (matplotlib)")
77         return
78 
79     species_sizes = statistics.get_species_sizes()
80     num_generations = len(species_sizes)
81     curves = np.array(species_sizes).T
82 
83     fig, ax = plt.subplots()
84     ax.stackplot(range(num_generations), *curves)
85 
86     plt.title("Speciation")
87     plt.ylabel("Size per Species")
88     plt.xlabel("Generations")
89 
90     plt.savefig(filename)
91 
92     if view:
93         plt.show()
94 
95     plt.close()
96 
97 
98 def draw_net(config, genome, view=False, filename=None, directory=None,
```

```
    node_names=None, show_disabled=True, prune_unused=False,
99              node_colors=None, fmt='svg'):
100     """ Receives a genome and draws a neural network with arbitrary topology
    . """
101     # Attributes for network nodes.
102     if graphviz is None:
103         warnings.warn("This display is not available due to a missing
    optional dependency (graphviz)")
104         return
105 
106     if node_names is None:
107         node_names = {}
108 
109     assert type(node_names) is dict
110 
111     if node_colors is None:
112         node_colors = {}
113 
114     assert type(node_colors) is dict
115 
116     node_attrs = {
117         'shape': 'circle',
118         'fontsize': '9',
119         'height': '0.2',
120         'width': '0.2'}
121 
122     dot = graphviz.Digraph(format=fmt, node_attr=node_attrs)
123 
124     inputs = set()
125     for k in config.genome_config.input_keys:
126         inputs.add(k)
127         name = node_names.get(k, str(k))
128         input_attrs = {'style': 'filled', 'shape': 'box', 'fillcolor':
    node_colors.get(k, 'lightgray')}
129         dot.node(name, _attributes=input_attrs)
130 
131     outputs = set()
132     for k in config.genome_config.output_keys:
133         outputs.add(k)
134         name = node_names.get(k, str(k))
135         node_attrs = {'style': 'filled', 'fillcolor': node_colors.get(k, '
    lightblue')}
136 
137         dot.node(name, _attributes=node_attrs)
138 
139     if prune_unused:
140         connections = set()
141         for cg in genome.connections.values():
142             if cg.enabled or show_disabled:
143                 connections.add((cg.in_node_id, cg.out_node_id))
144 
145         used_nodes = copy.copy(outputs)
146         pending = copy.copy(outputs)
147         while pending:
148             new_pending = set()
149             for a, b in connections:
150                 if b in pending and a not in used_nodes:
151                     new_pending.add(a)
152                     used_nodes.add(a)
```

```
            pending = new_pending
    else:
        used_nodes = set(genome.nodes.keys())

    for n in used_nodes:
        if n in inputs or n in outputs:
            continue

        attrs = {'style': 'filled',
                 'fillcolor': node_colors.get(n, 'white')}
        dot.node(str(n), _attributes=attrs)

    for cg in genome.connections.values():
        if cg.enabled or show_disabled:
            #if cg.input not in used_nodes or cg.output not in used_nodes:
            #    continue
            input, output = cg.key
            a = node_names.get(input, str(input))
            b = node_names.get(output, str(output))
            style = 'solid' if cg.enabled else 'dotted'
            color = 'green' if cg.weight > 0 else 'red'
            width = str(0.1 + abs(cg.weight / 5.0))
            dot.edge(a, b, _attributes={'style': style, 'color': color, '
    penwidth': width})

    dot.render(filename, directory, view=view)

    return dot


def animate_experiment(consensus_env, robot_orientation_list, genome,
    dirname, width=100, height=100, fig_height=5):
    """
    The function to create an animation of the experiment
    /!\ robot_orientation_list and genome must be related
    """

    arrow_size = 10
    dir_images = os.path.join(dirname, 'images_gif')
    os.makedirs(dir_images, exist_ok=True)

    # Animate path
    for i in range(len(robot_orientation_list[0])):

        # Initialize plotting
        fig, ax = plt.subplots()
        fig.set_dpi(100)
        fig_width = fig_height * (float(width)/float(height)) - 0.5
        fig.set_size_inches(fig_width, fig_height)
        ax.set_xlim(0, width)
        ax.set_ylim(0, height)

        ax.set_title('Image n  %03d / %03d' % (i , len(
    robot_orientation_list[0])))

        # Draw image
        for r, robot_heading in enumerate(robot_orientation_list):
            robot_location = consensus_env.agent_list[r].location
            arrow = plt.arrow(robot_location.x,
```

```
                                robot_location.y,
                                arrow_size * math.cos(geometry.deg_to_rad(
    robot_heading[i])),
                                arrow_size * math.sin(geometry.deg_to_rad(
    robot_heading[i])),
                                width=1.5,
                                length_includes_head=True)
            ax.add_patch(arrow)

        # Draw env
        _draw_env_(consensus_env, ax)

        # Invert Y axis to have coordinates origin at the top left and turn
    off axis rendering
        ax.invert_yaxis()
        ax.axis('off')


        #save_image
        filename_image = os.path.join(dir_images, 'image %03d' % i)
        plt.savefig(filename_image)

        plt.close()

    # Create gif
    path_gif = os.path.join(dirname, 'genome_%d_animation.gif' % genome.key)
    with imageio.get_writer(path_gif, mode='I') as writer:
        for filename in os.listdir(dir_images):
            path_image = os.path.join(dir_images, filename)
            image = imageio.imread(path_image)
            writer.append_data(image)

            #os.remove(path_image)

    #os.removedirs(dir_images)

def plot_headings(robot_orientation_list, genome, dirname=None, view=False):
    """
    Plots the difference to average heading of every robots step by step
    """
    steps = range(len(robot_orientation_list[0]))
    avg_heading = []

    for i in steps:
        headings_in_this_step = np.array([robot_orientation_list[j][i] for j
     in range(len(robot_orientation_list))])
        avg_heading.append(headings_in_this_step.mean())

    avg_heading_array = np.array(avg_heading)
    #plt.plot(steps, avg_heading, 'r-', label="average heading")


    for i,robot_heading in enumerate(robot_orientation_list):
        robot_heading_array = np.array(robot_heading)
        plt.plot(steps, robot_heading_array - avg_heading_array, 'b-', label
    ="robot_%d" % i)

    plt.title("Robots headings for genome n  %d" % genome.key)
    plt.xlabel("Robots headings difference to average")
```

```
263     plt.ylabel("Steps")
264     plt.grid()
265 
266     if dirname!=None:
267         filename = os.path.join(dirname, 'robots_headings.svg')
268         plt.savefig(filename)
269     if view:
270         plt.show()
271 
272     plt.close()
273 
274 
275 def _draw_env_(env, ax):
276     """
277     The function to draw the walls of environment and the points
        representing robots
278     """
279 
280     #draw env walls
281     line = plt.Line2D((0,0),(0,env.length), lw=1.5)
282     ax.add_line(line)
283     line = plt.Line2D((0,0),(env.height,0), lw=1.5)
284     ax.add_line(line)
285     line = plt.Line2D((env.height,0),(env.height,env.length), lw=1.5)
286     ax.add_line(line)
287     line = plt.Line2D((env.height,env.length),(0,env.length), lw=1.5)
288     ax.add_line(line)
289 
290     for robot in env.agent_list:
291         circle = plt.Circle((robot.location.x, robot.location.y), radius
        =2.5)
292         ax.add_patch(circle)
```

Annexe 26 : Orientation Consensus experiment geometry methods

```
#
# Here we define common geometric primitives along with utilities
# allowing to find distance from point to the line, to find intersection
   point
# of two lines, and to find the length of the line in two dimensional
   Euclidean
# space.
#

import math

def deg_to_rad(degrees):
    """
    The function to convert degrees to radians.
    Arguments:
        degrees: The angle in degrees to be converted.
    Returns:
        The degrees converted to radians.
    """
    return degrees / 180.0 * math.pi

def read_point(str):
    """
    The function to read Point from specified string. The point
    coordinates are in order (x, y) and delimited by space.
    Arguments:
        str: The string encoding Point coorinates.
    Returns:
        The Point with coordinates parsed from provided string.
    """
    coords = str.split(' ')
    assert len(coords) == 2
    return Point(float(coords[0]), float(coords[1]))

def read_line(str):
    """
    The function to read line segment from provided string. The coordinates
    of line end points are in order: x1, y1, x2, y2 and delimited by spaces.
    Arguments:
        str: The string to read line coordinates from.
    Returns:
        The parsed line segment.
    """
    coords = str.split(' ')
    assert len(coords) == 4
    a = Point(float(coords[0]), float(coords[1]))
    b = Point(float(coords[2]), float(coords[3]))
    return Line(a, b)

class Point:
    """
    The basic class describing point in the two dimensional Cartesian
   coordinate
    system.
```

```
    """
    def __init__(self, x, y):
        """
        Creates new point at specified coordinates
        """
        self.x = x
        self.y = y

    def angle(self):
        """
        The function to determine angle in degrees of vector drawn from the
        center of coordinates to this point. The angle values is in range
        from 0 to 360 degrees in anticlockwise direction.
        """
        ang = math.atan2(self.y, self.x) / math.pi * 180.0
        if (ang < 0.0):
            # the lower quadrants (3 or 4)
            return ang + 360
        return ang

    def rotate(self, angle, point):
        """
        The function to rotate this point around another point with given
        angle in degrees.
        Arguments:
            angle: The rotation angle (degrees)
            point: The point - center of rotation
        """
        rad = deg_to_rad(angle)
        # translate to have another point at the center of coordinates
        self.x -= point.x
        self.y -= point.y
        # rotate
        ox, oy = self.x, self.y
        self.x = math.cos(rad) * ox - math.sin(rad) * oy
        self.y = math.sin(rad) * ox - math.cos(rad) * oy
        # restore
        self.x += point.x
        self.y += point.y

    def distance(self, point):
        """
        The function to caclulate Euclidean distance between this and given
    point.
        Arguments:
            point: The another point
        Returns:
            The Euclidean distance between this and given point.
        """
        dx = self.x - point.x
        dy = self.y - point.y

        return math.sqrt(dx*dx + dy*dy)

    def __str__(self):
        """
        Returns the nicely formatted string representation of this point.
        """
        return "Point (%.1f, %.1f)" % (self.x, self.y)
```

```
110
111
112 def sum_points(point1, point2):
113     """
114     Return a new point that sums two points in the 2D space
115     """
116     return Point(point1.x + point2.x, point1.y+point2.y)
117
118
119
120 class Line:
121     """
122     The simple line segment between two points. Used to represent maze wals.
123     """
124     def __init__(self, a, b):
125         """
126         Creates new line segment between two points.
127         Arguments:
128             a, b: The end points of the line
129         """
130         self.a = a
131         self.b = b
132
133     def midpoint(self):
134         """
135         The function to find midpoint of this line segment.
136         Returns:
137             The midpoint of this line segment.
138         """
139         x = (self.a.x + self.b.x) / 2.0
140         y = (self.a.y + self.b.y) / 2.0
141
142         return Point(x, y)
143
144     def intersection(self, line):
145         """
146         The function to find intersection between this line and the given
    one.
147         Arguments:
148             line: The line to test intersection against.
149         Returns:
150             The tuple with the first value indicating if intersection was
    found (True/False)
151             and the second value holding the intersection Point or None
152         """
153         A, B, C, D = self.a, self.b, line.a, line.b
154
155         rTop = (A.y - C.y) * (D.x - C.x) - (A.x - C.x) * (D.y - C.y)
156         rBot = (B.x - A.x) * (D.y - C.y) - (B.y - A.y) * (D.x - C.x)
157
158         sTop = (A.y - C.y) * (B.x - A.x) - (A.x - C.x) * (B.y - A.y)
159         sBot = (B.x - A.x) * (D.y - C.y) - (B.y - A.y) * (D.x - C.x)
160
161         if rBot == 0 or sBot == 0:
162             # lines are parallel
163             return False, None
164
165         r = rTop / rBot
166         s = sTop / sBot
```

```
        if r > 0 and r < 1 and s > 0 and s < 1:
            x = A.x + r * (B.x - A.x)
            y = A.y + r * (B.y - A.y)
            return True, Point(x, y)

        return False, None

    def distance(self, p):
        """
        The function to estimate distance to the given point from this line.
        Arguments:
            p: The point to find distance to.
        Returns:
            The distance between given point and this line.
        """
        utop = (p.x - self.a.x) * (self.b.x - self.a.x) + (p.y - self.a.y) *
    (self.b.y - self.a.y)
        ubot = self.a.distance(self.b)
        ubot *= ubot
        if ubot == 0.0:
            return 0.0

        u = utop / ubot
        if u < 0 or u > 1:
            d1 = self.a.distance(p)
            d2 = self.b.distance(p)
            if d1 < d2:
                return d1
            return d2

        x = self.a.x + u * (self.b.x - self.a.x)
        y = self.a.y + u * (self.b.y - self.a.y)
        point = Point(x, y)
        return point.distance(p)

    def length(self):
        """
        The function to calculate the length of this line segment.
        Returns:
            The length of this line segment as distance between its
    endpoints.
        """
        return self.a.distance(self.b)

    def __str__(self):
        """
        Returns the nicely formatted string representation of this line.
        """
        return "Line (%.1f, %.1f) -> (%.1f, %.1f)" % (self.a.x, self.a.y,
    self.b.x, self.b.y)
```