Epigenetic learning of autonomous behaviours in a society of agents

François Suro

PhD Thesis

Laboratoire d'informatique, de robotique et de microélectronique de Montpellier

Supervisors:

Jacques Ferber .... Professeur, Université de Montpellier
Tiberiu Stratulat .... Maître de conférences, Université de Montpellier
Fabien Michel .... Maître de conférences HDR, Université de Montpellier

Examination committee:

Chairman: Olivier Simonin .... Professeur, INSA de Lyon
Reporter: François Charpillet .... Directeur de recherche, LORIA
Examiner: Risto Miikkulainen .... Professor, University of Texas
Examiner: Salima Hassas .... Professeur, Université Claude Bernard Lyon 1
Examiner: Eric Bourreau .... Maître de conférences, Université de Montpellier

Université de Montpellier
Department of Computer Science
2020

Во всяком случае, вся эта ваша технология... все эти домны, колеса... и прочая маета-суета, чтобы меньше работать и больше жрать. Все это костыли, протезы. А человечество существует, чтобы создавать... произведения искусства. Это бескорыстно в отличие от других человеческих действий. Великие иллюзии! Образы абсолютной истины! Вы меня слушаете, Профессор?

Anyway, all this technology of yours ... all these blast-furnaces, wheels ... and other vain junk, so that one could work less and devour more — it's only crutches, artificial limbs. And mankind exists for creating ... works of art... It's, anyway, unselfish, as a contrary to all other human actions. The great illusions... Images of the absolute truth... Are you listening to me, Professor?

A. Tarkovsky

Abstract

Humans and robots are autonomous agents acting within the constraints of the physical world. However, the intelligence and autonomy of humans is far superior to that of machines. Inspired by psychology and neurosciences, developmental robotics aims to give artificial agents the ability to adapt, learn and develop autonomously, in order to reach or even exceed the capabilities of humans. Many research fields are involved in the improvement of sensorimotor skill training, memory systems, emergent representations of symbols and languages, motivational systems, and the development of many learning strategies ranging from exploration to imitation and social learning.

However, most of these research projects are focused on a very specific and limited task. Few of them aim to bring together all aspects of embodied intelligence, from the initial development of behaviours to the interactions with other intelligent agents. There is therefore a real need to study which underlying structures can unify this heterogeneity of goals and methods in a perpetually evolving system.

Our goal is to provide such a structure, capable of learning sensorimotor skills as well as more complex skills that go beyond simple reactive behaviour. The main contribution of this thesis is a hierarchical architecture using modular properties to achieve cumulative skill learning, namely MIND. In MIND, sensory information and coordination commands between skills are both treated as signals, using a connectionist inspired approach.

Starting from preliminary work on social specialization in multi-agent systems, we conduct a series of experiments using a MIND hierarchy to accumulate behaviours, from simple sensorimotor behaviours to social behaviours. We first build complex behaviours based on simple reactive behaviours, then integrate simple memory systems with complex behaviours, and finally use these memory systems to learn social behaviours that replicate our initial model of social specialization.

We show that such an architecture is capable of managing the heterogeneity of the behaviours to be learned and the systems to be coordinated. The use of a connectionist approach, a signal-based system, as the underlying architecture made learning both motor control and decision behaviour possible, and also lead to the emergence of memory representations.

Beyond the benefits of MIND as a support for designing developmental agents, our work shows the feasibility of continuous development and the advantages of embodiment in grounding the emergent behaviour, which supports developmental robotics as an approach to general purpose AI.

Keywords: Developmental agents, Modular architecture, Curriculum learning, Emergent representations, Social specialization

Résumé

Les humains et les robots sont des agents autonomes qui agissent dans les limites du monde physique. Cependant, l'intelligence et l'autonomie des humains est bien supérieure à celle des machines. Inspirée de la psychologie et des neurosciences, la robotique développementale vise à donner aux agents artificiels la capacité de s'adapter, d'apprendre et de se développer de manière autonome, afin d'atteindre, voire de surpasser les capacités humaines. De nombreux domaines sont impliqués dans la recherche de meilleures méthodes d'apprentissage sensorimoteur, de systèmes de mémoire, de représentations émergentes de symboles et de langages, de systèmes de motivation, ainsi que le développement de nombreuses stratégies d'apprentissage allant de l'exploration à l'imitation et à l'apprentissage social.

Cependant, la majorité de ces recherches se concentrent sur un aspect particulier. Très peu d'entre elles s'attaquent au problème de l'intelligence incarnée dans son ensemble, du développement initial à l'interaction avec d'autres agents intelligents. Il existe donc un réel besoin d'étudier quelles structures sous-jacentes peuvent unifier cette hétérogénéité des buts et des moyens techniques dans un système en perpétuelle évolution.

Notre objectif est de fournir une telle structure, capable d'apprendre des compétences sensorimotrices ainsi que des compétences plus complexes allant au-delà du simple comportement réactif. La contribution majeure de cette thèse est une architecture hiérarchique, appelée MIND, utilisant une conception modulaire pour l'apprentissage cumulatif de compétences. Dans MIND, les informations sensorielles et les commandes de coordination entre les compétences sont traitées comme des signaux, en utilisant une approche connexionniste.

À partir d'un travail préliminaire sur la spécialisation sociale, nous menons une série d'expériences utilisant MIND pour accumuler des comportements complexes basés sur des comportements sensorimoteurs simples. Nous intégrons ensuite des systèmes de mémoires pour apprendre des comportements sociaux reproduisant notre modèle initial de spécialisation sociale.

Nous montrons la capacité de MIND à gérer l'hétérogénéité des comportements à apprendre et des systèmes à coordonner. L'utilisation d'une approche inspirée du connexionnisme, basée sur le signal, comme architecture sous-jacente a permis l'apprentissage à la fois de comportements sensorimoteurs et de décision, ainsi que l'émergence de représentations mémoires.

Au-delà des avantages de MIND comme support pour la conception d'agents développementaux, notre travail montre la faisabilité du développement continu et les avantages de l'incarnation dans l'ancrage du comportement émergent, ce qui soutient le point de vue de la robotique développementale comme approche pour une IA généraliste.

Mots-clés: Architecture modulaire, Apprentissage par curriculum,
Agents développementaux, Représentations émergentes, Spécialisation sociale.

Contents

Chapter 1

Introduction

As humans, our conception of intelligence is a function of a specialized organ in charge of solving real world problems which are based in space and time. The great diversity of real world problems, the changing conditions over different scales of time favoured systems that were able to evolve, adapt and finally learn. It is this ability to learn that allowed the human mind to exceed its initial purpose of surviving and prospering in a physical world, to consider abstract problems and understand the nature of our world beyond what is apparent.

In order to produce a system capable of simulating a human mind, Turing introduced in 1950 the idea of giving a simpler system the ability to learn. When subjected to an appropriate course of education, this simple system would accumulate knowledge, and maybe develop into an adult mind (Turing, 1950).

The process of learning, in humans, has been studied in the field of cognitive psychology. Jean Piaget (Piaget, 1954; Piaget and Duckworth, 1970) introduced a theory of cognitive development in humans as a dynamical process of coordination schemes through multiple stages (from sensory-motor schemas to abstract level operations). His main idea is that learning is done progressively through interaction between the children and their environment, more complex tasks being learned on top of simpler tasks. According to Piaget, the complexity of learning should progress along three axes: complexity of the environment, complexity of goals and motivation, and complexity of the required skill and behaviour structure. Piaget's genetic epistemology attempts to explain the origin of complex knowledge and representations on the basis of the developmental process of this epigenetic system.

At the crossroads between cognitive psychology, artificial intelligence and robotics, epigenetic and developmental robotics investigates the development of embodied intelligence in artificial agents. The approach is two-fold: design better autonomous robots using developmental psychology as inspiration, and investigate the theoretical models of cognitive psychology though experiments on simulated agents.

In this approach, the term learning is understood as the development of a situated entity, acquiring a wide range of skills. This definition of learning should not be confused with the industry driven trend in learning, deep learning, which focuses on the short term goal of perfecting the connectionist version of the old expert systems for practical use (Sirignano et al., 2016; Lin et al., 2017; Wang and Xu, 2018; Wu and Zhang, 2016).

When understood as long term research, developmental robotics is tasked with the ambitious goal of creating an artificial agent capable of learning to perform tasks which cannot be anticipated by the designer itself. This capacity of cumulative and life-long learning, of an agent whose lifespan is not as limited as ours, gives it the potential to exceed the initial purpose planned by the designer.

1.1 Context

Interest in learning and self organizing machines can be traced back to the 1950s and the pioneers in cybernetic, but it is not until the end of the XXth century that developmental robotics established itself as a field of research. Coinciding with the end of the second wave of AI (expert systems and symbol manipulation) and the return in favour of connectionism, developmental robotics can be qualified as a bio-inspired approach, seeking novel methodologies from studies in developmental psychology and neurosciences. Early examples include the application of body babbling of infants (Meltzoff and Moore, 1997) to produce a robot capable of associating movements learned through self-exploration to movements presented to it by a human demonstrator, thus displaying neonatal imitation capabilities (Kuniyoshi et al., 2003). Inspired by theories of language development (Halliday, 1975) an anthropomorphic robot developed a proto-language based on expressing its desires (using a specific vocalization to request an object) (Varshavskaya, 2002). This expression of a concept is the emergence of a symbol, and an approach to the symbol grounding problem. The work of Bernstein (Bernstein, 1967) in neuroscience was applied to robotic experiments in motor control (Lungarella and Berthouze, 2002) where it was shown that the progressive increase in degrees of freedom leads to a more robust behaviour, built on simple and efficient movement patterns.

Developmental robotics is based around the idea that learning and developing, which leads to an intelligent system, happens as the self-organization of dynamical interactions among brain, body and environment (Oudeyer, 2012). This idea comes from the simple observation that biological systems are not passively exposed to sensory input, but instead interact actively with their surrounding environment (Lungarella et al., 2003). Developmental robotics supports the claim that embodiment is required for the emergence of intelligence. A number of traditional AI problems can be simplified when embodied, such as image processing which can take advantage of depth by actively changing the point of view. The notion of object emerges from the simultaneous experiences of seeing (spatial bounds) and grasping (impenetrability). From this notion can begin categorization for practical use (food and non-food) which in turn serves as a basis for the emergence of symbols (Lungarella et al., 2003).

In order to create this embodied learning agent, it is given that the learning structures and techniques are task independent, and that the agent must be able to learn any behaviour within the limits of its physical abilities.

Tied to the learning techniques, the agent will require a form of motivational system such as simple extrinsic motivation, for instance hunger/harm which will directly influence the agent to perfect its skills (how do I reach food rapidly and without colliding with obstacles). At a meta-level, intrinsic motivations such as curiosity/boredom can direct the learning strategies (what do I want to learn) or self exploration (babbling and investigation of unexpected results).

Because of the vastness of the possible actions, the agent would benefit from social guidance. In addition to its abilities for discovery driven by its motivations, the agent can be trained and guided by an instructor (comparable to scaffolding), or take advantage of other agents experience by imitation or emulation (follow the well-fed ones).

To help the developmental process, the structuration of knowledge must favour the reuse of previously learnt structure. Previously acquired skills must be available for evaluation, selection and combination into new skills of higher complexity.

In fulfilling all these requirements a number of challenging issues remain. How motivational system and self discovery can coexist with social learning? What learning strategies would allow human-robot scaffolding of behaviour? What underlying structure would allow the accumulation of skills and the emergence of symbols? Can these structures be robust enough to support life-long learning?

> *A fundamental scientific issue to be understood and resolved, which applied equally to human development, is how compositionality, functional hierarchies, primitives, and modularity, at all levels of sensorimotor and social structures, can be formed and leveraged during development. This is deeply linked with the problem of the emergence of symbols, sometimes referred to as the "symbol grounding problem" when it comes to language acquisition. Actually, the very existence and need for symbols in the brain is actively questioned, and alternative concepts, still allowing for compositionality and functional hierarchies are being investigated.*
>
> Oudeyer (2012)

1.2 Contribution

In this thesis we will focus on the structure and architecture supporting the cumulative learning and development of an agent.

We aim to provide a bridge between learning strategies inspired by constructivism, of building skills of higher complexity based on the previously acquired skills, and observations on the modular nature of living systems. Our system must handle the acquisition of simple low level motor skills as well as complex deliberative skills beyond reactive behaviour, using as much previously acquired experience as possible and combining a great diversity of inputs and outputs.

We introduce the Modular Influence Network Design (MIND), an architecture encapsulating low level learning structures into independent modules and providing a simple and universal mechanism for the coordination of modules: the *influence*. Inspired from connectionism, the *influence* mechanism is a signal based control system which emulates the behaviour of low level input-output control. Using the *influence* mechanism, high level skill modules are able to modulate the signal to the actuators of concurrent low level skill modules in order to achieve coordination. Low level skill modules associate sensors to actuators directly, providing a short reflex-like path between sensing and acting which offers a rapid response. Using the same communication method as the sensors and actuators, MIND provides memory modules to extend the abilities of the agent beyond purely reactive behaviour by keeping and evolving internal states, representations and decisions.

Starting from preliminary work on social specialization in multi-agent systems, we conduct a series of experiments using a MIND hierarchy to accumulate behaviours, from simple sensorimotor behaviours to social behaviours. We build complex behaviours based on simple reactive behaviours, we integrate simple memory systems with complex behaviours, and finally we use these memory systems to learn social behaviours that replicate our initial model of social specialization.

From the analysis of these result, we highlight the advantages and limitations of MIND as a support for designing developmental agents and point out the perspectives offered by this work, in the context of developmental robotics as well as its generalization to machine learning and artificial intelligence.

1.3 Manuscript organization

Chapter 2 presents the different aspects involved in **developmental agents** and their related works. This chapter covers the **low level structures** used to represent knowledge and their associated **learning techniques**, the agent **control systems**, their associated **learning strategies** and **motivational systems**, the use and implications of **memory systems**, and Multi-agent **social organization** used to coordinate skilled individual agents.

Chapter 3 presents preliminary work on social specialization around the **MetaCiv** meta-model and the cogniton architecture. The cogniton architecture is a hybrid reactive architecture using **memory elements** and **reinforcement mechanisms** to design models leading to **emergent organization**.

Chapter 4 introduces our main contribution, **Modular Influence Network Design (MIND)**, an architecture **encapsulating skills** in separate modules able to **organize in a hierarchy** to achieve complex tasks. We define the principle of ***influence*** which allows skill modules to coordinate with each other and use indifferently **sensor** modules, **actuator** modules and **memory** modules. This architecture is specifically designed to meet the needs of **epigenetic agents**, **life-long learning** and **continuous evolution** of the system.

Chapter 5 describes the **experimental context**, the implementation of MIND, the simulated agent and environments, the learning algorithms and structures used by skill modules, and the learning process for a skill module inside a MIND hierarchy.

Chapter 6 presents a series of experiments using the **basic principles of MIND** to demonstrate its suitability to **cumulative learning**. We first teach an artificial agent a complex behaviour from scratch, by building a hierarchy of skills of increasing complexity. Then we experiment with focused skill retraining on identified bottlenecks to improve the behaviour of the whole hierarchy. Finally, we add a new constraint to the task, new sensors and add new skills to the hierarchy to extend the functionality of the agent beyond its original purpose.

Chapter 7 presents the use of **variables** in MIND, the **memory system** which provides the skills with the ability to store, retrieve and share information. The first experiment uses a target variable to determine where the agent should go. This variable must be set to the value of one of several orientation sensors, depending on context. The second experiment uses a variable to count steps in a sequence of action. No information on the current step of the sequence can be observed in the environment, this constraint forces the agent to keep track of the current state of the sequence in its own mind.

Chapter 8 presents the use of MIND in a **multi-agent** context. In the first experiment the agents forage for resources with a limited perceptual range and achieve reactive coordination through simple signals. In the last experiment, we replicate the social specialization of the preliminary work. In this experiment the agents must collect a proportional amount of two different resources, leading them to split their population into two groups, each collecting a different kind of resource.

Chapter 9 discusses our **contributions** from the perspective of developmental agents, multi-agent systems and machine learning in general. We also present our plans for **future works** around **MIND** and further research interests.

Chapter 2

Background

In contrast to other fields focusing on learning, developmental robotics aims to create artificial systems with skills that go beyond single-task learning. "The search for flexible autonomous and open-ended multi-task learning system is in essence, a particular re-instantiation of the long-standing research for general-purpose AI" (Lungarella et al., 2003). Developmental robotics, inspired by the only known general purpose intelligence, aims at the emergence of AI through progressive and life-long learning under the constraints of embodied agents. Under the concept of *ongoing emergence* (Prince et al., 2005), a developmental agent must satisfy six criteria:

1. Bootstrapping of initial skills
2. Continuous skill acquisition
3. Autonomous development of values and goal
4. Incorporation of new skills with existing skills
5. Stability of skills
6. Reproducibility (of emergence)

We previously exposed the interdisciplinary aspect of developmental robotics, taking inspiration outside of computer science and robotics and into neurosciences and psychology, but in order to fulfil all the requirements of ongoing emergence, developmental robotics relies on many disciplines within computer science itself. Learning algorithms, data structures, exploration, communication between systems, signal processing ...

Bootstrapping initial skills, and further acquisition of skills (items 1&2) requires a learning system capable to learn single-tasks in the first place. A vast number of learning algorithms exist, some were designed for agents or adapted to agent constraints later. Popular deep learning algorithms are being adapted to embodied context, to solve agent related problems. These existing algorithms in turn are adapted, and coupled to motivational systems, to extend their ability to autonomous learning and a developmental context.

The problem of guiding learning, such as providing a reward or setting a goal (item 3), is investigated under principle of motivation. Motivational systems of increasing complexity are designed. Internal motivation aims to analyse the environment from the point of view of the agent to find out what action is rewarding, instead of relying on a given reward signal as was the case with external motivation. Beyond this simple step forward in autonomous learning, intrinsic motivational systems are designed for autonomous development by guiding the agent not only on how to learn, but also on what to learn. Intrinsic motivation using analogies such as curiosity or flow (interest in tasks neither too easy nor too hard) will drive exploration and help direct the progressive structuration of skills, in order of complexity.

A fundamental issue to be resolved (Oudeyer, 2012) is finding the underlying structures and structure building methods that will accommodate the incorporation of new skills with existing skills (Item 4), following the progressive guidance of motivational systems (Items 2&3). The idea of building upon previous structures, to keep accomplished progress and branch into multiple behaviours involves guaranteeing the stability of acquired skills (Item 5). Learning the skill *HavingCupOfTea*, critical step of many plans (Russell and Norvig, 2009), by building upon *RaiseArm* and *Swallow*, is a problem of coordination of skills, either though combination or sequence.

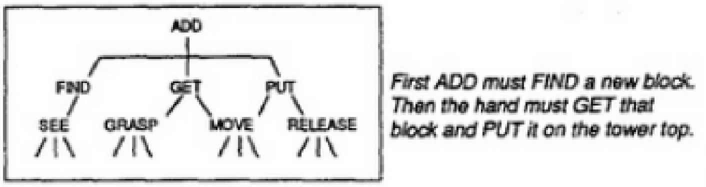


Figure 2.1: A hierarchy of skills used sequentially (Minsky, 1988)

Along with the increase in complexity of skills, developmental agents will need to evolve beyond purely reactive behaviour, which will require some form of internal representations. Internal representations play a role at many levels of the cognitive process, from *fragile* and *working* memory keeping short-term information vital in understanding immediate spatio-temporal problems, to long term memory of past experiences and learned symbols needed in abstract reasoning as we understand it. The acquisition of such symbols through an emergent process and their relation to meaning is an open question in the field of developmental agents referred to as the "symbol grounding problem".

Finally, for a developmental agent to come out of the experimental stage and into the real world, it will require an ability for social interaction, both with its kind and with heterogeneous populations of agents, including biological ones. Studies in the field of multi-agent systems deal with social organization of agent, from simple reactive agent able to exhibit complex emergent social behaviour to more complex simulations including message exchanges and spontaneous creations of groups with defined roles. Social interaction will closely interact with internal representations, in a notion of self for instance, but also in communication of intentions to solve collective problems that might require emergent symbol acquisition to form a proto-language. Social interactions will also play an important part in the learning process, with specifically designed algorithms exploiting social aspects such as imitation, or interaction with a human instructor through a form of communication much more natural than programming algorithms.

In this chapter, we present works on and around developmental robotics:

Section 2.1 describes structures and learning techniques developed for agent, or adapted from other fields.

Section 2.2 presents motivational systems, an element peculiar to developmental robotics used to guide the development of an agent.

Section 2.3 present architectures, structures and learning strategies supporting the development of agents.

Section 2.4 presents memory systems and possible internal representation methods for cognitive architectures.

Section 2.5 presents social interactions for agents in multi-agent systems and emergence of social behaviour.

2.1 Learning agents

Learning systems are of great interest outside developmental robotics, and even outside of behaviour and agent control. The vast resources sunk in the field of machine learning, such as the very publicized deep learning, to improve classifiers for industrial and commercial use will attest of that.

In this section we will focus on learning agents, only taking a detour when some technique can be adapted to the agent context.

Although there exist bridges between different approaches, the learning techniques and structures depend on the approach used for motor control. We will distinguish between two approaches: by motor primitives or by signal.

Using motor primitives has long been favoured, it provides a relatively high level of abstraction and is suited to discrete state-action association. At the end of a deliberative phase, the planner outputs the best motor primitive to be used exclusively ("turn left"). From an AI perspective, this approach can be viewed as symbolic, the primitive expressed being the symbol (atomic).

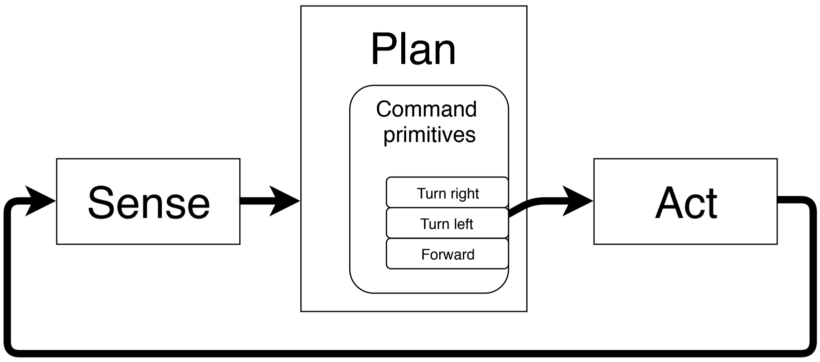


Figure 2.2: A common deliberative paradigm, planning from a set of primitive commands.

A different approach is to use signal directly, as a continuous value. This approach is largely inspired from neuroscience and can be likened to connectionist AI, for its use of signal and providing a result as continuous value. An early example is the Braitenberg vehicles (Braitenberg, 1986), in which the intensity of the signal output of a luminosity sensor directly controls the speed of the wheel of the vehicle.

2.1.1 Reinforcement learning

In a traditional reinforcement learning model the agent interacts with the environment via a perceived *state* and an *action* to be executed. For each *action* executed, the agent receives a *reinforcement signal* (a positive or negative reward). The agent is not given an example of ideal State-Action associations, it must instead map the optimal *action* to the corresponding *state* through interaction with the environment and an interpretation of the *reinforcement signal* it provides.

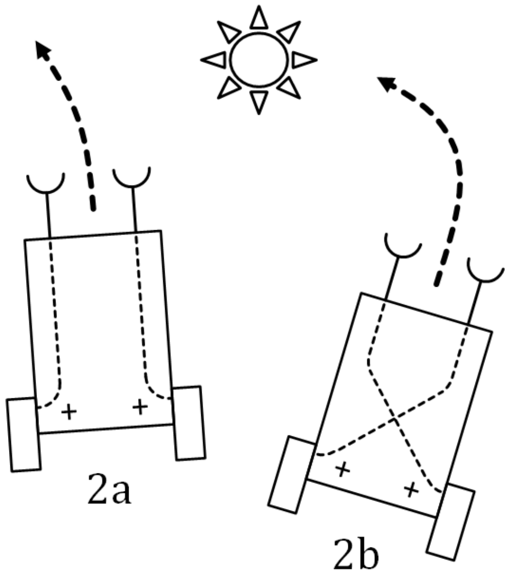


Figure 2.3: Braitenberg vehicles, "knowing" how to reach and avoid a light source (Braitenberg, 1986)

Considering the learning problem from the point of view of discrete states and action, this model tends to be suited to the use of motor primitives, the action being a single and indivisible command.

Choosing the optimal action for the current state means considering the immediate reward it will yield, but also future rewards that can be obtained from the possible actions of the next state. The delayed reward is the reward obtained after a number of steps in the causal chain State-Action-State-Action-etc. The number of future steps taken into account when evaluating the cumulated expected reward for an action is called the *horizon*. A general model of the optimal behaviour can be given as such:

$$\frac{1}{h}\sum_{t=0}^{h}\gamma_t r_t \tag{2.1}$$

With $_t$ the time step, h the horizon which can be finite or "infinite", γ a coefficient $0 \leq \gamma \leq 1$ and an optional $\frac{1}{h}$ if results are to be compared between different horizons.

The γ coefficient can be set at different values or vary between time step. A constant coefficient of 1 will consider all reward of same importance, while a coefficient decreasing in function of time will reflect the increasing uncertainty of future actions.

Q-learning

Q-learning (Watkins, 1989) is a widely used model-free reinforcement algorithm. Q-learning provides a simple formula to update an expected reward function based on instantaneous reward, as such it is independent of the reward function and the exploration strategy.

The reinforcement formula used is as follows:

$$Q_n(s,a) = (1-\alpha_n)Q_{n-1}(s,a) + \alpha_n(r_n + \gamma MAX(Q(s',a'))) \tag{2.2}$$

Where Q is the long-term reward function (Q_{n-1} the function in the previous step), s is the current state, a is the considered action, and s' the state resulting from the action, α_n is the learning rate, γ is the influence of long-term rewards, r_n is the maximum immediate reward, and $MAX(Q(s',a'))$ gives the highest long-term reward value achievable by all actions of the next state (s').

This method is not dependent on a particular model and could be applied to neural networks (Huang et al., 2005). In this case, the neural network is set up as a classifier system, the output neuron of highest value is selected as the result. Its output is a single label representing the chosen action. Q-learning provides a long-term decision, the label of an action, that the neural network memorizes using the *backpropagation* algorithm as a neural classifier would.

2.1.2 Artificial neural network

An Artificial Neural Network (ANN) is a structure able to map several input signals to several output signals, and generate output signals for every possible value of the input signals in the domain.

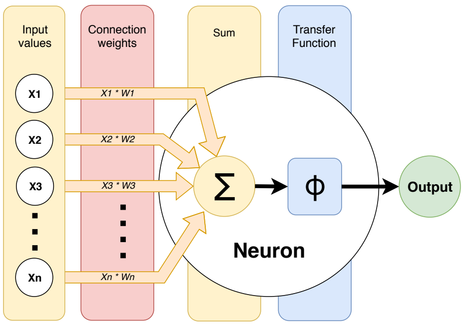


Figure 2.4: The model of a perceptron

The elementary unit of the artificial neural network, the neuron, is based on the model of the perceptron (Rosenblatt, 1958). Using the analogy of neurons and synapses, the perceptron multiplies each input by an associated weight (synaptic throughput), the weighted inputs are summed in the neuron and given to the transfer function to compute the output (Fig. 2.4).

The perceptron is able to represent any linear predictor function by adjusting the weights of its connections. As soon as progress was made in the learning algorithms used to adjust the weights, perceptrons were used in multi-layered networks (Rumelhart et al., 1988).

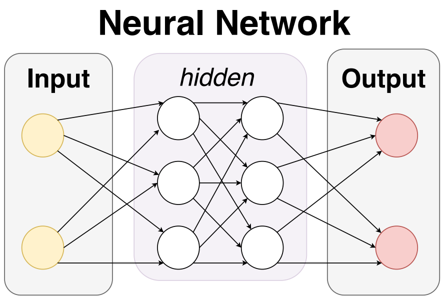


Figure 2.5: Multi-layer perceptron

The network structure of a multi-layered perceptron is a graph composed of an input neuron layer, an output neuron layer and a number of intermediate layers. A layer is a set of neurons that are not connected with each other. Instead, each neuron in a layer is connected to all the neurons in the previous layer from which it will receive the signal. It is also connected to all the neurons in the next layer to which it will send its signal. Following the model of the perceptron, each link between neurons has a weight that alters the value of the transmitted signal. Each neuron combines all the signals it receives into a new signal, applies a transfer function and transmits to subsequent neurons (Fig. 2.5).

The topology of the network can be adapted to suit the nature of the data. In Convolutional Neural Networks (CNN), used in image processing, input neurons are connected according to proximity of the pixel they represent, taking advantage of the spatial nature of the data (Krizhevsky et al., 2012). Recurrent Neural Networks (RNN) use a memory layer keeping previous states of the neurons in order to exploit the sequential nature of some types of data.

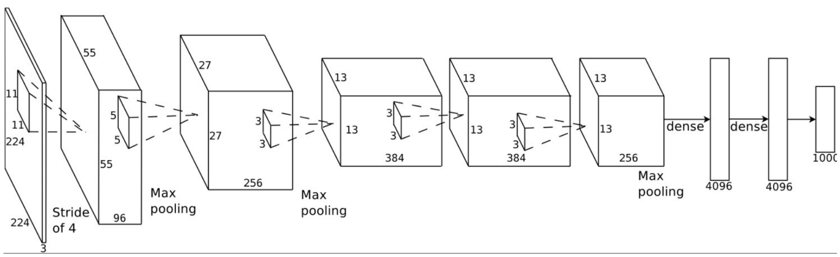


Figure 2.6: Convolutional neural network. From left to right: the image analysed locally to a fully connected network used for final classification (from Krizhevsky et al. (2012)).

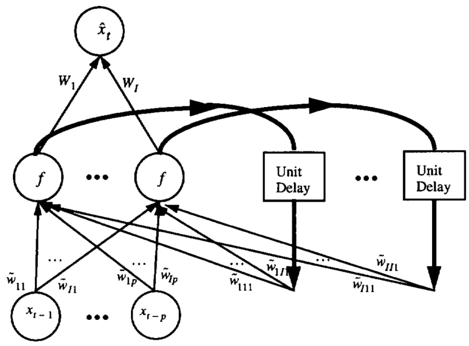


Figure 2.7: Recurrent neural network. On the right the "memory" elements (Unit Delay). (from Connor et al. (1994)).

An artificial neural network can be adapted to agent control in different ways. In the works of Huang Huang et al. (2005) each output neuron is connected to a preprogrammed action (advance, rotate, etc.), the neural network act as a classifier, choosing the action with the strongest output value for the given sensor input. In other works Levine et al. (2015b,a,a); Levine and Abbeel (2014); Robbins (2014) the output neurons directly control the torque or position of each motor, this flow of input signal to output signal is comparable to the approach used in Braitenberg vehicles (Braitenberg, 1986).

Backpropagation

The backpropagation algorithm (Rumelhart et al., 1985) is a neural network learning algorithm using a training set to adjust the weights of the network. A training set is a large collection of input-output pairs representing correct answers expected from the network. However large the collection may be, it can never realistically cover the entire input domain, from the given examples the network will learn to interpolate in order to provide outputs for inputs not presented in the training set. backpropagation is best used with large training sets, on which the learning process is repeated many times. Each time the weights are adjusted in small increments, as any modification of weights to fit a particular solution will impact all other solution. An analogy can be made with centring a cover on a bed, pulling only one corner at the time, each time a corner is pulled, the others are moved slightly out of place.

The backpropagation algorithm operates as follows:

1. Feedforward the input of the training set, keep the result for each neuron, compare the output of the network (O) with the output of the training set(T) to compute the squared error (E):

   $E_n = \frac{1}{2}(O - T)^2$

2. Compute the backpropagated error of the output layer, multiplying the squared error by derivative of the transfer function $f'(\sigma_n)$

   $\delta_n = E_n * f'(\sigma_n)$

3. Compute the error of the previous layer E_{n-1} by summing the square error derivative of all the linked neurons (δ_n) multiplied by the weight of the link (W_n)

   $E_{n-1} = \sum \delta_n * W_n$

4. Compute the backpropagated error of the current layer, multiplying the squared error by derivative of the transfer function $f'(\sigma_{n-1})$

   $\delta_{n-1} = E_{n-1} * f'(\sigma_{n-1})$

   Repeat step 3 and 4 for each layer.

5. Update the weights of the network. To the current weight is subtracted the output of the incoming neuron (O_n) multiplied by the backpropagated error of the following neuron(δ_{n+1}) and the learning rate (μ).

   $W_{n \mapsto n+1} = W_{n \mapsto n+1} - (\mu * O_n * \delta_{n+1})$

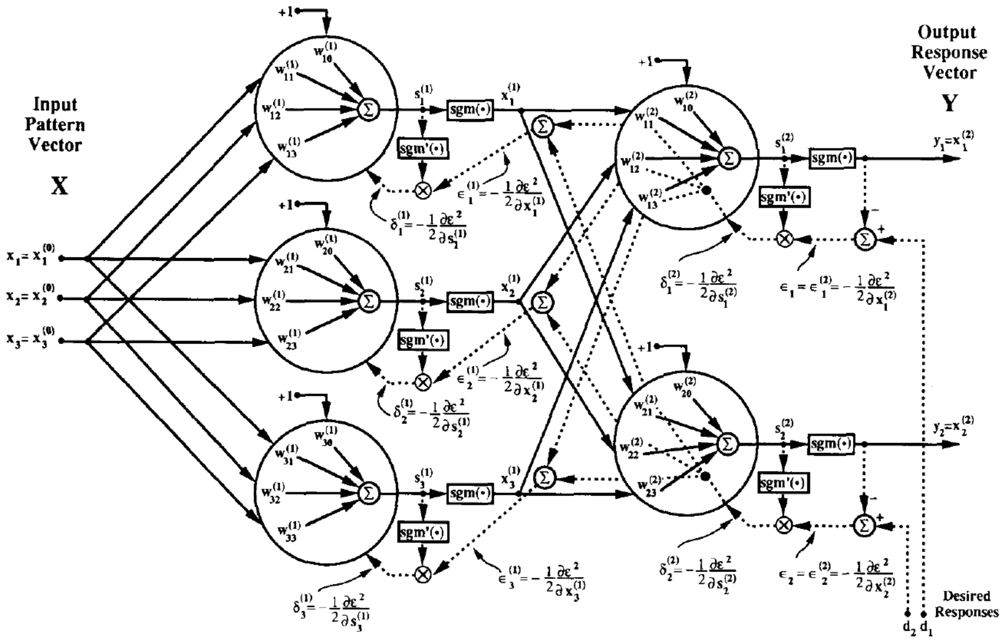


Figure 2.8: An example of backpropagation of error in a two layer neural network. (from Widrow and Lehr (1990))

The backpropagation algorithm is still at the core of Deep Learning (LeCun et al., 2015), and the breakthroughs in image processing, video, text and speech recognition and analysis (Weston et al., 2014). However, it relies on a training set being provided, which can be difficult to obtain in an agent context. We will present *guided policy search* which propose a solution to the generation of a training set.

Genetic algorithms

Genetic algorithms (Holland et al., 1992) are a family of learning techniques inspired by the evolution mechanism of biological organisms, using the gene analogy to describe the parameters of the learning system.

The genetic algorithm works as follows (Russell and Norvig, 2009). An initial population is generated randomly. Each individual is evaluated in an environment related to the task to learn and is given a score by the fitness function (or reward function) of the environment. The individuals with the best scores are selected and their genomes mixed and mutated (crossover and mutation) to generate a new population (generation) to be evaluated. The process is repeated until the end condition is reached (given number of generations is reached or a certain fitness value) and ends by returning the best individual according to fitness score.

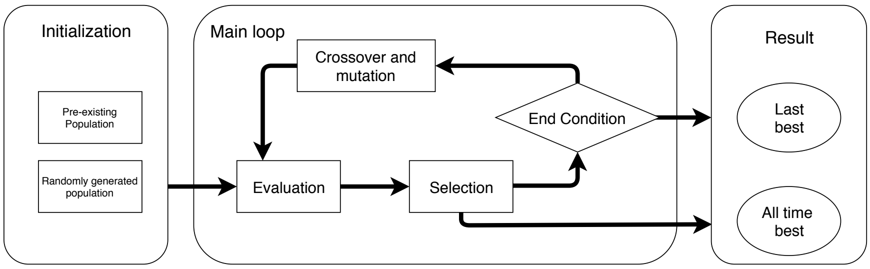


Figure 2.9: Genetic algorithm

Genetic algorithms are suited to evolve populations to fit an environment (to keep the analogy to biology) whose constrains and rewards are themselves changing. When using a genetic algorithm *as* a function optimizer, whose target function does not change, it is preferable to keep the all-time best individual instead of the best of the last generation (De Jong, 1992; Rudolph, 1994).

Genetic algorithms are very flexible and can be employed on a number of problems, provided the parameters to optimize can be represented as genes. In the case of a neural network, the weights of the connections between neurons corresponds to the genome of an individual from the point of view of the genetic algorithm (Jadav and Panchal, 2012). Using genetic algorithms to train neural networks has good exploratory properties compared to the backpropagation method which requires a training set. The genetic algorithm only need a way to measure if the performance of an individual is better or worse than the performance of other individuals. The NEAT algorithm (Stanley and Miikkulainen, 2002) goes a step further than evolving the weights of the network, and also evolves the topology of the network, progressively adding neurons to fit the complexity of the function to represent.

The freedom given in the evaluation of the genomes is what makes the flexibility of the genetic algorithms, it can range from measuring the error to known data points to running a complex simulation to evaluate the behaviour of a system, which is well suited to embodied learning agents. Compared to other reinforcement algorithms where the reward signal modifies the behaviour, with a genetic algorithm the quality of the behaviour is judged at the end of the life cycle of the individual. This allows the individual to get negative rewards if it will lead to a superior positive reward later on, thus circumventing the issue of delayed reward.

Most of the computational cost of the genetic algorithms comes from the evaluation of each genome of a population, which can be costly in the case of a simulation and impractical in real world environments. However, as the evaluation of each agent is completely independent, it can be run in parallel. This allows for the use of High Performance Computing solutions, as a result the real time evaluation of an entire generation is significantly reduced.

2.1.3 Guided policy search

Using neural networks with supervised algorithms, such as backpropagation, in the context of a learning agent raises the problem of constituting a training set.

The Guided Policy Search (GPS) method (Levine et al., 2015b,a; Levine and Abbeel, 2014) exploits the power of neural networks and supervised learning algorithms, the ability to accumulate and generalize from examples, by coupling to them an additional algorithm playing the role of "instructor" for the neural network whose function is to generate example *trajectories*.

The training set used to train the neural network is built from values taken in a valid space around the example *trajectories* generated.

Here the "instructor" algorithm used is a trajectory search method derived from optimal control theories, but there are no constraints on the source of these trajectories. Other mechanisms that generate learning data from demonstrations, other robots or human instructors could be used instead.

The guided policy search algorithm works as follows:

1. Generation of example trajectories (optimal control or demonstrations)
2. Looking for a valid space around the example trajectory
3. Use for supervised neural network training
4. Modify the trajectory space to eliminate those that lead to network failure
5. Loop on 3 until convergence

This method has been tested on several robotics problems, such as the balance of a bipedal walking cycle in case of random thrust, or precise manipulation of objects and tools by a robot composed of articulated arms and a camera, without using an intermediate representation. This last experiment uses a series of neural networks for both image processing and actuator control and gives impressive results, both in terms of learning robustness and movement accuracy.

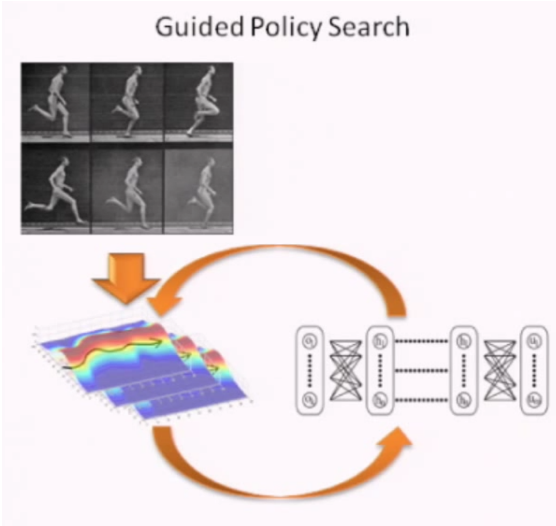


Figure 2.10: GPS operating diagram: on the left is the "instructor" algorithm, generating trajectories, that feeds the neural network on the right (from Levine et al. (2015b,a); Levine and Abbeel (2014))

2.1.4 Curriculum learning

Closely related to developmental agents and lifelong learning, curriculum learning (Bengio et al., 2009) is a machine learning method used to speed-up learning and even solve learning problems that are otherwise impossible. Learning requires a feedback, either from the environment directly or through a teaching entity, but when the learning task and environment are too complex, the feedback signal can end up being too complex to allow learning. For instance, consider the case of accumulating different reward sources for conflicting behaviours, the different values of the rewards interfere with each other and it is not possible to tell which action should be rewarded without the teaching entity's analysis of the context.

Many recent works are aimed at improving the teaching entity (Lopes and Oudeyer, 2012), using for instance the metaphor of motivation (Oudeyer and Kaplan, 2007), which we cover in section 2.2.

Another approach to the problem is to learn each behaviour with its own feedback as a curriculum, and then learn how to combine these well established behaviours. Instead of using a complex teaching entity, a curriculum is handcrafted using simple feedback (*external motivation*). Creating separate tasks greatly simplifies the process of designing learning environments, reduces the cost in supervision during training and also helps in the exploratory aspect of learning, focusing on the additional complexity of the new environment associated with the task.

Curriculum learning has been applied successfully to robotics and video games (agent related) problems (Narvekar et al., 2016). Here the skill is memorized by a single function approximator through the transfer learning of the various source tasks. Curriculum

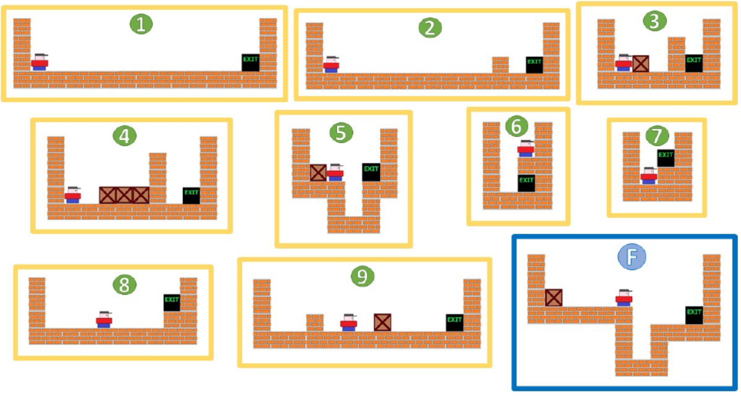


Figure 2.11: The final skill F (highlighted in blue) is learned by transferring all the previous skill learned on sub-tasks of the final task, such as reaching the exit(1), jumping on a block(2), pushing a block(3) ...(from Foglino et al. (2019))

learning has also been applied in a more abstract context to teach neural networks to approximate functions (Gülçehre et al., 2016). In this work the idea is to train the network to match a simplified version of the function and progressively change this target function to match the actual function we want to approximate. While this is an interesting progressive learning method, we could argue about the use of the term curriculum. This learning method just adds more complexity to the same learning task instead of being a collection of complementary learning tasks.

2.1.5 Layered learning

Layered learning is another progressive learning method close to curriculum learning, that takes into account the structures used to represent skills. The idea is to build a hierarchy whose elements are trained separately by a different task of the curriculum, and given responsibility for different functions of a complex task. Principle 4 of Stone and Veloso (2000) states that the key defining characteristic of layered learning is that each layer directly affects the learning at the next layer, this includes providing features used by the next layer. Figure 2.12 shows the low level behaviours of a soccer playing robot: *Pass Evaluate* is a single layer perceptron trained on a low level task, the output of this trained layer is to be used as input for a new layer which is trained on a higher level task, for instance, progressing towards the enemy goal. This method could be viewed as training a multi-layered perceptron one layer at a time.

This is not unlike the structure of Convolutional Neural Networks (CNN, Krizhevsky et al. (2012)) where low level kernels are in charge of simple shape recognition and fed to the next convolutional layer. In Devin et al. (2017) a CNN is trained on a sensorimotor task in such a way that it can be cut in the middle, the input side is referred to as the "task" module and the output side as the "robot" module (body). This method makes

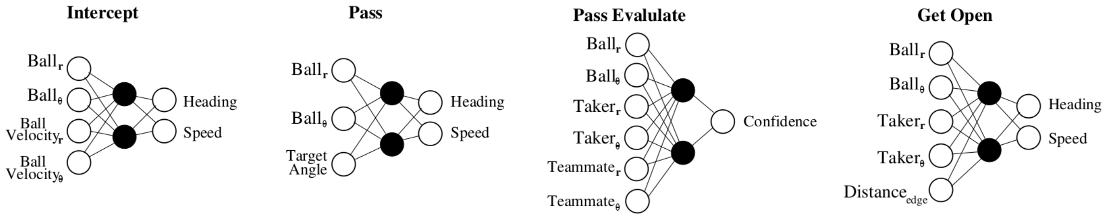


Figure 2.12: Low level behaviours of a soccer playing robot. The output of *Pass Evaluate* is used as input for higher level decision (from Whiteson et al. (2003)).

different modules interchangeable, allowing one robot module to perform different tasks or one task module to use different robot bodies.

Such methods operate by successively refining the input for a higher level decision, forming an "in-line" structure. Although each skill is identifiable, their interaction is not based on arbitration of concurrent behaviours.

2.1.6 Learning for developmental agents

All the learning methods presented here can be fitted to a developmental mechanism, however some will be better suited to the nature of the agent and the scope of its development.

For instance, if we are confident that we can provide an exhaustive list of the motor primitives of an agent, and that there will never be additional primitives for its actuators, then it would be acceptable to build skills upon these motor primitives. We can see however that a signal oriented approach offers the most possibilities for the acquisition of low level motor skills, the skill itself learning a motor primitive by directly controlling each actuator.

The same principle holds for the methods themselves, guided policy search shows that it is possible to adapt the backpropagation algorithm, which is not ideally suited to the training of situated agents, but this comes with its own constraints (finding a demonstrator for an hexapod walk cycle might not be trivial). Even if genetic algorithms have a high training cost, they have good exploratory properties and give the agent the most freedom in finding a solution.

When it comes to learning strategies, curriculum learning seems like the best choice for developmental agent. The initial training strategy consisting in learning simple skills before moving on to more complex skills will extend naturally to the autonomous development of an agent in the real world. Provided that an autonomous system is capable of analysing a new problem to extract a reward information, the new challenges an agent will face will be considered as new lessons, which it will have to learn based on the previous skills it acquired in training.

2.2 Motivational systems guiding agent development

All the learning methods presented rely on an evaluation of the quality of the behaviour. Reinforcement learning requires a reward, genetic algorithms use a fitness function, even backpropagation uses a training set which can be considered as the known behaviour of highest value. It is the role of the motivational system to evaluate behaviours and provide a reward to the learning system.

Motivation has been studied in psychology on many levels, from the training of animals to the ability of humans for self-determination classical conditioning and operant conditioning (Skinner, 1965) deal with the association of stimulus and behaviour through the use of reward and punishment. Drive reduction theory (Hull, 1943) evaluate behaviour based on the satisfaction of internal physiological or psychological states. Reduction of cognitive dissonance (Festinger, 1962) and optimal incongruity drive exploratory behaviour and reward the reduction of the incompatibility between the internal model and observations of the environment (respectively: from the discomfort cause by the unexpected, and from the curiosity in the unexpected).

2.2.1 External, internal and intrinsic motivation

The place of the motivational system in respect to the agent, external or internal, seems to have an impact on its capability for autonomous learning. In the case of external motivation the motivational system, for instance a trainer, is part of the environment. This trainer uses his own observations on the environment, the interactions of the agent with the environment, to compute a reward for the learning process. With internal motivation, the motivational system and learning process are part of the agent, which means the motivational system observes and computes the reward from the point of view of the agent, and is limited to the perceptive capabilities of the agent. Oudeyer and Kaplan (2007) argues that the distinction is not so great, and that placing the motivation internally is a matter of adding built in capabilities for self monitoring. From a design perspective Dorigo and Colombetti (1994) remarks that external motivation (a Reinforcement Program independent of the agent) being machine independent is portable from agent to agent. We extrapolate that a properly implemented internal motivational system, dependent on the agent configuration, should be portable from environment to environment, that is, able to compute rewards from a variety of perceptions.

Whether the reward is calculated internally or externally, the origin of the motivation is an immediate gain from the environment. This form of extrinsic motivation is efficient but lack exploratory properties, and does not give any solutions to reach unknown desirable states far removed from immediate gain. An agent relying on extrinsic motivation will require a carefully crafted training course, or curriculum, with intermediate rewards to guide it to non-obvious (or by nature incremental) solutions.

Intrinsic motivational systems are an attempt to provide developing agents with a generic, environment independent, solution to the exploration and discovery

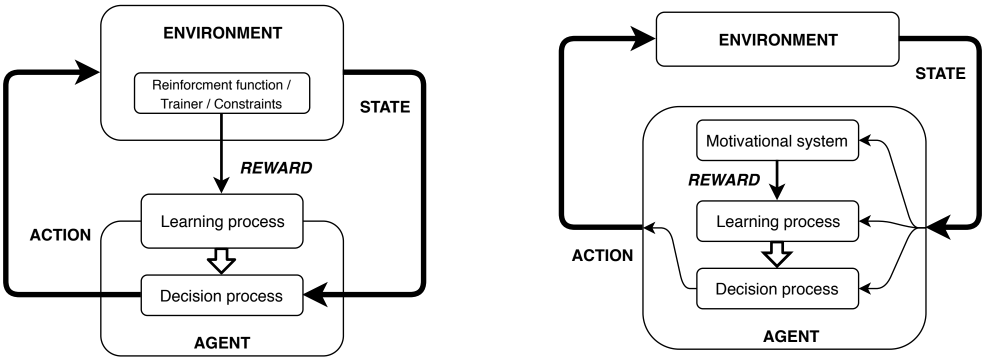


Figure 2.13: On the left: externally motivated behaviour, on the right: internally motivated behaviour (inspired from Oudeyer and Kaplan (2007) and (Barto et al., 2004)).

in autonomous learning. In addition to computing normal rewards based on concrete gain from the environment, the intrinsic motivational system adds an “imaginary gain” to the decision process in order to drive the agent towards exploration. Intuitively this can be understood as artificial curiosity, boredom leading to a break from a repetitive routine or dissatisfaction with one’s failure in predicting a future event.

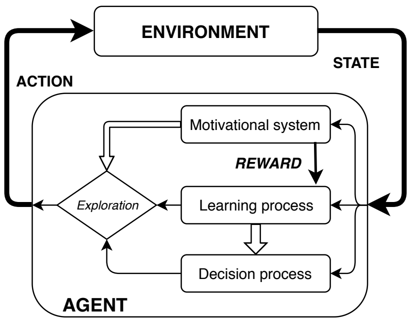


Figure 2.14: Internal intrinsic motivation (inspired from Oudeyer and Kaplan (2007)).

2.2.2 Variance and novelty intrinsic rewards

TEXPLORE-VANIR (Targeted Exploration with Variance And Novelty Intrinsic Rewards) (Hester and Stone, 2012, 2017) is an implementation of an intrinsic motivational system on TEXPLORE (Hester and Stone, 2010), a model-based learning algorithm. The model used in TEXPLORE is a random forest, a collection of decision trees trained by different subsets of the agent’s total experiences. The predicted outcome of the random forest model is the average of the predictions of each tree.

TEXPLORE-VANIR make use of the variance in prediction between the trees of the random forest as one of its intrinsic reward (equation 2.3).

The decision process takes into account this value to lead the agent where this difference in prediction is great, starting an active learning process that will solve this

$$D(s,a) = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{m} D_{KL}(P_j(x_i|s,a)||P_k(x_i|_{s,a})) \tag{2.3}$$

Variance calculation, with $D(s,a)$ the total difference in prediction, j and k each pairs of decision trees, and i each feature.

difference in prediction through experience. This variance based method is comparable to the reduction of cognitive dissonance discussed previously (Festinger, 1962).

This method is balanced by another intrinsic reward based on novelty, which stimulates exploration early on, and help build the models, and limits the influence of variability later on, in cases where no progress in convergence of the models is made.

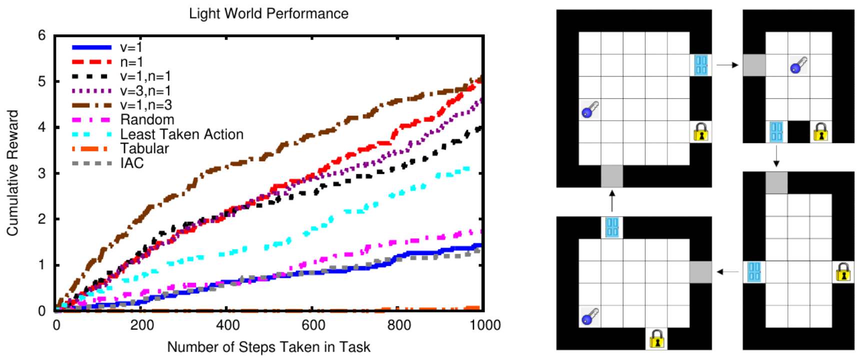


Figure 2.15: On the right: lightworld, the agent must pick up the key, open the lock and reach the door. On the left: result show VANIR above all other method when using the novelty motivation (Best score is obtained with a novelty coefficient of 3 and a variance coefficient of 1) (Hester and Stone, 2017)).

2.2.3 Motivation for developmental agents

In the interest of autonomy for developmental agents, the question of motivational systems is of great importance. As mentioned before, an intrinsic motivational system is the missing piece to the automatic generation of new "lessons" for an autonomous developmental mechanism based on a curriculum approach.

However, it is still interesting to develop approaches based on extrinsic motivation, internal or even external, where agent learning is guided by a curriculum carefully planned by the designer. Such methods are referred to as *shaping* (Dorigo and Colombetti, 1994, 1998), while this strongly guided approach lack in high level autonomy, it still retains the advantages of development, the ease of extension of the agent's abilities, and can be considered a high level agent "programming" method.

2.3 Structures supporting agent development

Designing a developmental system will have us consider the interaction and the integration of the various subcomponents of the model with each other (Lungarella et al., 2003). These subcomponents can be learning subsystems, motivational systems, information and memory sources, or simply the new skills to be incorporated with the existing ones. The different aspects of developmental robotics find their unification in structure, and structuration techniques. To fulfil the developmental requirements, the structures must achieve the coexistence in a single system of a wide variety of representation of skills (classifiers, neural networks, pre-programmed skills...), acquired by diverse means (intrinsic motivation, supervised training, social learning...), using a variety of sensors and in potential competition for the control of the same actuators.

An implication of building a complex behaviour based on simpler, and non-alterable, skills is the potential competition for control of the actuators. The *turn right* and *turn left* skills cannot both, at the same time, have control of the agent's motion. Hence, the complex behaviour must solve a problem of coordination of skills, either though combination or sequence.

2.3.1 Sequential composition of skills

The most straight forward method of skill combination is their sequential use. Using skills in this manner fits the common deliberative paradigm Sense-Plan-Act as well as reactive systems.

The subsumption architecture (Brooks, 1986) is a reactive robot control architecture that combine skills by priority, from long term to short term solutions. Skills are arranged in a hierarchy, the lower levels perform the most basic functions, such as avoiding an immediate collision, the higher level skill perform functions of an increasing complexity, such as exploration. The higher level skill can either perform its own function or let its subordinate perform its function. For instance, the skill *explore* starts heading to a distant place, if the agent meets an obstacle, *explore* lets its subordinate skill *avoid* take control. As soon as the obstacle is out of the way, *explore* takes the control back from *avoid* and continue on its way to its objective. The subsumption architecture is a method for designing robot control architecture, it isn't specifically designed for learning.

In *skill chaining* (Konidaris and Barto, 2009) a high level skill learns, through reinforcement learning, how to order subskills (themselves learnt through reinforcement learning). The main skill runs a subskill that approach the goal state up to the point it recognizes a particular state where further use of the same subskill won't allow progress. This state serves as a trigger for a new subskill that will learn how to get even closer to the goal, the idea being that the way to get closer to the goal is completely different from the previous subskill and therefore incompatible.

This method is used on a navigation problem where the main skill is in charge of navigating an environment and the subskills are in charge of navigation from point to point.

The main skill and its subskills are represented by linear approximations (using *Fourier basis learner* (Konidaris, 2008)), which taken together represent a complex function (defined by parts). The main skill learning method is SARSA (Sutton and Barto, 2011), an optimization of the Q-learning algorithm presented in the previous section, subskills are learned by Q-learning (Watkins, 1989).

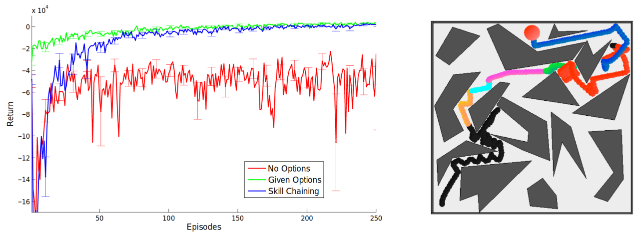


Figure 2.16: Experimental results of skill chaining, on the left the results: No options uses a single skill, Given option learns the main skill based on given subskills, Skill chaining learns the main skill and subskills. On the right an example of the trajectories, each colour represent a subskill (Konidaris and Barto, 2009).

Figure 2.16 shows that *skill chaining* succeeds in solving a navigation problem where attempts with a single skill do not.

2.3.2 Learning hierarchies of skills

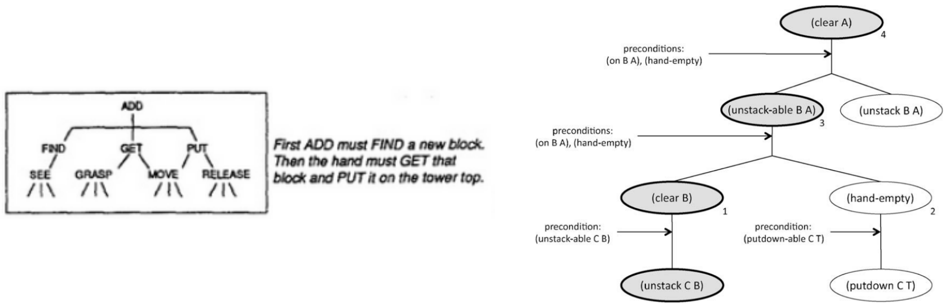


Figure 2.17: Decomposition into subskills (left: Minsky (1988), right (Langley and Choi, 2006))

One of the most intuitive approach to a cognitive architecture was introduced in *the society of mind* (Minsky, 1988) with the example of a child building a block tower. On top of simple and identifiable skills are built complex skills which coordinate the simpler skills to achieve their goals. The complex skills are themselves identifiable by their goals, and are available to higher level skills for coordination.

Many cognitive architectures incorporate such hierarchical structures, such as the ICARUS architecture (Choi and Langley, 2018). ICARUS includes teleoreactive (Nilsson, 1993; Morales et al., 2014) skill execution, a goal oriented (teleo) method of execution able to form a hierarchy of subgoals whose satisfiability are evaluated continuously (reactive). Prior to execution, low level percepts are evaluated in a bottom-up manner by a belief hierarchy. Several percepts combine to make a belief true, and several beliefs can combine to make a higher level belief true. These beliefs serve in evaluating the satisfiability of goals.

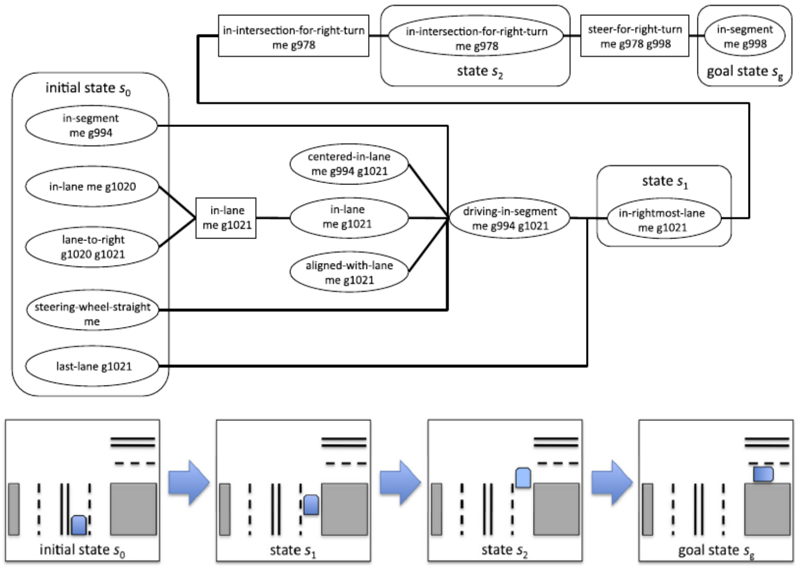


Figure 2.18: ICARUS solving urban driving problem (from Choi and Langley (2018); Langley et al. (2009))

The execution starts from a main goal and selects its most appropriate subgoal, and in turn selects the most appropriate subgoal of the subgoal. This process continues until a primitive action is reached, this action is executed until the parent goal is satisfied (or a change of belief renders it unsatisfiable). The parent goal in turn selects its other subgoals until its parent goal is satisfied. In effect the goal hierarchy is a directed acyclic graph evaluated in a depth first order. Actions, and goals, are carried out until a change of state (beliefs) can be observed, either success or failure, making this process reactive. However, contrary to a simple reactive hierarchy the skill selection process doesn't start over from the main goal but from the currently active subgoal. In case of failure the subgoals evaluate the alternate actions it can take before returning control to its parent goal. This design gives ICARUS a balanced approach between commitment to high level plans and purely reactive behaviour.

Robot shaping (Dorigo and Colombetti, 1994, 1998) is a set of techniques focusing on the learning and developmental aspect of reactive skill hierarchies. In robot shaping, behaviours are learned as a curriculum and represented as individual skills, these skills are then combined to achieve higher level behaviours. The articles cover both the architectural and didactic aspect: Multiple architectures are discussed, from monolithic to multi-level hierarchies, learning methods and reward methods tailored towards artificial agents are proposed.

In robot shaping, the curriculum starts by the low level skills controlling the actuators (for instance: chase, feed and escape). Skills are learned by a Classifier System (CS) that outputs a binary string comprised of the motor command and a “coordination message” (in the case presented: a single bit telling if the classifier proposed an action). Then a coordination skill learns the final behaviour by coordinating low level skills. The coordination skill is also a classifier system that takes as input the coordination message from the low level skills and use it to control a composition rule (for instance: a switch). In this case the result is a flat architecture, all the skill are one level under the main coordinator (Fig. 2.19).

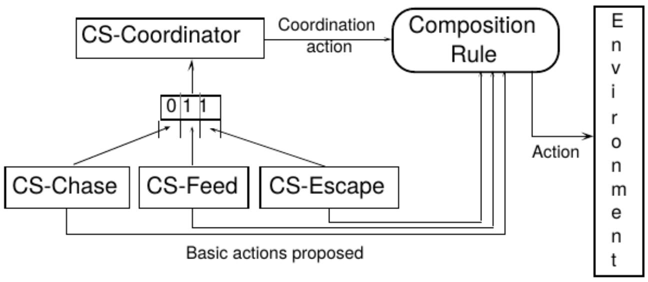


Figure 2.19: The coordinator classifier system using the messages from each low level skills to control the composition of the different motor commands. This example is a flat architecture with only one level (From Dorigo and Colombetti (1994)).

To build a hierarchical architecture, lower level skill are coordinated by separate coordination skills. In turn, the coordination skills are themselves coordinated thus forming the hierarchy (Fig. 2.20).

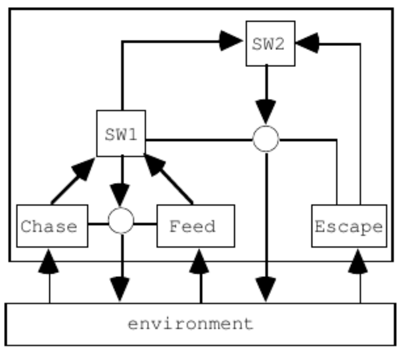


Figure 2.20: *An example of three-level switch architecture for the Chase/Feed/Escape behaviour. Besides the three basic behaviours can be seen the two switches, SW1 and SW2.* From Dorigo and Colombetti (1994)

As we explained, the lower level skills are the only ones with access to sensor data and send requests for action to the higher level skills (coordinators). Based solely on these requests, the higher level skill chooses which and how subskills should be coordinated. This one of the drawback of this architecture: dealing with how to solve a problem involves synthesizing, distorting and discarding a part of the information or signal. The low level skill *escape* (of the *chase/feed/escape* hierarchy presented in Dorigo and Colombetti (1994)) which solves the problem of escaping a predator does not need to know if the predator in an immediate threat or keep track of other priorities to perform its task. Information as to *why*, *when* and *if* is discarded, leading to the same response when subtle details in the context would call for an entirely different approach to succeed. For instance, the information about the current distance of the predator and the importance of hunger, which are not used by *escape*, could influence the choice between continuing to feed a while longer or fleeing immediately. This was improved in later works (Larsen and Hansen, 2005) by giving the higher level skills direct access to sensor data, including data from sensors not involved with the subskills.

One of the limitations of robot shaping hierarchies is its use of binary strings as outputs for its skills. While it has a low resource cost, it only allows sequential and exclusive skill use.

2.3.3 Simultaneous composition of skills

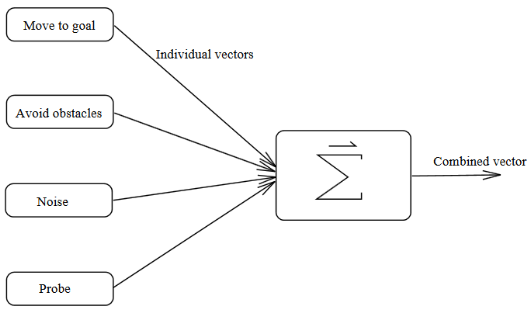


Figure 2.21: Vector summation in AuRA (from MacKenzie et al. (1997))

Another approach to skill coordination is their simultaneous use though a merging process, such as weighted sum of their output vectors. An early example of this method of coordination between concurrent behaviours is the works on *boids* (Reynolds, 1987). Boids are virtual birds designed to experiment with flocking and heard behaviour. In order to exhibit a flocking behaviour, the boids must satisfy three constraints: cohesion (staying close to the group), separation (keeping a minimum distance with other individuals) and alignment (heading in the same direction as the group). Three independent motor skills are in charge of each of these constraints, and the flocking skill becomes, as Reynolds puts it, a problem of *arbitrating independent behaviours.*

*Each behaviour says: "if **I** were in charge, **I** would accelerate in that direction." [...] It is up to the navigation module of the Boid brain to collect all relevant acceleration requests and then determine a single behaviourally desired acceleration. It must combine, prioritize, and arbitrate between potentially conflicting urges.*

Reynolds (1987)

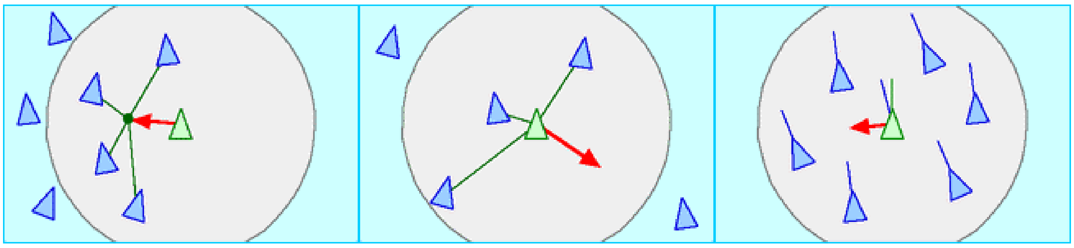


Figure 2.22: the cohesion, separation and alignment behaviours of the *boids* combine into a single complex flocking behaviour (Reynolds, 1987)

This mechanism of output vector composition is used in a number of control architectures, usually for low level motor control. In Sat-Alt (Simonin and Ferber, 2000)(satisfaction-altruism), a model for multi-agent cooperation, once the deliberative process has determined the bearing to the goal, the motor command to reach the goal is combined with other low level behaviour such as obstacle avoidance (Fig. 2.23).

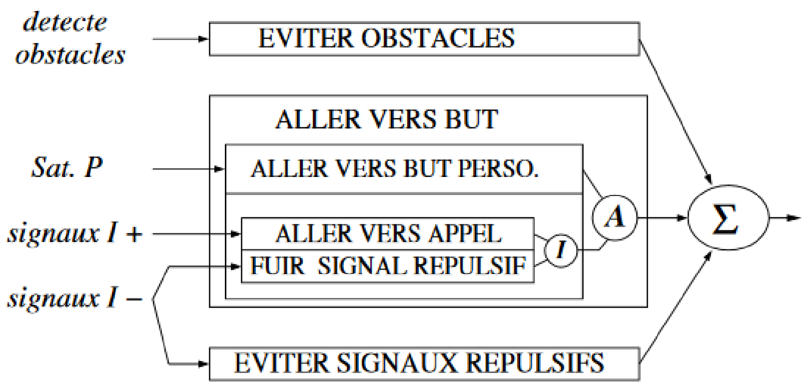


Figure 2.23: Satisfaction-altruism model. The middle box represent the deliberative process to set the goal, which is then combined with avoiding obstacles and avoiding repulsive signals (Simonin and Ferber, 2000)

AuRA (Arkin and Balch, 1997) (Autonomous Robot Architecture) is a fairly complex deliberative/reactive hybrid architecture for robot control composed of many modules such as action planning, environment mapping or user interface. This architecture is build upon a reactive motor control module which uses vector summation to combine multiple *motor schemas* (simple behaviours) into complex behaviours (*schemas*).

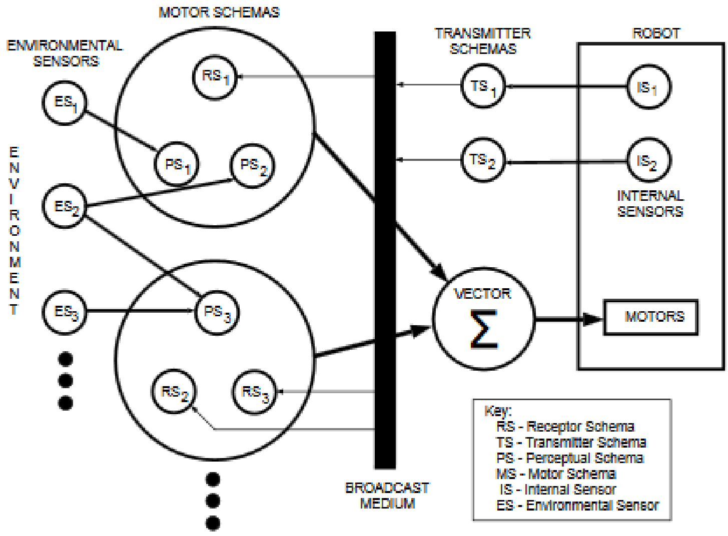


Figure 2.24: Diagram of the reactive component of AuRA (Arkin and Balch, 1997)

The *motor schemas* can either be given or learned independently, the modular nature of the architecture allow the combination of different methods. Representing *motor schemas* as independent modules facilitates adding and replacing *motor schemas* in an existing system.

The high level *schemas* resulting from the vector summation are available to the action planning modules to be used sequentially.

A limitation of the reactive module of AuRA is that the combination operation is a simple sum, leaving the “Arbitration” part to the low level *motor schemas* themselves. For instance, *Move-to-goal* gives a constant motor command of medium strength directing the robot towards the goal and *Avoid-static-obstacle* gives a variable motor command to move away from an obstacle depending on the proximity of the obstacle. By a simple sum, when an obstacle is too close the strength of the *Avoid-static-obstacle* command will have more impact on the motion than *Move-to-goal*, as the robot moves away from the obstacle, the *Avoid-static-obstacle* strength will weaken and *Move-to-goal* will progressively have more influence over the motion of the agent. This is an important distinction to make from the *boids* (Reynolds, 1987), where the coordinator is in charge of the “arbitration” and is able to suppress a low level behaviour even if it sends a strong motor command.

Vector combination was also used in complex locomotion problems (Heess et al., 2016) to a contemporary standard of complexity (up to a 54-dimensional humanoid). This approach covers a single agent sensory-motor development using low level behaviour

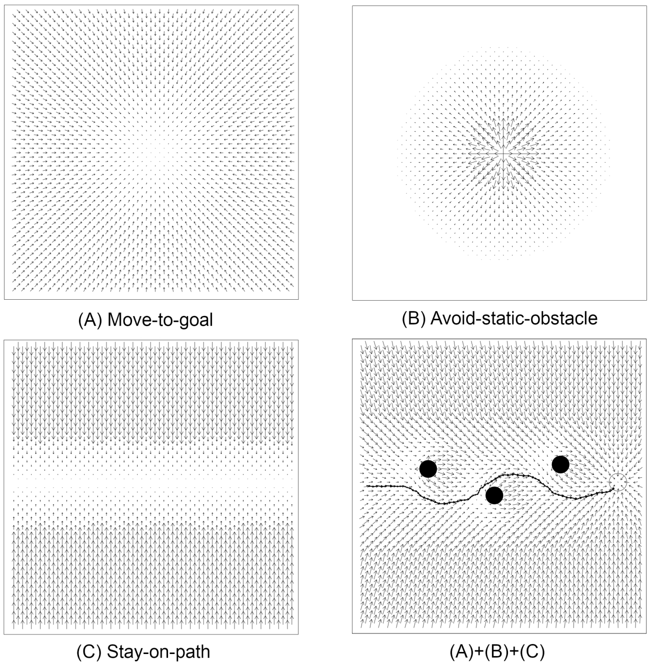


Figure 2.25: A reactive path generated by combining 3 motor schemas (adapted from Arkin and Balch (1997))

elements, qualified as "spinal", to learn sensory-motor primitives. These behaviours are then coordinated by high level "cortical" elements which drive behaviour by modulating the inputs to the spinal network. The purpose of the cortical-spinal analogy is to emphasize the time scale difference, allocating more resources for fast sensor acquisition at the "spinal" level, but the cortical elements are a good contemporary example of Reynold's behaviour arbitration.

Vector composition is integrated in a multi level hierarchical architecture in works on open-ended evolution of virtual creatures (Lessin et al., 2013, 2015). As the name implies, this approach lets low level signal oriented components evolve to fit a task, the resulting organisation solidifies into a skill which can then be used for combination. This contrasts with RSH declaration of skills whose controller are then trained to perform tasks, coming from a more supervised "shaping" philosophy. Both methods are faced with the problem of coordinating subskills. In the evolving virtual creatures, commands are transmitted as signal and combined using various operators (including the "pandemonium" which is used where mutual exclusion is required). When a particular combination of inputs, outputs and combination operations is successfully evolved to perform a task, it is encapsulated as a skill. The encapsulated skills are then available as output for new combinations.

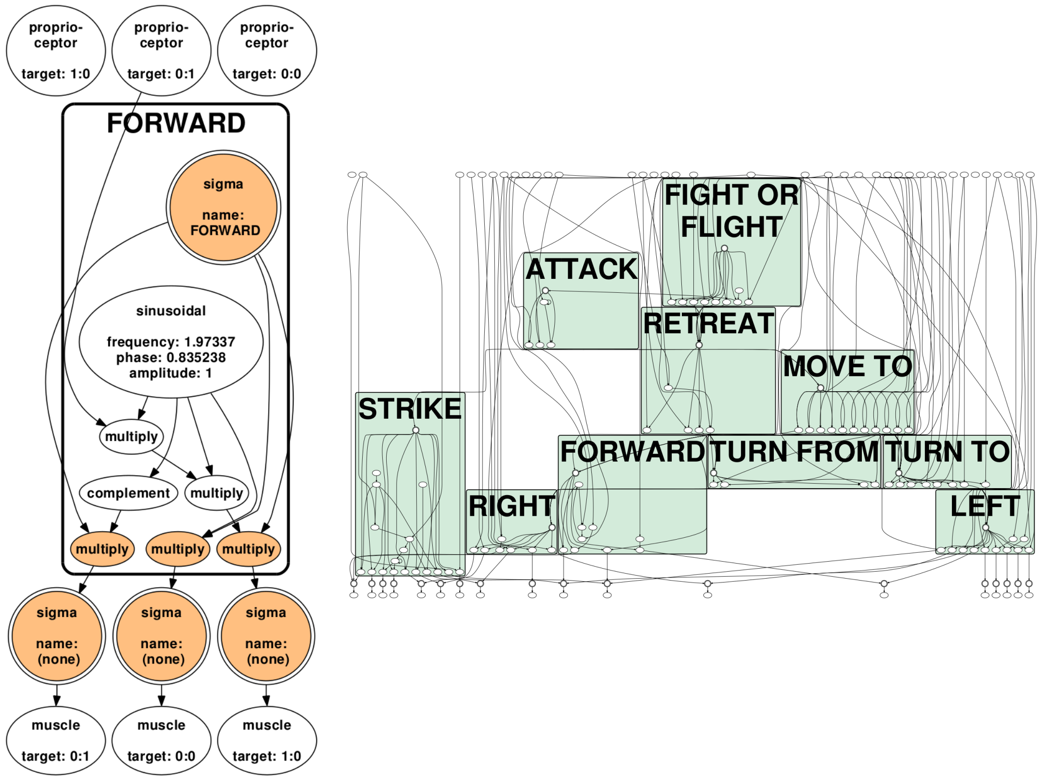


Figure 2.26: Open-Ended evolution of virtual creatures. Left: inputs, outputs and combination operations are encapsulated in a skill. Right: several encapsulated skills organize into a hierarchy (from Lessin et al. (2013)).

2.3.4 Structures for developmental agents

We have already explained the importance of proper structuration of skills to support learning, to favour reuse of previously acquired behaviour and speed up the learning process. This structuration becomes even more important in the context of open ended development, were learning as a lifelong cumulative process takes all its meaning.

Approaches using skill hierarchies, decomposing behaviours into sub behaviours and assigning an identifiable skill in charge of a well defined sub behaviour, seem the most suited to developmental agents. The clearly defined area of responsibility of a skill makes it available for multiple combinations in separate sub hierarchies. For instance, *hurrying to work* and *playing football*, two very different high level behaviours, can both use a subskill whose responsibility is the quick navigation towards a target.

For the same reasons that a signal oriented approach is preferable to learn low level motor control, a hierarchy using simultaneous skill composition is preferable to sequential composition. Many intermediate behaviours can be reached by combining two skills, and thus these intermediate behaviours do not need skills to represent them. Methods using composition of skills conforming to Reynold's behaviour arbitration have the advantage of being capable of both simultaneous and sequential (exclusive) composition of skill. Exclusive execution is achieved by attributing to one skill the maximum possible influence and no influence to all the other skills. This makes simultaneous composition more generic, although in practice it adds to the computing cost (unnecessary weight computations using real numbers).

Method	Simultaneous composition	Weight arbitration	Multi-level hierarchy	Learning method independence	Substructure independence
Robot shaping (Dorigo and Colombetti, 1994)	✗	✗	✓	✓	✓
Composite robot behaviour (Larsen and Hansen, 2005)	✗	✗	✓	✓	
ICARUS (Choi and Langley, 2018)	✗	✗	✓	✓	✓
AuRA (MacKenzie et al., 1997)	✓	✗	✗	✗	✗
Modulated locomotor controllers (Heess et al., 2016)	✓	✓	✗		
Virtual creatures (Lessin et al., 2013)	✓	✓	✓	✗	✗
Our contribution: MIND	✓	✓	✓	✓	✓

Table 2.1: A comparison between the previously discussed structures along key points of open-ended agent development which our contribution, MIND, will address.

2.4 Memory systems and internal representations in cognitive architectures

The development of an agent, as an individual, involves persistence of knowledge outside the environment, in the agent itself. We have seen so far the acquisition of skills, the importance of their structuration and persistence, the memorization of 'savoir-faire' knowledge (or procedural knowledge). Various motivational systems also depend on internal representation, such as those based upon *drive reduction theory* (Hull, 1943), as well as representation of an agent's internal states, physiological or psychological.

2.4.1 Neuro-inspired low-level memory

A step beyond reactive agent involves memory as understood in common language: recording past states of the environment. In robot shaping (Dorigo and Colombetti, 1994), the authors already included a memory of the past state of the agent's sensors (following remarks from (Whitehead and Lin, 1993)). It is noted that this kind of memory *need not to be regarded as a "representation" of anything*. This aspect of short term memory is close to the idea of *iconic memory* or *fragile memory* in the human brain. Both are short-lived high-capacity memories used in storing a much larger perceptual information than can be immediately processed. Effectively, a raw duplicate of the previous perceived state. Recent studies (Vandenbroucke et al., 2014) indicate *fragile memory* has some level of processing, although not on the same level as *working memory*. This could be likened to the memory elements of a recurrent neural network (See subsection 2.1.2 Fig. 2.7) which stores the state of intermediate neurons, or *reservoir computing* methods such as *echo states networks* or *liquid state machines*. In reservoir computing instantaneous input are fed to the reservoir network which accumulates and enriches the state space. Readout are done by another network feeding from the reservoir and trained by traditional methods such as backpropagation.

This is contrasted by *working memory*, which is a (relatively) long-lived low capacity memory used to retain processed perceptions, that can be consciously addressed. *Working memory* can be understood as storing high level elements for short term planning, for instance the spatial coordinates of a labelled entity in the context of a navigation problem. An overview of these distinct short term memories in the human brain and a examination of their respective properties is given in Vandenbroucke et al. (2014).

2.4.2 Memory in cognitive architectures

Cognitive architectures include long term memory systems, for procedural knowledge as stated before, but also for declarative knowledge, association and past experiences. Current elements of working memory or perception are mapped to the appropriate long term memory structure in order to gain higher level knowledge or prediction.

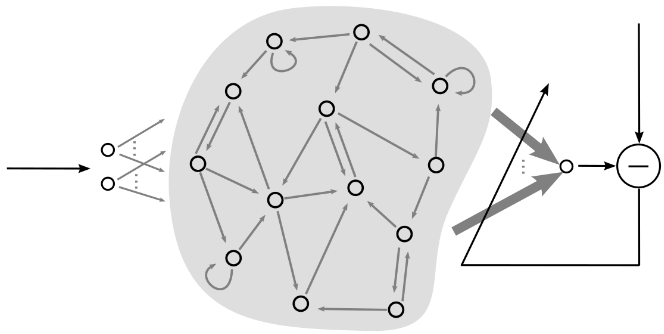


Figure 2.27: Reservoir computing: the instantaneous input on the left is fed to the reservoir network (in grey). On the right, the readout is done by another network (from Lukoševičius and Jaeger (2009)).

Briefly introduced in 2.3.2, the belief hierarchy of the ICARUS architecture (Choi and Langley, 2018) is a long term memory of percept associations. In the example given in Fig. 2.28, from the position of 3 lines on a road the agent uses primitive beliefs (right-of) determine their order (the line 2 is between line 1 and line 3), and from there the existence of a lane between line 1 and 2. Comparing his position to the line 1 and 2 with primitive beliefs, the agent determines he is in the lane 1-2. The semantic structure associating beliefs is the long term memory element and reflects a knowledge about the world that should hold true (if there are no lines between two lines on a road, then the two lines form the boundaries of a lane).

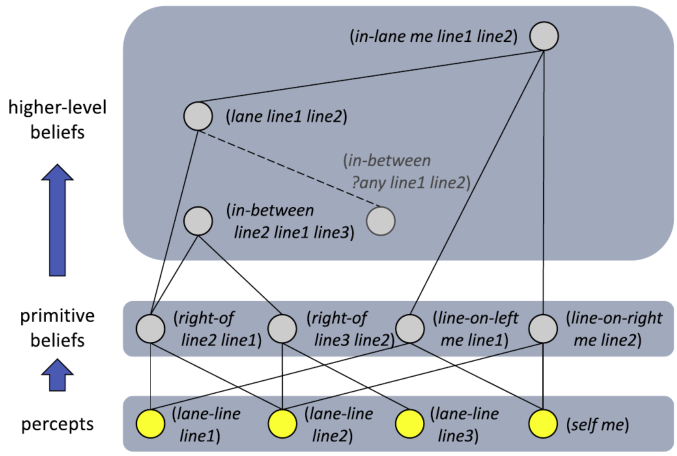


Figure 2.28: An example of the ICARUS belief system. The agent determines he is in the lane 1-2 from the perception of the relative position of 3 lines and his own position (from Choi and Langley (2018)).

In addition to long term semantic memory, the extended SOAR architecture (Laird,

2008) include episodic long term memory. Episodic memory retains sequences of states, perception and working memory elements, that were experienced by the agent. When the agent experiences a succession of states that matches a partial sequence in the Episodic memory, the rest of the sequence can be considered in order to make a prediction. Episodic memory was experimented with on the TankSoar environment (Nuxoll and Laird), a 2D maze environment where an agent must fire at enemies, dodge enemy projectiles and collect energy charges (recharge battery). The energy search task gives a clear example or the use of episodic memory. During the course of the game the agent will perceive an energy charge which he does not need at the present moment. However, the steps or successive states leading to the perception of the energy charge are stored in the episodic memory. This sequence might contain: 1 step down a corridor/1 step down a corridor/at crossroad turned left ... further steps leading to the energy charge. When the agent needs to find the energy charge he will navigate the maze until he finds a “familiar” set of circumstance, for instance: I took two steps down a corridor, I find myself at a crossroad that has a possible left turn. The agent matches this sequence of events he just experienced with the episode in memory we described, and by following the actions taken in the episode he reaches the energy charge. Results on this particular task indicates the use of episodic memory is ten times more efficient than a random search.

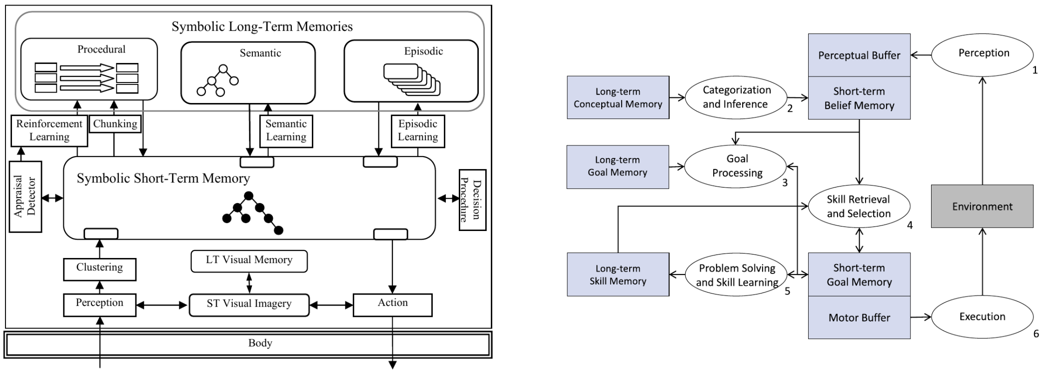


Figure 2.29: On the left: the extended SOAR architecture, showing a flat mapping between long term and short term memory (from Laird (2008)). On the right: the ICARUS architecture, mapping long term to short term memory of different components at different stages of the deliberative process (from Choi and Langley (2018))

Although the example given is a navigation problem, the encoding of the episodic memory is task-independent (as far as there remains a temporal relationship).

Making the representation of knowledge (encoding) strongly dependent on the task and type of knowledge is a common practice, such as in the case of environment mapping. The agent posses dedicated sensors to record an accurate spatial map of the environment, LiDAR time of flight information are used to generate clouds of 3D points, in turn these 3D points are then interpreted as simple volumes, bounding boxes and planes which are stored in memory. This representation of the world is not only convenient for humans to understand, but the agent can be given algorithms specific to this representation, for instance path planning algorithms. A dedicated spatial representation of the world, or model, also allows for simulations and projections that can guide the decision process.

2.4.3 Internal representation and symbol-grounding

Whether or not it is justified to provide embodied agent with a dedicated spatial memory, developmental robotics will require a general purpose memory system. Specifically, the mechanism of encoding high dimensional state or intermediate elements of the working memory into a highly compressed representation is a potential key element to emergent symbol grounding. The emergence of high level symbols opens the way to general purpose artificial intelligence, and high level reasoning.

In the formation of such general purpose long term memory elements, social interaction will certainly play an important role. The individual experience and synthetization process of the experience will lead to the generation of many symbols. Through exchange of such symbols in a community bound to encounter a similar experience, their validity can be cross-checked, the symbols can be refined or rejected, merged or simplified, their expression standardized. This process constitutes a vastly distributed experimental machine where the concept can go through much more testing and refinement, with a much wider range of possible condition than would be possible in the lifetime of a single agent. Symbols surviving this process are adopted by the majority as a culture and will form the basis of communication.

2.4.4 Internal states for developmental agents

Although it is well known that intelligent behaviour can emerge without internal representations (Brooks, 1986), in the interest of pushing development through increasingly complex behaviour, developmental agents should be provided with some form of internal representation.

Should the skills of the developmental agent use recurrent neural networks, some form of low level memory system will already be included, but some intermediate level memory system should also be included at the level of the control structure of the agent. Such memory system would allow the agent to commit to a task and offer alternatives in its structuration by, for instance, generalizing some behaviour around the representation of a concept.

A developmental agent could benefit of the use of high level representations, however the main drawback of these representation systems is their dependency on the type of knowledge they represent. Seeing all the different types of memory systems used by SOAR, one can ask the same question as with control through motor primitives: are we confident we listed all of them?

2.5 Social interaction between agents

In the introduction to the First International Workshop on Epigenetic Robotics (Zlatev and Balkenius, 2001), a precursor to the general field of developmental robotics, the authors stresses the importance of social interactions among interactions leading to cognitive development. Personal development through guidance and imitation, but also development of a social organization though individual roles. In order to find its place into the real world, a developmental agent will require the ability for social interaction, both with its kind and with heterogeneous populations of agents, including biological ones.

In parallel to the field of developmental robotics, Multi Agent Systems (MAS) studies the interactions of situated agents, both with the physical and social environment. The study of multi agent systems has a two-fold approach, similar to developmental robotics:

- Understanding complex dynamical systems through simulation of physical or social interaction. For sociology (Sawyer, 2003), modelling ecosystems (Bousquet and Le Page, 2004), but also in sciences such as medicine where agent-based models are used on multiple levels to gain insights into the function and actions of biological system (An et al., 2009).
- Create new models and tools to fit and take advantage of the complexity and large number of interactions. Applications are found in many fields such as smart power grid management (Merabet et al., 2014), production line organization (Leitão et al., 2012), internet of things (Calvaresi et al., 2017), swarm robotics and fleets of autonomous vehicles (Carlési, 2013).

2.5.1 Multi-Agent Systems

Multi-Agent System (MAS) are system made of artificial and/or natural autonomous entities called agents, evolving in an environment and interacting to produce behaviours that are collective. In (Ferber, 1995), Ferber proposes following definition of an agent:

> ***Agent:*** *A real or virtual entity, operating in an environment, capable of perceiving and acting upon it, that can communicate with other agents, that exhibits a behaviour which can be seen as a consequence of his knowledge, skills and experience, interactions with other agents and its goals.*
>
> Ferber (1995)

The study of multi-agent systems has shown the benefits of an agent-oriented vision and social interactions in a system. Work on the modelling of insect societies (Deneubourg and Goss, 1989; Resnick, 1993) has shown that the coordination of very simple reactive agents can lead to complex behaviours, capable of solving problems far beyond the simple sum of the individual capacities of each agent.

The limitation of each individual agent makes the strength of the multi agent system, by forcing the local and independent resolution of the global task. In the early example of the termite colony model, the individual agent has an extremely limited perception (it can only sense what it is directly standing on) and a very simple reactive control mechanism, and yet collectively they are able to collect resources(chips of wood) to form a pile, and even sort different kinds of resources, without any planification, contract (beyond the fact that each agent is identical) or central control.

The termite algorithm is as follows:

- If I'm not carrying a chip, wander around until I find one to pick up.
- If I'm carrying a chip, wander around until I find a new pile of chips
- If I'm carrying a chip and I found a pile of chips, wander around until I find a free space to put down my chip

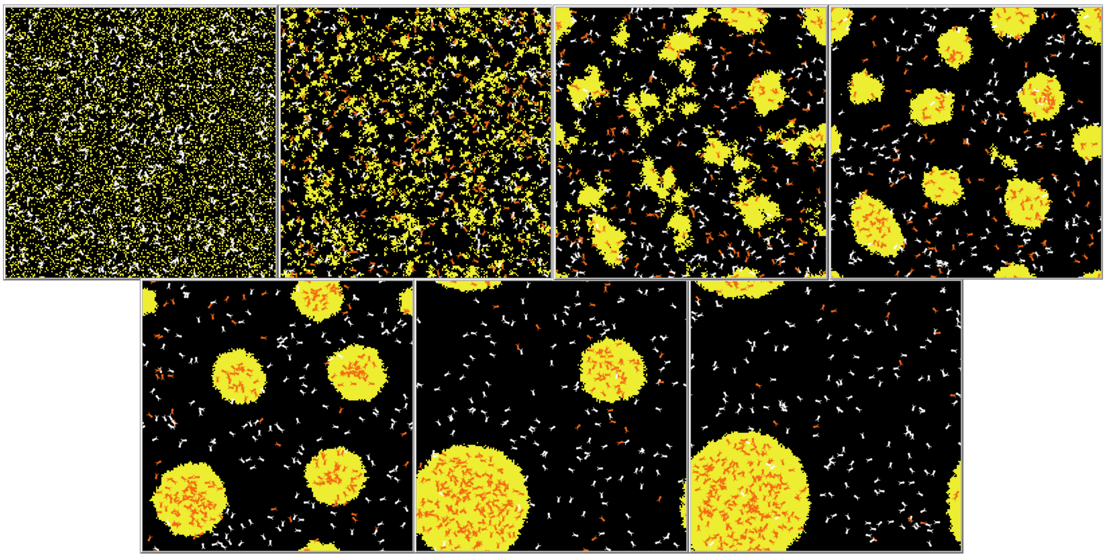


Figure 2.30: The termite colony model, top left: initial state, bottom right: final state.

Figure 2.30 shows that even with such simple reactive agents operating without communication, the simulation converges towards self-organization.

By using simple signals, such as trails of pheromones left in the environment, tasks of higher complexity can be achieved without an increase of complexity in the reactive agents themselves. In the ant foraging model presented in figure 2.31 (Deneubourg and Goss, 1989), once an ant finds (or rather, stumble upon) food, it picks up a piece and leave a trail of pheromones on the way back to the nest. Other ants follow the pheromones back to the food and reinforce the pheromone trail. The system reduces the exploration time by sharing the benefit of the discovery made by an individual ant. Using the same behaviour model with different food distribution in the environment lead to foraging paths comparable to the paths made by different species of ants, whose environment has the corresponding food distribution.

Figure 2.31: Top row: foraging patterns of three different army ant species, bottom row three runs of the same simulation model using corresponding food distribution patterns (adapted from Deneubourg and Goss (1989)).

A number of solutions to spatial problem for artificial agents were derived from the analysis of such behaviour models, such as exploration, path planning, covering or patrolling an environment. The EVAP model (Chu et al., 2007) for multi-agent patrolling is based on the evaporation of digital pheromones. Each agent leaves a trail of pheromones in a shared representation of the environment, through an evaporation process the quantity of pheromones in one place decreases over time. This process creates a gradient in the environment, the lowest values represent places which have not been visited for the longest time, patrolling is accomplished by following the descending gradient. This model was implemented on a fleet of UAVs in charge of patrolling an airbase against coordinated intrusions (Legras et al., 2008).

2.5.2 Social organization: Agent-Group-Role

By increasing the complexity of communication, from the exchange of simple signals to the exchange of messages, it is possible to move from simple reactive coordination to collaboration between agents. Collaboration implies the planning of heterogeneous tasks, either by a heterogeneous population with different abilities, or a homogeneous population assuming different roles.

Each complex collaboration task requires its specific social structure, this relation between agent does not necessarily fit other complex tasks. In order to organize efficiently, multiple formal relation between agents must be defined, which can be dynamically activated depending on the situation.

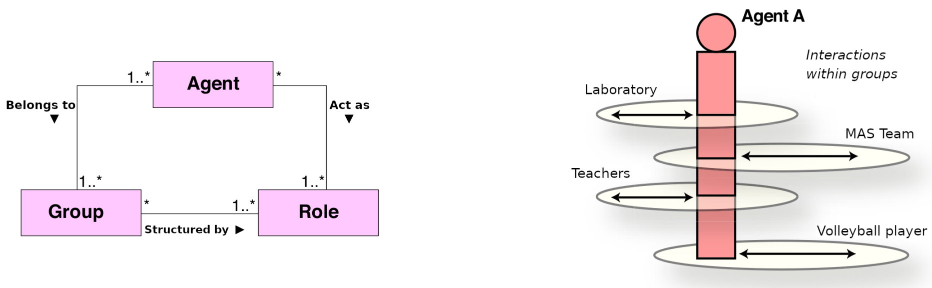


Figure 2.32: On the left: the UML diagram of AGR. On the right a familiar agent belonging to several communities (translated from Gutknecht (2001)).

The Agent-Group-Role (AGR) organization (Gutknecht, 2001) give a minimalist framework for defining such relations, which can be adapted to all kinds of architecture.

In AGR, an *agent* can join *groups* in which he will play one *role*.

The group is a socio-cognitive class which define the relation between Roles, each definition is suited to accomplish a specific collaboration task. These static definitions can be instantiated as the situation requires.

From this simple system, complex models can be formed when agents belong to several groups. In the model of the travelling agency, the travel agent belongs to the customer group and the professional group and acts as a broker between agents belonging to these two groups. When a client request matches an offer from his professional groups, a separate contract group is formed with one professional and one client, the nature of their relation being already established.

This organization into clearly defined groups greatly simplifies communication between agent, by restricting the context of the exchange of messages. An agent sending a message *as a buyer* to an agent considered *as a seller* within a contract group already carries a lot of meaning. For instance a simple exchange of number back and forth is enough to represent bargaining for the price, until the numbers match.

This simplification of the number of possible messages, and of the information content of messages, saw use in the context of collaboration in a fleet of autonomous underwa-

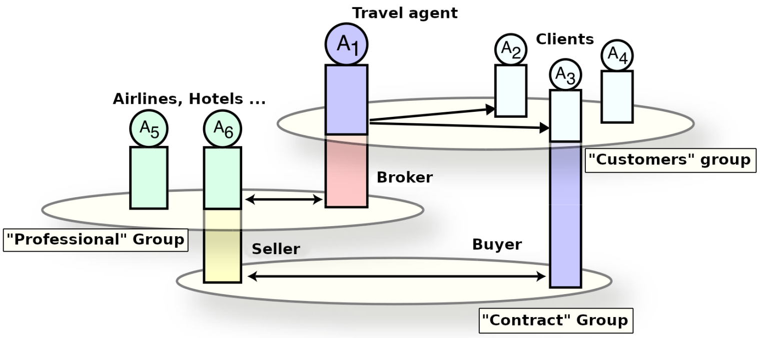


Figure 2.33: The AGR model of the travel agency (translated from Gutknecht (2001)).

ter vehicle (Carlési, 2013), where the underwater environment put many constraints on wireless communication.

In the context of models and simulations, AGR saw further development (Ferber et al., 2004) in the form of AGRE (Agent-Group-Role-Environment) (Ferber et al., 2005), the MadKit Platform (Gutknecht et al., 2000) and ultimately to the MASQ and MetaCiv frameworks.

2.5.3 MetaCiv

To understand the evolution of human societies is an extremely challenging problem because it involves multiple viewpoints expressed by experts from different fields: sociologists, anthropologists, geographers, archaeologists, historians, economists, fishery scientists, soil scientists, etc. Existing approaches do not consider enough the generic structures that are involved and do not make use of a solid conceptual analysis. They do not take into account the complex structures that could emerge, i.e. strong emergence Müller (2004) or multi-level emergence Beurier et al. (2002). Human societies are extremely complex systems and modelling them implies to consider together a whole set of *socio-cognitive items* such as entities, relationships, and structures from various perspectives, i.e. cognitive, social, cultural, organizational, economic, environmental, etc. In addition, in order to understand the dynamics of such systems, it is necessary to consider all the socio-cognitive entities in interaction and adopt a global and integrative perspective. For example, when studying how a goods market and the corresponding new economical forms are created, it is important to consider not only the basic economical mechanisms, but also the movements of people for creating new settlements or the social stratification they have, such as roles and status.

MetaCiv is a generic framework for modelling and simulating complex human socio-cognitive systems based on an extension of AGR: the MASQ meta-model. MASQ (Ferber et al., 2009; Stratulat et al., 2009; Dinu et al., 2012) is a meta-model that offers a map for understanding complex social systems and a tool to consider them from the perspective of the 4-Quadrants approach by Wilber (Wilber, 2001). According to MASQ, a social system is seen as a multi-agent system whose components that can be positioned along two axes: individual-collective and internal-external. The intersection of these axes gives rise to four quadrants as shown in figure 2.34. The main scenario in MASQ is that of an individual which uses its mind to make decisions (quadrant I-I) and its body (quadrant I-E) to act in a physical or social space where it will interact with other bodies or objects (quadrant C-E). The result of this interaction is perceived and interpreted internally by the agent according to the culture in which the agent is immersed (quadrant C-I). MASQ allows an intuitive mapping of the agent-group-role to respectively, Individual, Collective-external and Collective-internal.

Structuring the socio-cognitive items by following the conceptual framework of MASQ, MetaCiv is able to integrate social and cultural aspects of interaction, merging the cultural aspects of groups (norms, standards, shared beliefs) with individual motivations at the decision-making level of each agent. This allows a modeller to have a vision of culture and norms as a source to influence the behaviour of the agent and not as regimented or compulsory system, as is usually the case for normative agents (Stratulat, 2002).

To consider both reactive and cognitive aspects to model the agent mind, MetaCiv uses a hybrid agent architecture based on *cognitons* (Ferber, 1995) for its decision process. The *cogniton*, a cognitive unit represent a part of the state of mind of an agent that will influence the choice of action of the agent.

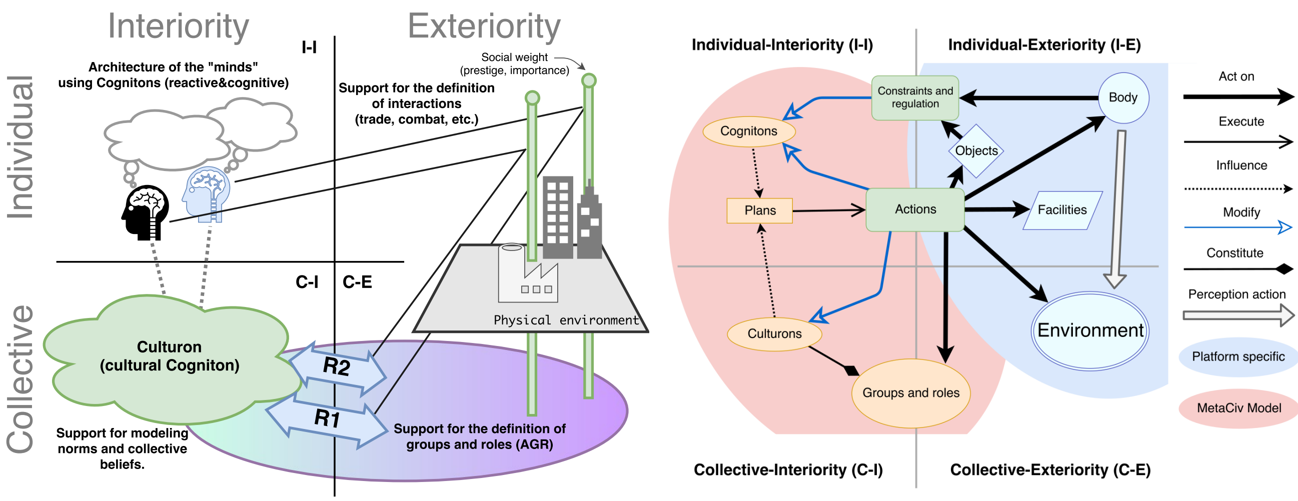


Figure 2.34: On the left: MASQ, on the right: MetaCiv.

The next chapter will present the cognition architecture in depth and show works with MetaCiv on social specialization.

Chapter 3

Preliminary work on Multi-Agent Systems

3.1 Introduction

This chapter presents preliminary work on MASQ (Ferber et al., 2009; Stratulat et al., 2009; Dinu et al., 2012) and MetaCiv meta-models and the Cogniton-based agent architecture (Ferber, 1995). Here, we explore ability to learn a social specialization in a multi-agent simulation.

The cogniton architecture is one of the sources of inspiration for our main contribution, as the use of the term Influence testifies. Compared to other meta-models for multi-agent simulations, one of the particularity of MetaCiv which concerns us is how detailed the individual-interiority of the agent is, and the potential complexity of its mental states.

By creating and experimenting with CogLogo, an implementation of MetaCiv and its cogniton architecture, we were able to see the potential and the limitations of the models in regard to our work. Namely, agents are able to learn a social specialization through reinforcement mechanisms, but are only able to choose from a pre established set of plans (programming procedures). This is of course the difference in scope between a multi-agent simulation built for the purpose of experimenting with social theories and the control system of a learning agent.

The following presents the cogniton architecture, its implementation and an experiment on social specialization.

3.2 MetaCiv

In MetaCiv, according to MASQ principles, each agent has a physical body (its Individual Exteriority) which is separated from its mind (its Individual Interiority). The Cogniton-based architecture (Ferber, 1995) is intended to unify cognitive and reactive aspects to represent the mind of an agent. We qualify the architecture of unifying in that it differs from hybrid architectures by not distinguishing a reactive level and a cognitive level, but by incorporating all factors allowing an agent to make a decision such as beliefs, percepts, skills, norms, etc. Although this type of architecture is not new *per se* (it was introduced in the early '90s to model reactive agents), it draws its power in its ability to integrate individual and collective aspects by just using the same representation.

3.2.1 Cogniton-based agent architecture

The cogniton-based architecture consists of two basic elements: *cognitons* and *plans*.

1. The cognitons are "basic cognitive units" (Ferber, 1995) that can be dynamically added or removed from the mind of the agent. Each cogniton has its own weight in the agent mind, this value can evolve during the lifetime of the agent. The cognitons can represent beliefs, knowledge or percepts. All the cognitons present in the mind of an agent at a given time form its mental state.

2. The plans represent the different activities that an agent can perform. Each plan consists of a sequence of actions of varying complexity. The decision-making process of the agent assigns to each plan a weight determined by the cognitons present in the mind of the agent. A plan can then be selected according to various strategies: by selecting the largest weight, by a random draw that favours the strong weight, etc.

In order to compute the weight of a plan, we propagate the weights of the cognitons to each plan through the *influence links*. The total weight of a plan can be computed using the formula 3.1.

$$weight = \sum_{i} C_i L_i \tag{3.1}$$

Where i is an influence link, C_i is the weight of the cogniton linked by i to the plan, and L_i is the influence value of the link. The sum is performed on all the influence links to the cognitons that affect the plan. The influence of a cogniton on the plan is therefore described by a multiplicative function, a high weight amplifies the influence of a cogniton, while a low weight reduces it.

Figure 3.1 illustrates the use of cognitons in a simple context. In this example the cognitons A and B have links that affect the weights of Plan 1 and Plan 2. Plan 1 is influenced only by the cogniton A of weight 3 and the weight of the influence link is 4. Plan 2 is affected by both A and B of weights 3 and 2 respectively for which the weights of the influence links are -1 and 6. The final weights of Plan 1 and Plan 2 are respectively 12 and 9.

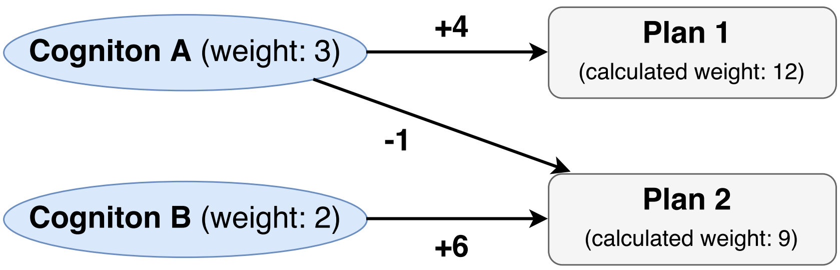


Figure 3.1: An example of the influence mechanism.

Plans allow an agent to act on its body, on the objects it manipulates, on the surrounding environment or on the groups to which it belongs. Plans may also affect the mind of the agent by acting directly on the agent's cognitons (through deletion, addition or modification of its weight). This feedback creates a reinforcement mechanism that strengthens or weakens the weights of the cognitons that contributed to the selection of the current plan.

3.2.2 Groups and culturons

The MetaCiv meta-model extends AGR (Ferber et al., 2004) by associating to each role within a group a set of culturons that represent cultural elements common to all agents sharing the role. The culturons work similarly to cognitons: when an agent plays a role, it assimilates all the culturons associated to the role, which will then act on the weights of the plans from its mind, like the cognitons. Figure 3.2 illustrates this mechanism. In this example the agent belongs to the group α and plays the role β which contains two culturons, X and Y. By playing the role β, Plan2 will also be influenced by the culturons X and Y that alter its total weight.

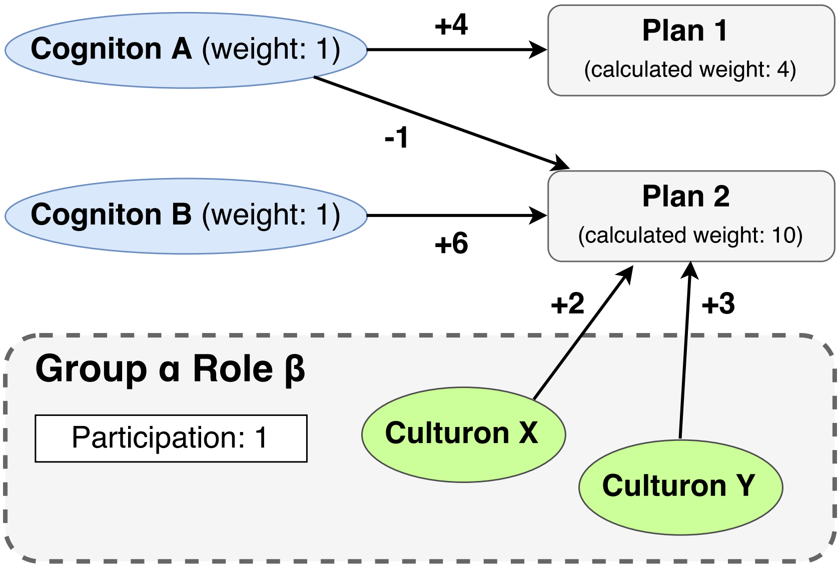


Figure 3.2: Culturons influences on the agent's plans.

As in AGR, an agent can belong to several groups and play many roles. All culturons from all the roles played by the agent are taken into account in calculating the weight of the plans. But unlike cognitons, the influence of culturons is also affected by the degree of participation of the agent to the group in which it plays that role. This degree is used as a factor in calculating the real impact of the culturon, as shown in formula 3.2.

$$weight = \sum_i C_i L_i + \sum_j (A_j \sum_k L_k Cu_k) \tag{3.2}$$

The weight is computed based on the influences coming from all the groups j in which the agent plays at least one role. A_j is the degree of participation in the group, and L_k are the weights of the influence links of the culturons Cu_k associated to the role. The final weight is composed therefore by influences coming from cognitons $C_i L_i$ and culturons.

Each agent belonging to an instance of a group can modify the weights of the culturons for his instance of the group, the weight values of the culturons are shared by all the agents belonging to this instance of the group (a different instance of the same type of group will have its own culturon weight values).

3.2.3 Environment, buildings, objects and bodies

The cogniton architecture is dedicated to modelling the mind of the agent, the impact on the environment and objects all take place through the agent's body. In Figure 2.34 you can see that MetaCiv only covers the elements of the cogniton architecture while letting you bring your own implementation via a platform of your choice, where you will describe the environment, buildings, objects and all the physical interactions between them and the body of the agent.

3.3 Experiments with CogLogo

In order to illustrate the advantages of the architecture based on cognitons, we present an example of simulation developed with CogLogo (Suro, 2017), a NetLogo (Wilensky, 1999) implementation of MetaCiv (described in subsection 3.3.2).

3.3.1 A simulation example

In this example we simulate a society of agents who all live directly from agriculture. The agents initially need the wheat produced by their farming to survive. The goal is to simulate a single transition to a society made up of farmers and artisans. The former always produce wheat, while the latter produce tools. Farmers are more effective in producing wheat if they have tools, and artisans need wheat produced by farmers to survive. It is therefore necessary to show how the goods exchange between agents leads to their specialization into artisans and farmers.

At the beginning of the simulation each agent in the system starts with a $Farmer$ and an $Artisan$ cogniton, which represent the inclination of the agent to do agricultural or crafts work. The $WheatStock$ and $ToolStock$ cognitons correspond to the knowledge about the amount of wheat or the number of tools available to the agent. They are reinforced if the agent has an important reserve of the resource. The $WheatNeed$ cogniton indicates that an agent does not have sufficient reserves of wheat and if it should get them. The $WheatDemand$ and $ToolDemand$ cognitions represent the belief the agent has about the current demand on the market or the needs of others agents concerning wheat or tools.

Each agent is also endowed with four plans, which represent the four types of activities that can be carried out: TradeWheatForTools, TradeToolsForWheat, GrowWheat, ProduceTools.

Figure 3.3 represents the different influence links that bind cognitons and plans in this model. The cognitons are represented by ellipses, and the plans by rectangles. The dotted arrows represent links that inhibit the urge to execute a plan, while the solid arrows represent links that reinforce the need to execute a plan.

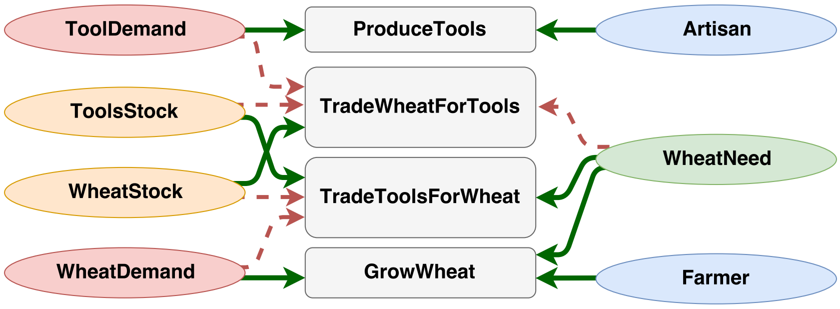


Figure 3.3: Representation of influence links between cognitons and plans.

- TradeWheatForTools is reinforced by $WheatStock$ and weakened by $ToolsStock$. Indeed, an agent does not wish to buy tools if it already has enough of them, or sell wheat if its stocks are low. It is also inhibited by $WheatNeed$ and $ToolDemand$, the agent is less likely to try to acquire tools if it knows they are scarce.
- TradeToolsForWheat works symmetrically with the previous plan.
- GrowWheat is reinforced by $WheatNeed$, $Farmer$ and $WheatDemand$. Indeed, if an agent knows that wheat is a popular commodity, it will find more interesting to produce it.
- ProduceTools is similar to the previous plan, but is not affected by a need.

Now that the factors that affect the decision-making process have been defined, it is necessary to specify how the actions of the agents change their cognitons so that they adapt their behaviour. The feedback thus obtain is shown in Figure 3.4. The arrows symbolize the impact of a plan on a cogniton which can be of various kinds: add, remove, modify weight, etc.

The set of actions that can change the reserves of wheat or tools will indirectly strengthen or weaken $WheatStock$ and $ToolStock$ cognitons. $WheatNeed$ is influenced by the same factors as $WheatStock$.

When an agent attempts to trade, it updates the market demand by observing the current tool and wheat stocks of the marketplace. This value is transmitted to the $Demand$ cognitons representing the needs of the community. This mechanism allows the group of agents automatically regulate their exchanges. Indeed, when a resource is not produced in sufficient quantity by the community, the corresponding $Demand$ cogniton is reinforced. Strengthening these cognitons will motivate the agents to produce the resource and satisfy the need.

When an agent succeeds in trading wheat or tools, it will strengthen the associated cogniton ($Artisan$ or $Farmer$), which will allow the agent to specialize (social learning). Simply put, this represents the fact that earning a living through an activity makes you a professional (or specialist), not the exercise of the activity itself.

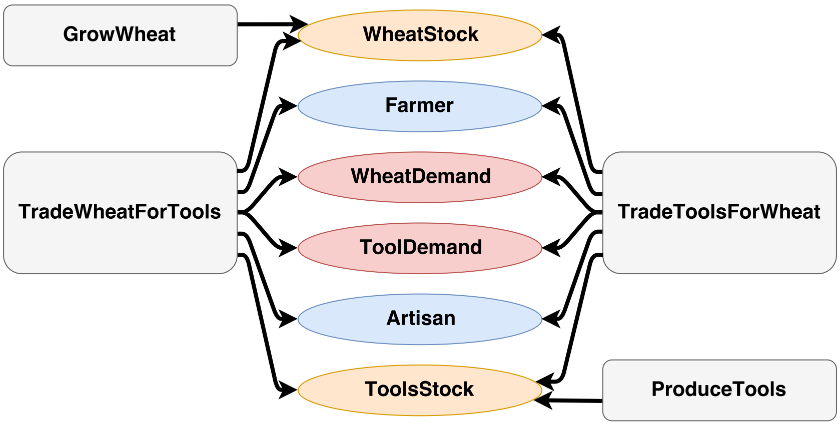


Figure 3.4: Influences of agent activities over cognitons.

To summarize let's consider the agent shown in Figure 3.5. The weights of *Artisan* and *ToolDemand* are high, which strengthen the ProduceTools plan. However, the most interesting plan for this agent remains TradeToolsForWheat, because it has a large stock of tools and its cogniton *WheatNeed* has a high influence value. If this agent executes the plan of trade and finally obtains the wheat, the need for wheat and the stock of tools will decrease and the stock of wheat will increase. These changes will affect the weights of the corresponding plans and the next decision of the agent.

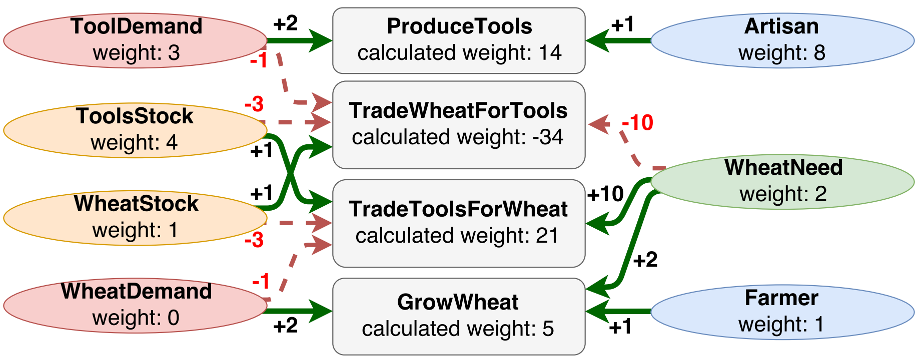


Figure 3.5: An example of the mental scheme of an agent during simulation.

CogLogo can model very intuitively the multi-agent system we have just presented. The simulation is dynamic, self-regulating, and is able to handle transitions: initial agents are all farming by necessity, and end up specializing in either artisans or farmers. The hybrid nature of the architecture allows mixing reactive reagents aspects such as the need for food with cognitive elements, such as taking into account the needs of the community, through *Demand* cognitons.

3.3.2 The CogLogo extension

CogLogo (Suro, 2017) is a NetLogo (Wilensky, 1999) extension providing an implementation of the cogniton architecture which integrates with NetLogo's description of environment, physical agents, objects and interactions.

CogLogo integrates with NetLogo by providing procedures to modify the cognitons weights, organize groups of agents and modify culturons weights. The procedure *coglogo:get-next-plan* is called at the modeller's discretion, and returns a string which can be used to call any corresponding NetLogo or user defined procedure.

CogLogo has a graphical editor to design the internal cognitive architecture of the agents: the *Cognitive Scheme*. The *Cognitive Scheme* describes the individual thought process with cognitons and collective elements with culturons.

The cognitive scheme editor is used to create the cognitons and culturons, and the influence links to the plans. The value of each influence link can be defined as a positive or negative real number.

CogLogo also offers two kinds of links not discussed before, the conditional links and the reinforcement links, which we will briefly explain.

The conditional links play a role in the simulation by telling the decision maker which plans are available. Cognitons can be activated and deactivated, and a plan can take part in the selection process only if it has at least one conditional link to an active cogniton (even if the cogniton weight is 0). This means that even if, through other cognitons, a

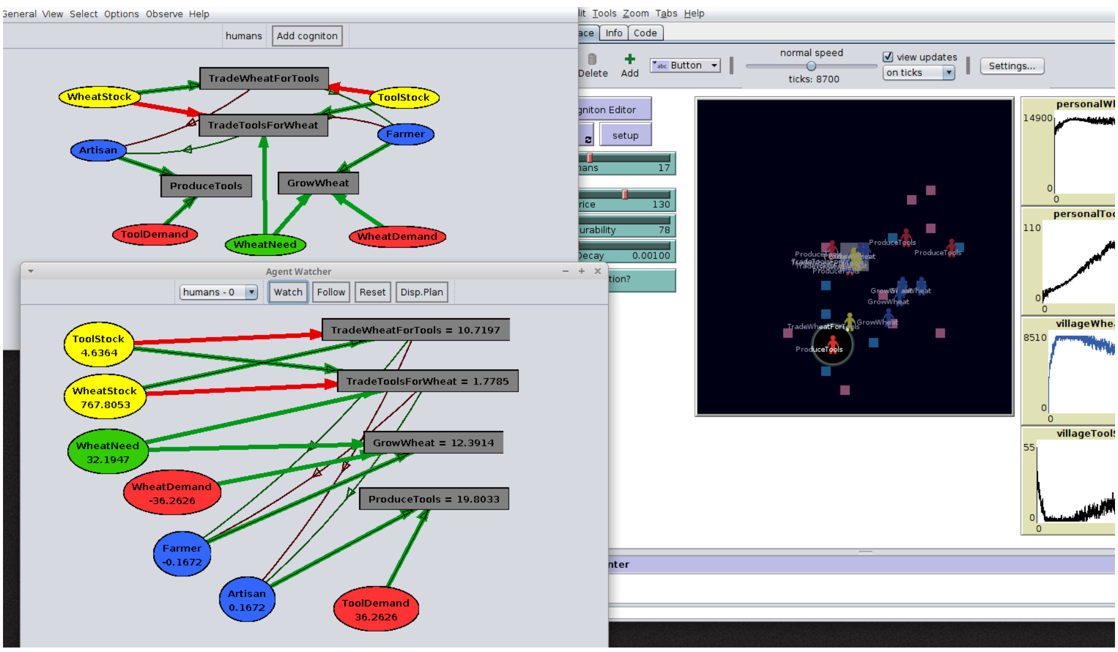


Figure 3.6: On the right the NetLogo window, on the top left the cognitive scheme editor, on the bottom left the agent watcher displaying the state of the cognitons and the calculated values of the plan in an agent's mind

plan obtains the highest calculated weight, it will never be selected without a conditional link to an active cogniton. For instance, many factors could motivate an agent to buy food, however the *BuyFood* plan will never be considered by the agent if it knows it does not have money. The *Money* cogniton, which might have reached a negative value, has been deactivated.

The reinforcement links provide a simple and more readable way to implement the reinforcement mechanism. While the evolution of certain cognitons are more a matter of perception and regulation (such as hunger, climate influence or age), others are involved in the long term behaviour of the agent and reflect a learning process (such as a social specialization or an attitude towards external factors). When an agent runs a plan, a feedback operation (*coglogo:feed-back-from-plan*) can be called with a value parameter, this value is multiplied by the weight of the reinforcement link defined in the cognitive scheme and added to the corresponding cogniton.

The use of reinforcement links is entirely optional and the same effect can be accomplished by using the regular procedures to set the values of the cognitons. They are simply meant to simplify reinforcement mechanisms and help the modelling process by making them visible in the CogLogo interface.

Multiple Cognitive Schemes can be defined and used in the same model to represent different species interacting or different social organization in cultures. Cognitive Schemes can be assigned to different agent kinds (NetLogo's *breeds*). Each Cognitive Scheme chooses its decision maker:

- MaximumWeight: the plan with the maximum weight is selected.
- WeightedStochastic: the weight of each plan represents the probability for the plan to be selected. If PlanA = 5 and PlanB = 3 , the probability of being selected is: PlanA = 5/8 = 0.625 and PlanB = 3/8 = 0.375. a random function (0,1) is then called to select the plan.
- BiasedWeightedStochastic: works the same way as the WeightedStochastic, but increases the probability of selecting the better plans according to the bias factor. The plans are sorted from the highest probability to the lowest, covering the range from 0 to 1 (the highest probability covers the range from 0 to P_0, then the next plan from P_0 to P_1, and so on). The random function result is then elevated to the degree of the bias specified, which will bias the random function towards giving values closer to 0. (note: a bias of 1 has thus no effect on the random draw and will give the same results as the regular WeightedStochastic decision maker, but with a slightly higher computing cost)

CogLogo also provides tools to observe its elements during the simulation. The agent watcher window will show the weight of each cogniton and the calculated weights of the plans for a selected agent in real time.

Coglogo is an open source project and is available at:

https://gite.lirmm.fr/suro/coglogopublic
https://github.com/suroFr/CogLogo

3.3.3 Setting up the simulation with CogLogo

To prove the merits of MetaCiv and illustrate the ease of use of the CogLogo extension, we implemented the simulation example described above and observed the results. The model for the cognitive scheme that we used has already been described, the specific weights of the influence links are shown in figure 3.7.

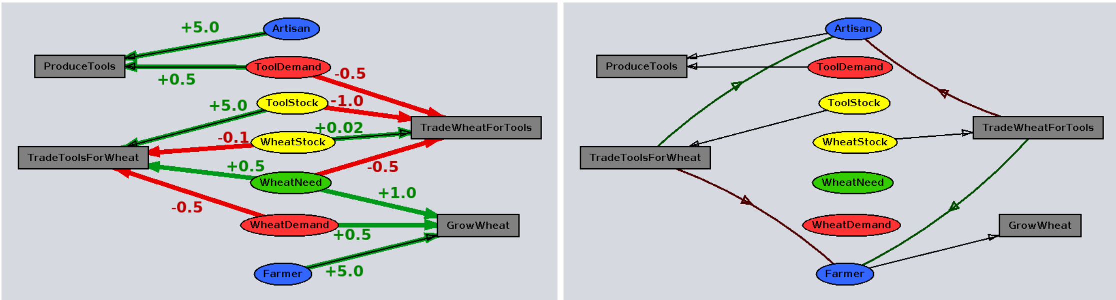


Figure 3.7: On the left, the influence links and their respective values. On the right, the conditional links (straight lines) and the reinforcement links (arcs).

In the right panel of Figure 3.7 we can see the use of the conditional links. In our simulation the only cogniton that is activated and deactivated is the $ToolStock$ cogniton, the others are always active. The $ToolStock$ cogniton is deactivated each time the agent has no tools, and activated when it gets one. What this means is that even if the $TradeToolsForWheat$ has the highest calculated weight, because of the influence of the $WheatNeed$ cogniton, it will not be selected if the agent does not have any tool to trade (the $ToolStock$ cogniton will have been deactivated).

We also described the reinforcement mechanism of the $Artisan$ and $Farmer$ cognitons with reinforcement links. In our simulation, a success in trading tools for wheat motivates the agent to specialize in craftwork (and decreases the interest in farming), while a success in trading wheat for tools does the opposite (see Listing 3.3).

In addition to this reinforcement mechanism, a regulation function is applied by a NetLogo procedure, keeping the values of $Artisan$ and $Farmer$ between bounds and adding an eroding effect: if the agent does not participate in any trade activity, its $Artisan$ and $Farmer$ cognitons will very slowly return to their initial values (0.0).

Listing 3.1 shows the agent main loop. As you can see the output of *coglogo:choose-next-plan* is not run directly but stored in a variable. This allows for the selected plan to be run over several ticks of the simulation, for as long as the *planTimer* variable is different than zero. This is a simple way to allow for persistence of action while at the same time allowing to interrupt the action under certain conditions (a plan can simply set the variable *planTimer* to zero, forcing a call to the decision at the next tick).

Listing 3.1: the agent main loop, called at each tick of the simulation

```
to goHuman
    regulatePerceptionCogniton
    regulateSpecialisationCogniton
    checkStarvation
    coglogo:report-agent-data
    ifelse planTimer <= 0
    [
      set currentPlan coglogo:choose-next-plan
      set planTimer planDuration
      run currentPlan
      set wheatStock wheatStock * (1.0 - foodDecay)
    ]
    [
      set planTimer planTimer - 1
      run currentPlan
    ]
end
```

Listing 3.2 shows how the simple "perceptive" cognitons are assigned their values.

Listing 3.2: transferring precepts to the agent's cognitons

```
to regulatePerceptionCogniton
  coglogo:set-cogniton-value "WheatDemand" wheatDemand
  coglogo:set-cogniton-value "ToolDemand" toolDemand
  coglogo:set-cogniton-value "WheatStock" wheatStock
  coglogo:set-cogniton-value "ToolStock" toolStock
  coglogo:set-cogniton-value "WheatNeed" ((800 ) - wheatStock)
end
```

Listing 3.3 shows the $TradeWheatForTools$ plan. In our model, all trade takes place in the village at the center of the map, which has a stock of both tools and wheat. This allows for simple trade without planning (finding a buyer and seller). These village stocks are used to analyze the market and influence the wheat and tool demand.

You can see at line 10 that in case of a successful trade, the *coglogo:feed-back-from-plan* procedure is called, reinforcing the Farmer behaviour If a successful trade has taken place, that means the agent now possesses a tool, so we call *coglogo:activate-cogniton* on the $ToolStock$ cogniton.

If the trade was successful, or if the village does not have any tools to trade, the planTimer variable is set to zero, forcing a new decision to be made at the next tick. This does not happen when the agent is on its way to the village ($goToVillage$).

Listing 3.3: the TradeWheatForTools plan

```
to TradeWheatForTools
  ifelse villageToolStock > 1
  [
    ifelse patch-here = patch 0 0
    [
     set villageToolStock villageToolStock - 1
     set villageWheatStock villageWheatStock + (1 * baseToolPrice)
     set toolStock toolStock + 1
     set wheatStock wheatStock - (1 * baseToolPrice)
     coglogo:feed-back-from-plan "TradeWheatForTools" random-float 10
     coglogo:activate-cogniton "ToolStock"
     set wheatDemand wheatDemand + (baseDemandIntensity
                 * ( - demandPerception))
     set toolDemand toolDemand + (baseDemandIntensity * demandPerception)
     set planTimer 0
    ]
    [goToVillage]
  ]
  [
    set wheatDemand wheatDemand + (baseDemandIntensity * ( - demandPerception))
    set toolDemand toolDemand + (baseDemandIntensity * demandPerception)
    set planTimer 0
  ]
  set color yellow
end
```

Finally, the resources are managed as follows. The wheat stock of the agent decreases with time, and decreases proportionally to the current stock. This reflects the impossibility to accumulate perishable resources indefinitely, and the necessity for constant production and trade. The tool stock decreases only when a trade takes place or when the agent produces wheat: some of the tool's durability is exchanged for a higher wheat output. Tools are the only resources the agents are able to accumulate.

The complete code of the model is available in the CogLogo project repository (Suro, 2017)

3.3.4 Simulation results

Here we show a simulation run with 25 agents over a 500 000 ticks period.

The colour of the agents indicate their current task. Red means they are producing tools, Blue is for producing wheat and yellow indicates they are on their way to the village to trade.

The colour of the private land of the agent indicates the activity carried there over the last 25 000 ticks. Each tick a tool is produced, a fraction of red is added. Each tick wheat is produced, a fraction of blue is added. Purple indicates an equal time is dedicated to wheat and tool production.

We can observe specialization early on. Different factors affect the speed of the specialization, the right panel of Figure 3.8 shows the proximity to the village. This is so simply because the shorter travel time allows the agent to trade more often, and by the definition of our model, to trigger the reinforcement mechanism more often.

From tick 100 000 to tick 200 000 (Figure 3.9) almost all agents have specialized. There is slight surplus production of tools.

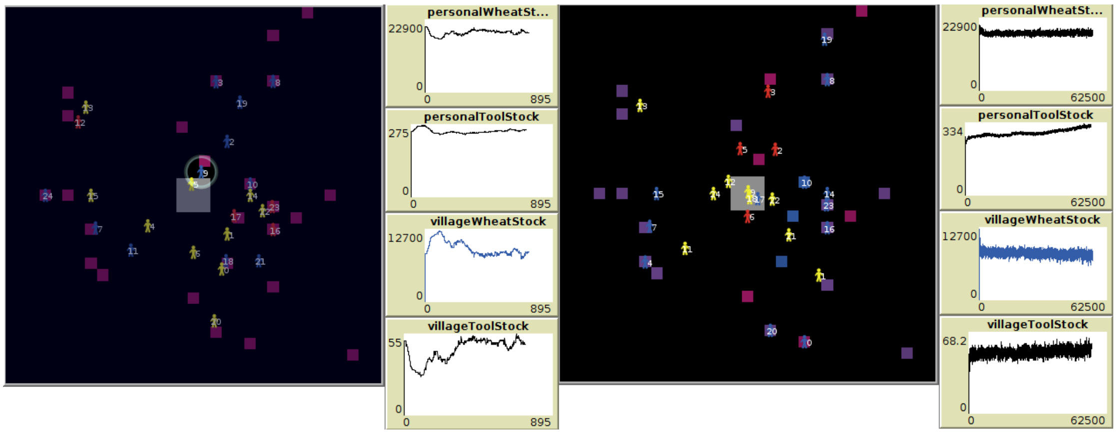


Figure 3.8: On the left, the initial state of the simulation, on the right the simulation at 27 000 ticks.

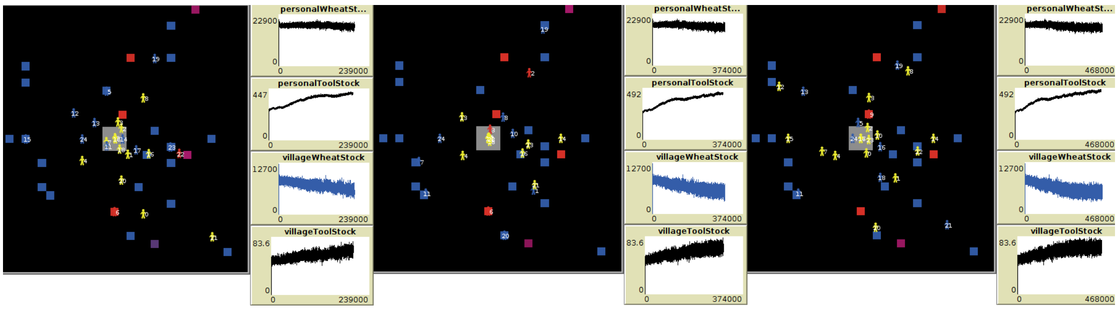


Figure 3.9: From left to right, the state of the simulation, at 100 000, 150 000 and 200 000 ticks.

On the right panel of Figure 3.11 we can see that the agent 6 (circled) is producing wheat despite its land showing that the main activity is tool production.

In Figure 3.12, we see that despite being an artisan, because of the demands of the market and its own needs, it is more advantageous for agent 6 to grow its own wheat.

The behaviour of agent 6, and other agents under the same constraints, has had an impact on the global production. In the personalToolStock plot of Figure 3.13 we can see the time used by the less specialized artisans to produce their own wheat has made a dent in the global production of tools. It seems it even affected the market as shown by the village stock.

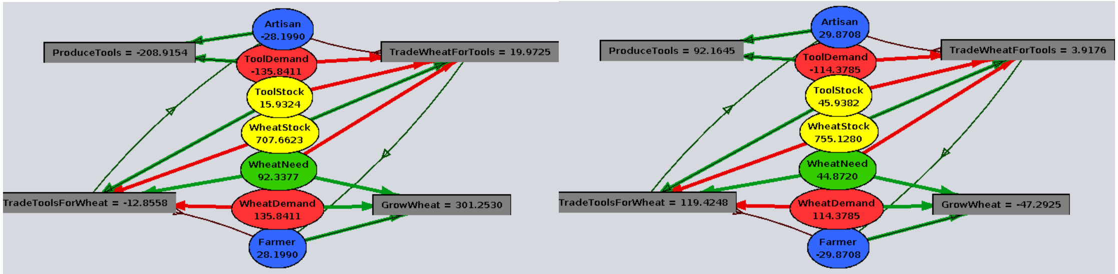


Figure 3.10: Internal states of agents: on the left, an agent specialized as a farmer, on the right, as an artisan

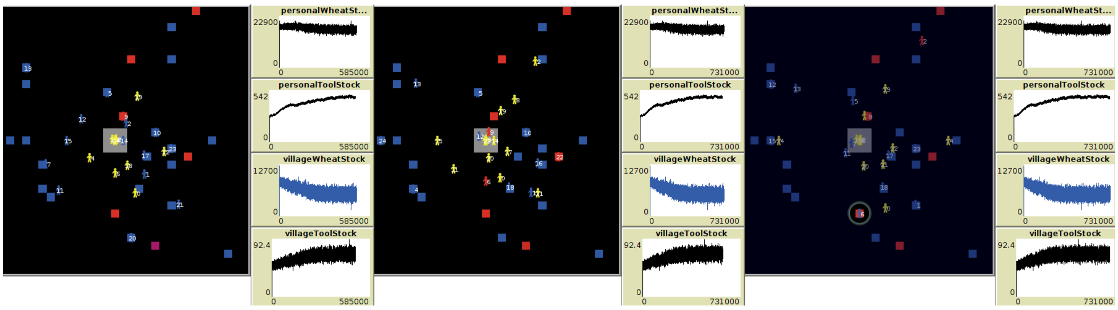


Figure 3.11: From left to right, the state of the simulation, at 250 000, 300 000 and 350 000 ticks

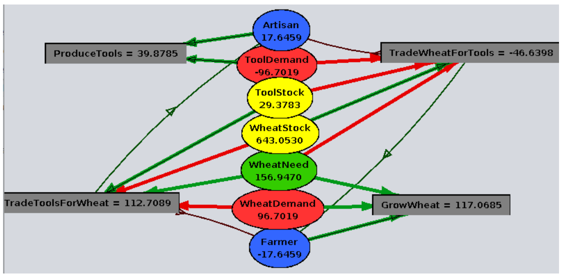


Figure 3.12: The internal state of agent 6 at tick 350 000

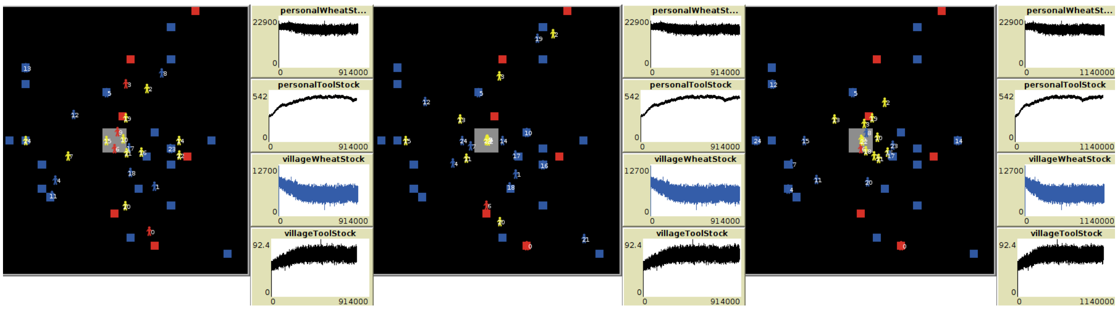


Figure 3.13: From left to right, the state of the simulation, at 400 000, 450 000 and 500 000 ticks

In comparison, Figure 3.14 shows another simulation run, at tick 300 000. This particular run evolved with a deficit of artisans. As you can see there are only 4 lands which produce tools full time, instead of 6 in the previous run. You can observe that agent 12 produces tools even if its land indicates that its main activity is farming. This is actually the case of an agent changing of activity due to the high demand of tools. The right panel of Figure 3.14 shows the internal state of an agent during a similar transition. Even if this agent is still specialized as a Farmer, the high demand for tools makes *ProduceTools* compete with *GrowWheat* in our stochastic decision process, and since other farmers are more specialized than this agent (they have a higher *Farmer* cogniton weight), it is the first to be influenced by the demand. By producing more and more tools and trading them the agent will eventually specialize into an artisan, providing more tools and diminishing the tool demand and thus preventing others from changing specialization.

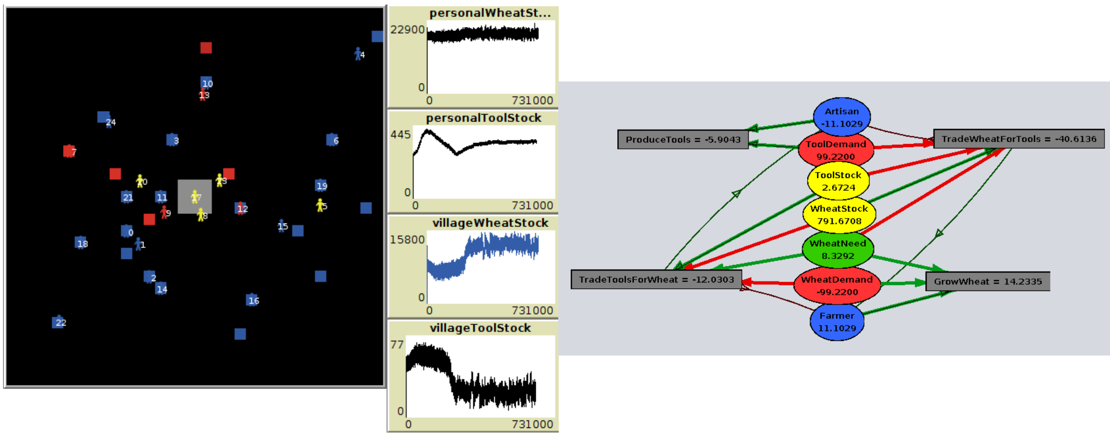


Figure 3.14: An other run of the same simulation at 300 000 tick. The initial deficit of artisans is stabilizing.

A Video of this particular outcome is available at the following address: https://hal.archives-ouvertes.fr/hal-02924858v1

3.3.5 Analysis

The Coglogo simulation we present illustrates the ease of use and potential of the software, but beyond that it is intended to demonstrate the power of the MetaCiv meta-model and its cogniton architecture. The simulation shows the emergence of a specialization in the population through the reinforcement mechanism, driven by trade. This run of the simulation, along with others, shows the occurrence of an economic equilibrium approaching from both the over and under production of tools, and the transition of roles for individuals to fit the economy. This validates MetaCiv as a powerful meta-model for simulating human societies.

3.4 Conclusions

The purpose of MetaCiv is to be a meta-model for multi-agent simulations of complex social systems. As a multi-agent simulation, it is set in a certain scope, or granularity, in order to simulate a moderate number of agents at a reasonable computing cost. Its aim towards the emergence of social systems does involve a learning aspect in learning a social specialization, however it presupposes already skilled agents and does not consider the development of skills themselves.

The emergence of a society of learning agents is an ambitious and impassioning goal from the developmental robotics point of view, that must be reached through a process that starts with the acquisition of sensori-motor skills.

Nevertheless, MetaCiv offers us a few leads towards the creation of an autonomous learning control system:

- The relation of the Cogniton to the Plan through a weighted influence link offers a connectionist approach to the decision process (which is later turned back into symbols through the plan selection).
- The Cogniton is a numerical value, a real number, which represents indifferently external stimulus or internal states of the agent.
- Cognitons which represent a specialization of the agent accumulate values over a long period of time, through a reinforcement mechanism.
- It is obvious in successive state-action chains, that the actions of an agent will affect its perceptions, for instance movement will change its point of view. However, the reinforcement mechanism can suggest that an action can also affect an internal state, without necessarily causing a perceptible change in the environment.

During discussions on giving a greater complexity to the decision process, allowing for a finer granularity of action, it was suggested that cognitons could affect Meta-Plans representing general ideas of what to do. Once a Meta-plan is selected, the cognitons would influence the selection in a group of sub-plans, representing the different or alternative steps of accomplishing the general purpose of the Meta-Plan. This is comparable to the ICARUS (Choi and Langley, 2018) skill execution and balanced commitment, the beliefs being represented by cognitons. This idea was abandoned in favour of keeping the defined scope of MetaCiv, but it lived on in MIND, our main contribution which we present in the next part.

Chapter 4

MIND: Modular Influence Network Design

As we discussed in chapter 2, the field of developmental robotics has provided many contributions towards the creation of autonomous agents capable of learning. However, open questions remain on the ability to integrate this great diversity of techniques to cooperate and support each other in a process of lifelong development (Oudeyer, 2012).

To build a developmental agent, the different aspects of developmental robotics, from control primitives to learning algorithms, motivational systems and memory structures, must meet in a single system. Ideally this system must provide structures, and structuration techniques suited to the great diversity of state-of-the-art solutions; for skills (classifiers, neural networks, pre-programmed skills, etc.) or learning mechanisms (intrinsic motivation, supervised training, social learning, etc.). Our work will focus on providing a unifying architecture for diverse methods by use of modularity and delegation, that will suit the progressive and cumulative learning goals of developmental agents.

MIND (Modular Influence Network Design) is a hierarchy of modules encapsulating skills which are able to coordinate through a mechanism called *Influence*, allowing the simultaneous composition of behaviours (see subsection 2.3.3). By combining, prioritizing and arbitrating between different subskills, MIND is able to accomplish complex tasks. The modular principle and coordination through influence fit the requirements of ongoing emergence (Prince et al. (2005), discussed in chapter 2) for skill representation: the continuous skill acquisition and their integration with previously acquired skill, their stability and identifiability for further use. The benefits of MIND reside in the following properties:

1. **Encapsulation**: generic behaviours will be encapsulated and combined with others in various ways to achieve different goals, rather than specialize a global behaviour to a specific task and losing the ability to branch from the original generic behaviour.

2. **Identifiable behaviours**: MIND gives the ability to identify and modify behaviour locally, working on a single aspect at a time.

3. **Unifying methods**: MIND places no constraints on the decision method used by each module, except for the input and output domains. This enables MIND to use neural networks, programming procedures and various other functions in the same network. MIND provides a way for the modules to organize with each other as a network, driven by influence.

4. **Flexibility of MIND**: behaviour modules can be replaced either by a new module or a hierarchy of modules favouring constant evolution of the system.

5. **Flexibility of body**: MIND provides a generic solution to the organization of sensors and actuators. The method used to coordinate related sensors and actuators into local groups is also used to coordinate all the groups in the system together. This also means new sensors and actuators can easily coordinate with an already existing system without losing previously acquired behaviours.

4.1 Base skill, complex skill, and influence

In the following we consider an agent whose sensory information and motor commands are represented as vectors of real numbers, normalized between 0 and 1. It is possible to create a module that encapsulates a function $f(x)$ that reads the input vector $V_I = [I_1, I_2, ..., I_i]$ and outputs the vector $V_O = [O_1, O_2, ..., O_j]$ (Eq. 4.1). The function f can be implemented as a programming procedure, or it can be a function approximator such as a neural network, or any other kind of function that associates two vectors of real numbers. We will call such a module a *skill*, and a module whose output vector is used directly as motor commands a *base skill*.

$$V_O = f(V_I) \tag{4.1}$$

The input vector V_I is supplied to the internal function $f()$ of the skill to produce the output vector V_O.

Braitenberg vehicles (Braitenberg, 1986), are examples of agents that directly associate an input vector of analog signals to a similar output vector. A MIND agent could use a single *base skill* to represent the wiring of a Braitenberg vehicle.

Using a single *base skill* provided with all the sensory inputs and all the motor outputs of the agent would be sufficient to learn how to perform a complex task, each lesson of the curriculum being memorized in the same unique structure. This monolithic skill would be the sum of all the different experiences, with no way to differentiate what has been taught.

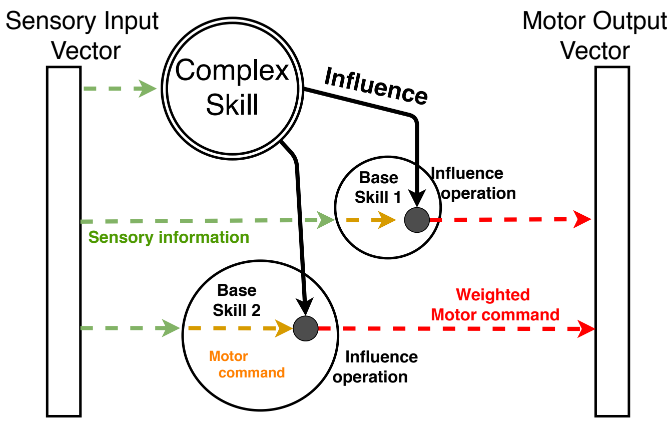


Figure 4.1: A *complex skill* influencing two *base skills*.

Instead of performing a complex task by a single skill, the complex task can be divided into subtasks, some even conflicting, to be performed by separate *base skills*. Each *base skill* only associates the inputs and outputs necessary to accomplish its designated task.

To perform the complex task, a *complex skill* is created which will coordinate several *skills*, that we call its *subskills* (Figure 4.1). A *complex skill* accomplishes coordination by sending to its *subskills* a signal called *influence* which determines how much weight (influence) a *subskill* has on the resulting action. This can be understood as delegating to one or a combination of *subskills* the resolution of the current task in the same fashion as the Boid brain coordinates its sub behaviours to accomplish the task of flocking (Reynolds, 1987).

A *complex skill*, as any skill, encapsulates a function that takes an input vector from the sensors V_I and outputs a vector of real numbers V_O, but the output is directed to its subskills. This output vector is called the influence vector $V_{Infl} = [Infl_1, Infl_2, ..., Infl_k]$, and its elements $Infl_x$ are called *influences*.

A complex skill can have other complex skills as its subskills, thus creating hierarchies of skills. At the top of the hierarchy is the *master skill*, a complex skill whose only particularity is to receive a constant influence of 1.0, an impulse setting the whole process in motion.

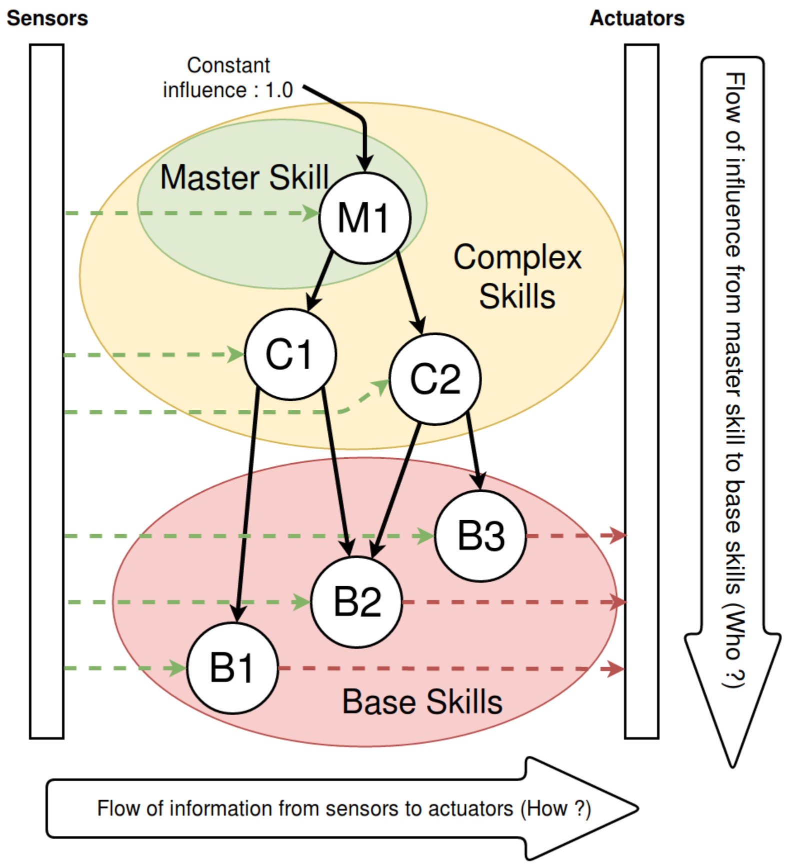


Figure 4.2: A skill hierarchy, a *master skill* influences *complex skills* which in turn influence the *base skills*.

This hierarchy of skills forms a directed acyclic graph (Figure 4.2). The influence flows along a vertical axis from the master skill down to the base skills and determines *who* (and with which magnitude) is in charge of the resulting action. The information from the sensors reaches all the skills of the hierarchy and the motor commands are output from the base skills to the actuators forming a horizontal information flow. Its purpose is to determine *how* the resulting action is going to be executed.

Figures 4.1 and 4.2 show that sensory inputs are available to every skill, including complex skills. This enables a complex skill to perform subtle coordination based on information that is not needed by the subskills.

4.2 Using Influence to determine motor commands

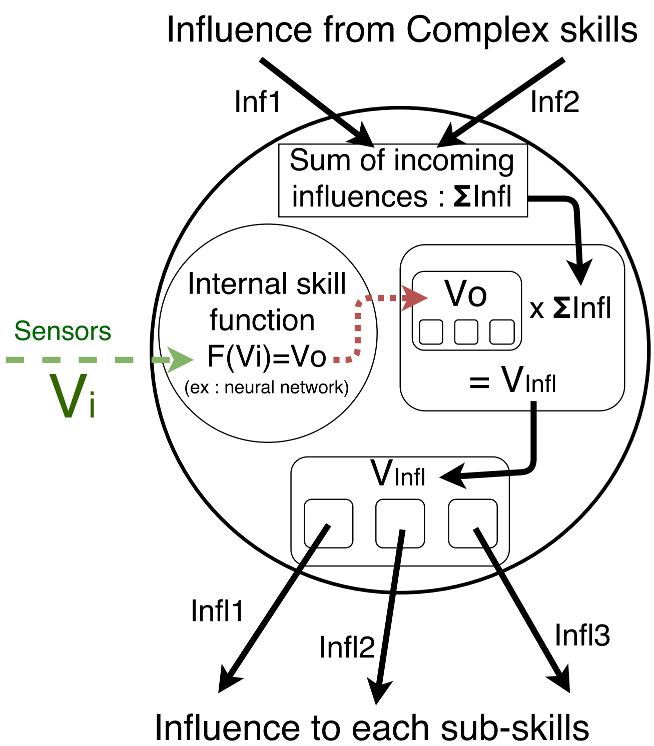


Figure 4.3: Internal architecture of a skill.

Starting from the master skill, each complex skill computes its output vector V_O and multiplies each element by the sum of the influences it received, forming the influence vector V_{Infl}, as shown in equation 4.2. The skill then sends each element $Infl_x$ of the influence vector to the corresponding subskill (figure 4.3).

$$V_{Infl} = V_O * \sum_{c\mapsto s=1}^{Cs\mapsto s} Infl_{c\mapsto s} \tag{4.2}$$

With V_{Infl} the influence vector to the subskills, $V_O = f(V_I)$ the output vector of the internal function of the skill, and $\sum_{c\mapsto s=1}^{Cs\mapsto s} Infl_{c\mapsto s}$ the sum of all influences the skill received (also noted $\Sigma Infl$).

The base skill, like any other skill, computes its output vector and multiplies each element by the sum of the influences it received, similarly to equation (4.2), forming the motor command vector $V_{Com} = [Com_1, Com_2, ..., Com_l]$. The base skill then sends each element Com_x of the motor command vector to the corresponding motor module along with the sum of the influences ($\Sigma Infl$) the base skill received.

Each motor module then computes the corresponding motor command for its actuator as a normalized weighted sum according to equation 4.3.

$$M = \frac{\sum_{b=1}^{Bs} Com_b}{\sum_{b=1}^{Bs} \Sigma Infl_b} \tag{4.3}$$

With M the resulting final motor command, b the index of the base skill that is sending a motor command, Com_b the weighted motor command for this motor module from the base skill b, $\Sigma Infl_b$ the sum of influences from the base skill b.

Equation 4.4 gives the complete computation of a motor command from the master skill to the actuator in a three level hierarchy.

$$M = \frac{\begin{array}{c}\sum_{b=1}^{Bs\mapsto M}(F_b(Vi_b) \mapsto_M * \\ \sum_{c=1}^{Cs\mapsto b}(F_c(Vi_c) \mapsto_b * \\ F_{Ms}(Vi_{Ms}) \mapsto_c *1.0))\end{array}}{\begin{array}{c}\sum_{c=1}^{Cs\mapsto b}(F_c(Vi_c) \mapsto_b * \\ (F_{Ms}(Vi_{Ms}) \mapsto_c *1.0))\end{array}} \tag{4.4}$$

With M the resulting final motor command, $Bs \mapsto_M$ the *base* skills connected to the motor module, $Cs \mapsto_b$ the *complex* skills connected to the Base skill b, F_{Ms} the internal function of the *master* skill, $F(Vi) \mapsto_X$ the element directed to X of the output vector of the skill internal function F processing the input vector Vi.

4.3 Integrating variables for internal representations

To represent internal states MIND uses *variable* modules. The goal is to provide a simple memory system as close as possible to the mind hierarchy, and subjected to the same mechanisms for regulation, access and use. The use of this memory system, the representations stored in it and the interpretation of these representations, must be learned by the MIND hierarchy as if they were part of behaviours taking place in the environment.

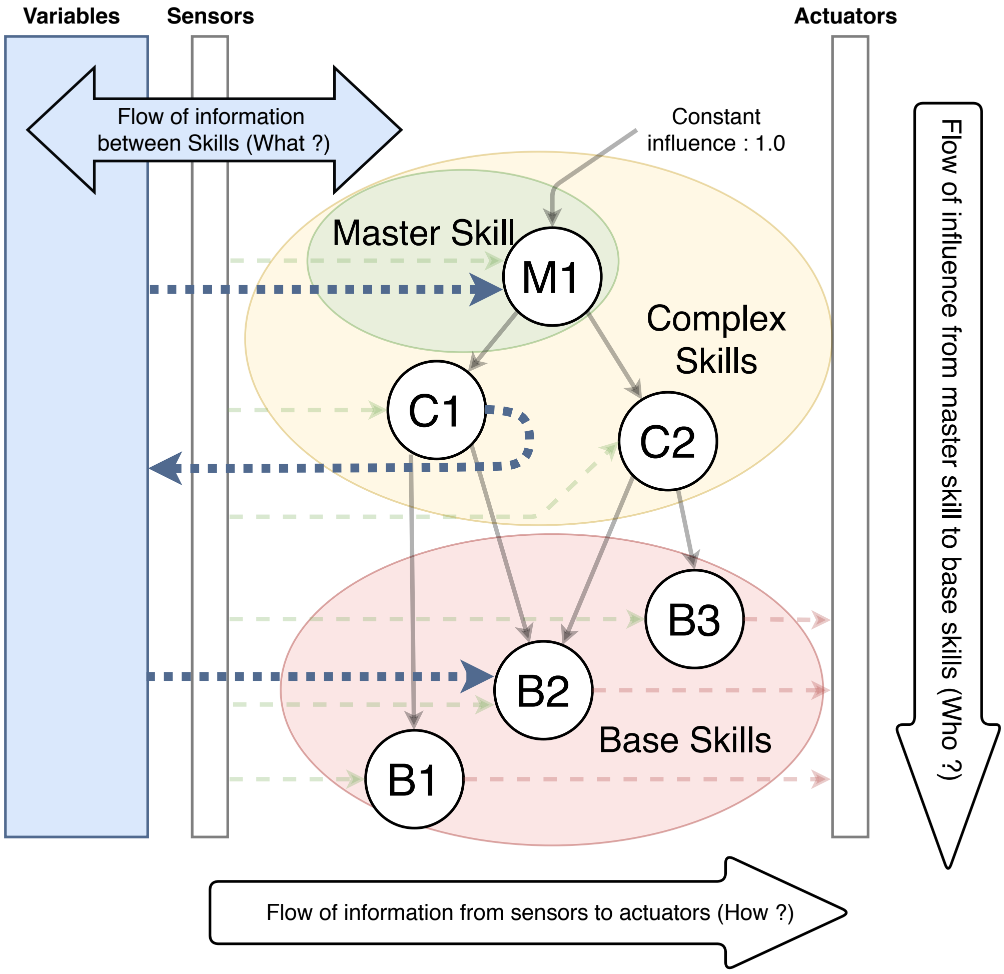


Figure 4.4: Variable integration in a MIND hierarchy

Such a memory system can be paralleled to *iconic* and *fragile memory*, up to *working memory* in humans (see section 2.4), as a way to store past observations and information in an intermediate stage of processing.

As a variable system, it can be used by skills as input or to share information and synthesize several outputs according to the influence mechanism. Providing a buffer for information shared by various skills impacts the design of hierarchies: information can be centralized and distributed among skills, its meaning identified by the variable's name making it available for combination and assuring the stability of its dedicated purpose in the same way skill modules provide identifiable behaviours.

By using these variable modules as memory or internal states, a MIND agent gains the ability to commit to a task, and represent its individual drives motivating its behaviour. As a result, MIND hierarchies can rise above purely reactive behaviours and start exhibiting low level cognitive behaviours, learned autonomously.

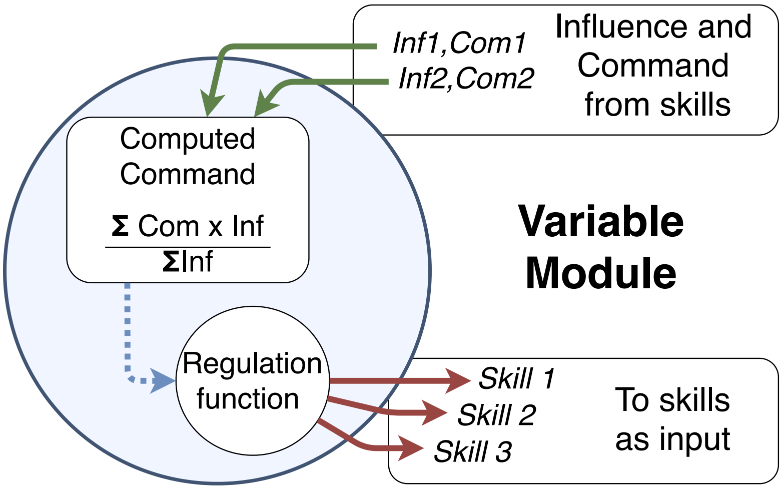


Figure 4.5: Internal architecture of a *variable* module

Variable modules provide information to skills in the same way sensor modules do. Variable modules can also receive commands from skills in the same way motor modules do. The transmission of a command to a variable module conforms to the influence mechanism, the value of the command is determined in the same manner as for a motor command, described in equation 4.3. In a manner similar to a motor module calling a process to compute the resulting motor command to be sent to the corresponding actuator, a variable module calls a regulation process. Depending on this regulation process, a variable module can serve many purposes such as memory management, counters, signal generators, internal clocks, etc.

A variable is meant to represent, memorize and identify a concept such as for instance: the orientation of a target, time, an alert level, a role, etc. Its low level representation makes it generic enough to represent many kinds of information. Like all information exchanged in the MIND architecture, it uses a signal approach and is represented by a real number, which allows for a very rich representation of the concept. In the case of motor control, this could represent a binary choice (above 0.5, light on, under 0.5 light off) or fine control of an actuator (from 0 to 1, from full speed forward to full reverse), this command would then be translated by a driver layer to the specific command codes of the actuator. In the case of the variable, skills with the ability to learn will both provide and use the information of the variable. This means that skills that write into a variable and skills that read this variable will have to agree on the meaning of its values through an emergent process, by subdividing this variable into classes whose bounds are grounded in experience. For instance, if a skill would set a variable to represent one of 3 possible roles the agent can play, it would pick 3 different values to represent these 3 roles. A skill which reads this variable to act according to its role will have to learn to classify this value into 3 different classes. It does not matter if 10 roles, 24 hours, 26 letters or 360 degrees have to be represented, the limit is placed on the precision of the skill internal function, and ultimately, on the precision of real number encoding.

Here follows a brief overview of 3 implementations of the variable module: the (simple) variable, the sin wave generator and the counter. However, these are only possible implementations, in the same way skills can accommodate many kinds of internal functions, variable modules will support any regulation process that conforms to the input/output rules of the influence mechanism.

Variable

The simple *variable* uses a linear transfer function as its regulation process. Figure 4.6 shows the evolution of the value of a simple variable connected to two skills.

Wave generator

The *wave generator* outputs a sinusoid function of time, the input value sets the frequency. Figure 4.8 shows the evolution of the value of a wave generator over time. The input value show can be result of one or multiple skills after the normalization process described in eq.4.3. Wave generators can have multiple uses, such as a measure of time or a random value generator to help agents in a reactive deadlock situation.

Counter

The *counter* outputs the current recorded count as a fraction of its maximum possible count. By default, a counter is set to a maximum of 10, its possible values are 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0. The counter is incremented on the rising edge of the input value above the *increment threshold* (by default 0.8). The counter is reset to zero on a falling edge of the input value below the *reset threshold* (by default 0.2). In case of overflow, the counter resets to zero. Figure 4.7 shows the evolution of the value of a counter over time. The input value shown can be the result of one or multiple skills after the normalization process described in eq.4.3. The first rising edge brings the value to 0.1, the second rising edge to 0.2, the following falling edge resets the value to 0.0.

Figure 4.6: A simple variable: the top graph shows the commands from 2 skills, the middle graph shows the influence from the same two skills, the bottom graph shows the resulting value of the variable over time

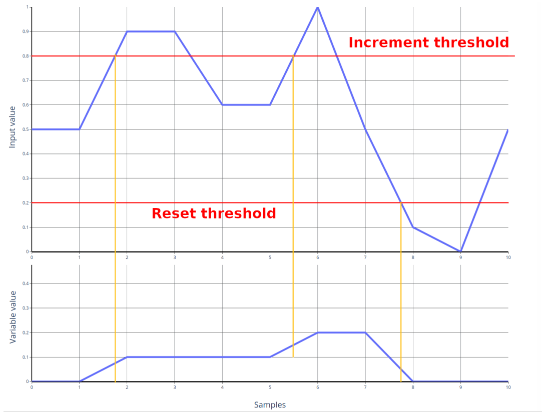


Figure 4.7: A counter: the rising edges of the input increments the counter, the falling edges resets it

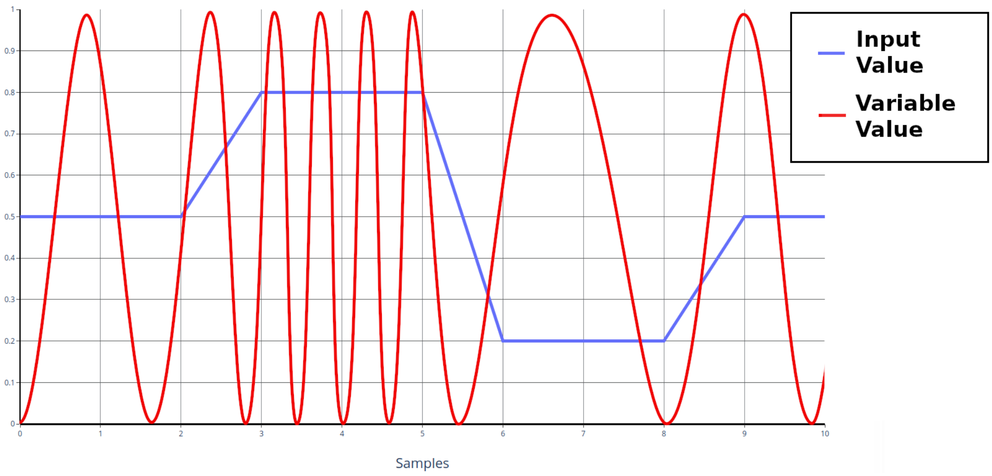


Figure 4.8: A wave generator: the input value controls the wavelength

4.4 MIND as an architecture supporting developmental agents

MIND is an architecture designed to support cumulative learning and the developmental process of intelligent agents. Its main features can be summarized as:

1. The skill modules are able to encapsulate any function controlling the behaviour of an agent. From the hierarchy's point of view, learning or non learning, approximation functions or programming procedures are equivalent.

2. The encapsulation of skills into identifiable behaviour makes them available for combination and assures their stability. Subsequent training makes use of previously acquired skills without altering them.

3. The hierarchy of modules provides the flexibility needed for developmental agents. Existing hierarchies can be built upon, sub-hierarchies can be retrained or replaced, new modules can be added to interface additional physical elements.

4. The influence mechanism unifies all modules using a connectionist approach. Modules themselves are wrappers for their internal components, allowing their integration in the system.

5. The built in memory system allows skills to store, retrieve or exchange information. Through influence, an internal state can be set to a value resulting from arbitration between several skills. Its proximity to the skills results in a strong coupling between behaviour and internal representations.

The following chapters present experiments using MIND on a complete developmental process, from early sensorimotor skill acquisition to complex social behaviours.

After presenting the implementation of MIND and the experimental context in chapter 5, chapter 6 will demonstrate its suitability to cumulative learning by building a hierarchy of reactive skills of increasing complexity, selectively improving aspects of the behaviour by retraining corresponding skills and extending an existing hierarchy with new skills and physical components.

Chapter 7 introduces the use of variables, as a means of identifying and exchanging information between skills and as a means to represent internal states necessary to rise above simple reactive behaviour.

Finally, chapter 8 presents the use of MIND in a MAS context, learning reactive coordination at first, and then with the use of variables to represent the agent's role as an internal state, learning a mechanism of social specialization.

Chapter 5

Experimental Context

To prove the effectiveness of the MIND architecture, we experimented in a simulator with a reactive agent, a (simulated) robot, using neural networks as internal functions trained by a simple genetic algorithm.

The first set of experiments consist of building a simple hierarchy to accomplish the task of collecting an object in an environment with obstacles. The next set of experiments uses variables to achieve similar goals, the last set is designed to experiment with social learning and specialization.

This section presents EvoAgents, the simulation software used in the experiments (sec. 5.1), the neural networks and genetic algorithm used and their relation to skills (sec. 5.2), and finally, an overview of the learning process (sec. 5.3) with the simulated robot's capabilities and the fine art of tuning fitness functions for genome evaluation.

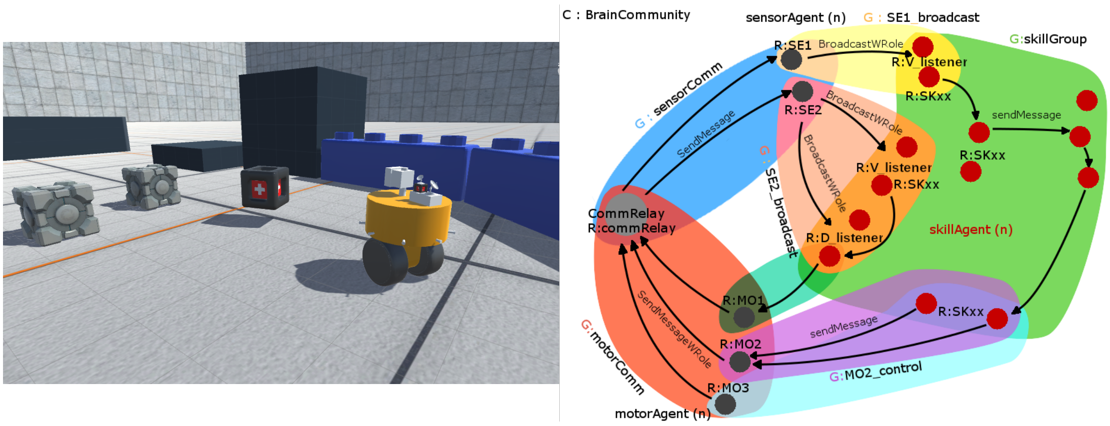


Figure 5.1: The EvoAgent Project is an ongoing project dating back to 2015 (Suro et al., 2015). On the left: an early implementation of the environment using Unity3D with 3D physics support. On the right: a dynamic version of MIND, using MaDKit 5 (Michel, 2015) agents to support modules (the diagram shows the AGR relationship (Ferber et al., 2004)). The entire architecture was built around a network socket to interface with various simulation software.

5.1 EvoAgents

EvoAgents is a custom framework written in Java designed to carry on experiments with agents using the MIND hierarchy as their control system. It includes the implementation of MIND and its different base modules, a set of learning algorithms for neural networks using the Encog library[1] for the skills internal functions, and a number of interfaces and simulation environments:

- A 2D physics environment, both single and multi agent, supported by the JBox2d physics library[2] and a viewer using Java Swing library.
- A mono-agent 3D physics environment using the JBullet physics library[3] with its 3D viewer using the JavaFx library.
- An interface for network communication through TCP or UDP to control a remote robot, external simulation environment or other programs such as video games.

EvoAgents is designed to run multi-threaded learning algorithms in headless configuration and has been used on HPC clusters.

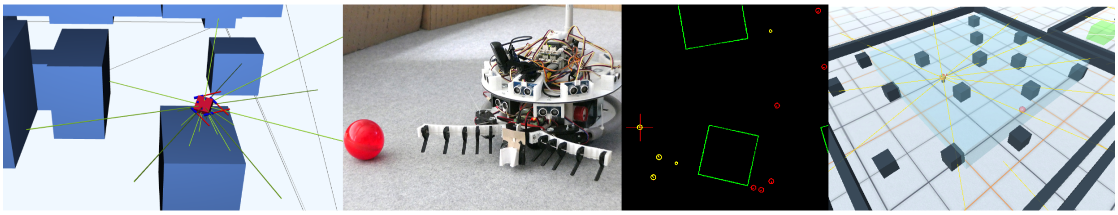


Figure 5.2: From left to right: 3D physics environment, our remote robot, multi agent environment, a remote environment in unity game engine

EvoAgents is an open source project and is available at:

https://gite.lirmm.fr/suro/evoagents2020

5.1.1 Software architecture

EvoAgentApp

EvoAgents uses task description files to set up the task to run. This file will determine the type of task (learning/demo), the environment and agent body to use, the MIND hierarchy and a number of other parameters. *Environment* and *BotBody* are Java interfaces used to define and switch between various custom environments and agent bodies. The MIND template is a folder containing skill description files, in plain text,

[1] Encog: www.heatonresearch.com/encog
[2] JBox2d: www.jbox2d.org
[3] JBullet: www.jbullet.advel.cz

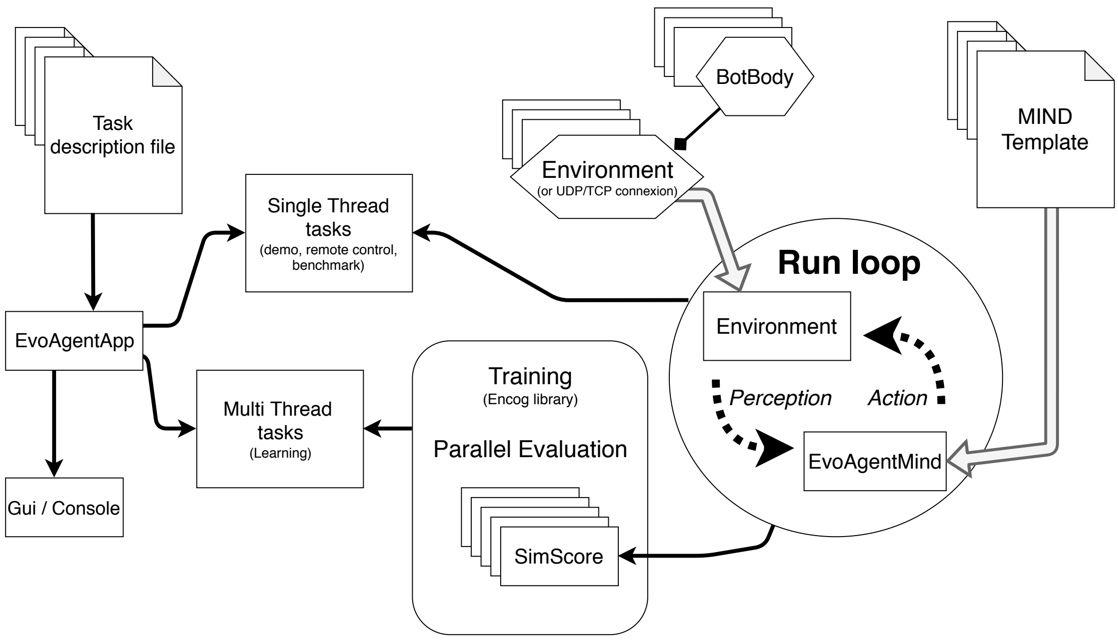


Figure 5.3: Diagram of the EvoAgents program

and associated files to set up an instance of a MIND hierarchy. The run loop is in charge of the successive exchange of perceptions and actions between the environment and the MIND hierarchy (*EvoAgentMind* class). Multiple instances of the run loop are created for multi threaded learning tasks, each one instancing a *EvoAgentMind* from the common MIND template files (preloaded).

EvoAgentMind

The *EvoAgentMind* class contains the implementation of the MIND architecture. In its current incarnation, and instance of *EvoAgentMind* is generated from a preloaded template, its structure is set and cannot be modified during execution, and the evaluation of its module is sequential. These choices favour fast execution for the learning algorithms, however, earlier implementations had the ability to start and stop modules as independent processes, with parallel processing in mind.

Figure 5.4 shows the evaluation process and the various modules involved. We can see the 3 implementations of the MIND variables (sec. 4.3) and two possible implementations of the skill modules: *NeuralNetworkModule* and *HardCodedSkillModule*. The *NeuralNetworkModule* encapsulates a neural network which can be saved and loaded from a file, several topologies are compatible (using the *MLRegression* interface). The *HardCodedSkillModule* encapsulates a Java class which is loaded dynamically (using the *HardCodedSkill* interface).

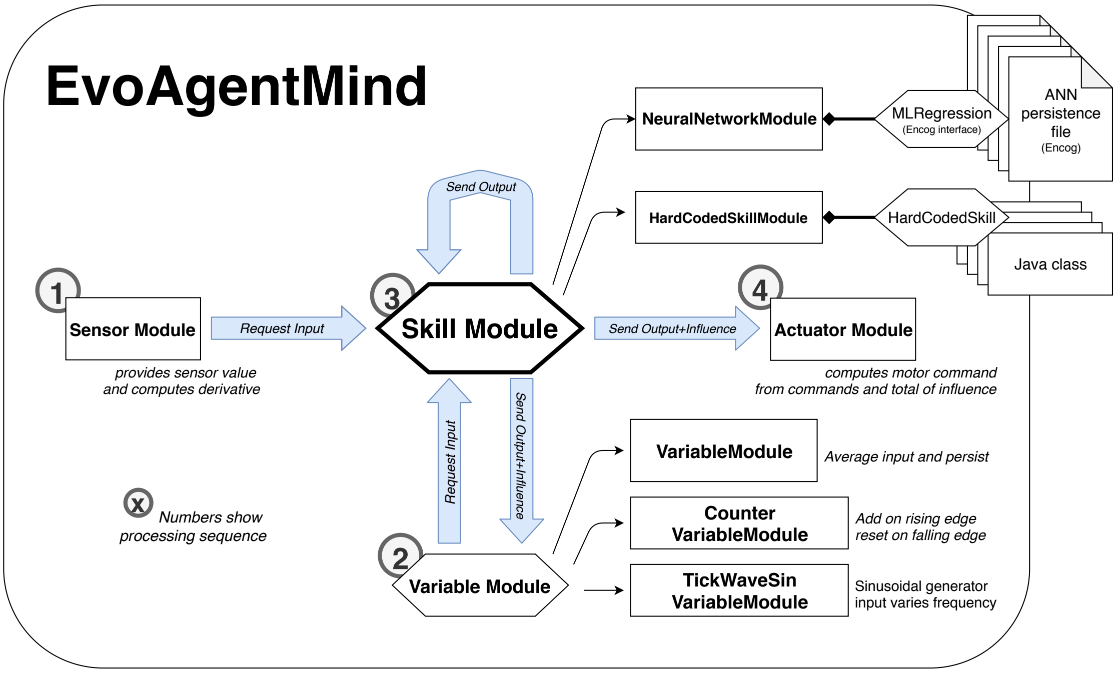


Figure 5.4: Diagram of the EvoAgentMind component

The evaluation process works as follows:

1. All *sensor* modules are evaluated: the input of the corresponding sensor is converted to bounded values ([0,1] range), past values are recorded and a derivative is calculated.

2. All *variable* modules are evaluated: the skill commands are processed, the regulation function is applied, past values are recorded and a derivative is calculated.

3. All *skill* modules are evaluated, starting from the higher levels down to the base skills. Each skill retrieves its input from the *sensor* and *variable* modules, applies its internal function, and sends its output to lower level skills and/or *actuator* modules. Commands sent to *variable* modules are kept by the *variable* modules to be evaluated at the next cycle.

4. All *actuator* modules are evaluated: the skill commands are processed and the bounded value ([0,1] range) is converted to a command relevant to the actuator.

5.1.2 Defining a MIND hierarchy

An agent is defined by a folder containing files to describe sensors, actuators and variables and folders containing the skills and tasks files.

Sensors and actuators

A file with the *.botdesc* extension describes the sensors and actuators available to the MIND hierarchy. The names of the sensors and actuators listed are the ones used by the skill files. Optional parameters can be specified, such as type information or ranges and dead zones.

(see appendix A.1).

Variables

A file with the *.vardesc* extension describes the variables available to the MIND hierarchy. The names of the variables are the ones used by the skill files. The type of variable must be specified:

- VariableModule (simple memory)
- TickWaveSinVariableModule (a clock)
- CounterVariableModule (a counter, increments on a rising edge)

For more details, see appendix A.2.

Skills

A MIND hierarchy is a collection of skill modules organized in a network, therefore, only skills have to be defined.

A skill is defined by a folder containing a skill description file using the *.ades* extension, and files associated with the internal function of the skill (neural network persistence file).

The skill description file lists the inputs (sensors and variables), the outputs (actuators, variables and sub skills) and the type of internal function to use.

For more details and examples, see appendix A.3.

Tasks

The task folder contains the task files describing experiments to run. Task files use the extension *.simbatch*. Parameters include:

- the type of task to run (demo/learning 2D/3D/remote)
- the type of agent to use
- the environment to use
- the name of the master skill
- a drive module to use
- parameters for learning, such as the algorithms to use, the name of the skill to train or the time limit

For more details and examples, see appendix A.4.

5.1.3 The drive module

The master skill of a fully developed MIND agent would be able, in theory, to define its own goals and fulfil all of its needs, and continue its development *for itself*. In practice, we want control over what the agent is trying to learn or accomplish.

The drive module is an entry point in the deliberative process of the agent, with access to all the MIND modules available to the agent. The drive module lets you define a Java procedure called each time the hierarchy is evaluated, after the inputs are updated (Fig. 5.4 step 1 and 2) but before the skills are evaluated (Fig. 5.4 step 3).

This procedure can be used to set the value of a variable, add a concurrent command to an actuator or send influences to skills (including skills not present in the hierarchy).

One of the applications of the drive module was the learning a *GoToTarget* behaviour, where the skill uses a variable to represent its target. In practical use this variable is set by a higher level skill, but when learning *GoToTarget* for the first time, *GoToTarget* is the master skill.

To solve this problem, early versions of the program used a programmed master skill setting the variable and calling *GoToTarget*. However, as this problem became more common, and scaffolding master skill impractical, the drive module was added offering a more generic solution and defining this specific control behaviour at a task level instead of altering the skill hierarchy.

(see appendix A.5.4 for more details and examples).

5.1.4 Defining simulation elements (Java programming)

The software architecture of the EvoAgents app is designed to facilitate the definition of experimental environments.

A new agent body can be created by extending the *BotBody* abstract class, and adding existing sensors and actuators or adding custom ones by extending the *Sensor* and *Actuator* classes.

In the same way, new environments are created by extending *SimulationEnvironment* classes (2D or 3D) and adding mechanics and world elements.

Finally, classes for *Reward functions* and *Control functions* are provided to define the reward necessary for the learning process and various interruption conditions.

See appendix A.5 for more details and examples.

5.1.5 Simulation viewers

EvoAgents provides viewers for the simulation. 2D and 3D views are available, as well as a MIND hierarchy viewer which shows the current state of the MIND hierarchy and its sensors, actuators and variables.

Figure 5.5 shows the 2D simulation environment used in the experiments presented in this thesis. The red circle represents our robot, green squares and borders represent obstacles. The long white lines connecting the robot to the targets represent the orientation sensors. The shorter white lines represent the obstacle detection sensors. A full

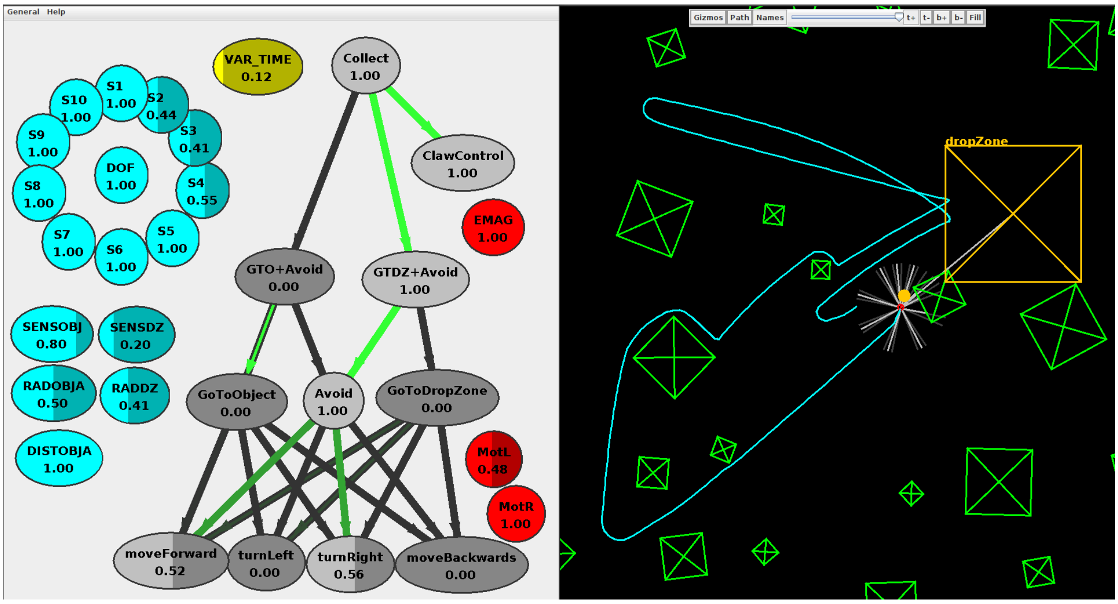


Figure 5.5: On the left: the state of the MIND hierarchy and its sensors, actuators and variables, on the right: a 2D view of the simulation.

line means that the sensor has not collided with an obstacle within its range. A shorter line means the sensor has detected an obstacle and will give its distance as a fraction of the maximum range. Other information might be represented in blue such as signals or in this case the path of the agent.

The hierarchy viewer shows the skills in grey with their current influence displayed under the name. The influence links are represented by lines connecting the skills. The colour gradient of the link represents the influence transmitted: from black for the value 0 to green for the value 1. The central line represents the command of the skill alone and the outer line represents the influence actually transmitted (see Fig.4.3 and Eq.4.2). Other ellipsis represent the input and outputs: blue for sensors, yellow for variables and red for actuators.

5.2 Skill internal function and the learning algorithm

Unless otherwise specified, all skill internal functions used are neural networks, using a genetic algorithm *as* a function optimizer (De Jong (1992), Rudolph (1994)). Some skills performing trivial tasks have been given non-learning programming procedures as internal functions. We also experimented with special network topologies and learning algorithms.

5.2.1 Initial skill internal functions

Using neural networks as controllers for agent behaviour is a well established practice (Huang et al., 2005; Lessin et al., 2013, 2015; Levine and Abbeel, 2014; Pérez et al., 2017; Devin et al., 2017), and many topologies can be used. We chose to implement the internal function as a simple multi-layered perceptron, for which the input layer corresponds to the input vector of the skill and the output layer to its output vector. Depending on the skill, its neural network will use 2 or 3 hidden layers of N_{HIDDEN} neurons, with:

$$N_{HIDDEN} = Max(N_{Vi}, N_{Vo}) + 2 \tag{5.1}$$

N_{Vi} the number of neurons on the input layer

N_{Vo} the number of neurons on the output layer

As the internal functions are intended to be neurocontrollers with direct control on the output (Pérez et al., 2017; Devin et al., 2017) rather than classifiers with action selection (Huang et al., 2005), we found it more suited to set the last layer of the neural network to a linear transfer function, whose output is clamped between 0 and 1.

We acknowledge that this configuration has a high convergence time for the genetic algorithm we use but it is generic enough to cover all kinds of skills.

Unlike monolithic approaches which use a unique skill with a single neural network that connects all sensors and actuators, in our hierarchical approach, each skill has its own neural network using only the appropriate actuators, sensors or subskills.

5.2.2 Other skill internal functions

Our initial experiments were designed to prove the ability of MIND to form a functional reactive hierarchy (see chapter 6), with minimal *a priori* on the tasks each skill has to perform. To this end we chose the most general network topology for our internal functions, i.e. a fully connected network with an excess of neurons in the hidden layers. Unnecessary connections between neurons are effectively disconnected by the learning process when their weights are set to 0.

However, in subsequent experiments we implemented other topologies in an effort to reduce the cost in resources. Skills with numerous inputs would require relatively large networks (see eq. 5.1).

As was done in other works (Levine and Abbeel, 2014; Devin et al., 2017), we experimented with a convolutional neural network (CNN) topology, using both a spatial and temporal convolution layer. We used this topology on skills using the obstacle sensors, an array of 10 proximity sensors, which are (relatively) numerous and have a high degree of correlation (when the robot moves, an obstacle detected by one sensor is likely to be detected by its neighbours in the near future).

The Temporal kernel takes as input three states of a single sensor: its current value, and its past values at 5 and 10 sampling back (ticks). We included this kernel as an alternative to using RNN (Pérez et al., 2017) to deal with time series.

The Spatial kernel takes as input three values: the output of the temporal convolution layer corresponding to the sensor and its two neighbours (right and left).

The output of the spatial kernel, along with other sensor inputs not involved in convolution, is then fed to a standard multi-layer perceptron as described above.

Such a setup greatly reduces the size of the network, a kernel only connects a subset of the input. For instance, the spatial kernel connects the sensor and its two neighbours only, costing 3 weight values to tune. The same kernel is used on all the inputs of convolution layer which has the advantage of keeping the number of weight values to tune constant to the number of inputs. Another benefit is that the experience of all sensors is integrated in the same kernel, for instance, the relation between the episodes of a temporal sequence of values of a proximity sensor is the same no matter which sensor is used (in simple terms, is an object getting closer or moving away?).

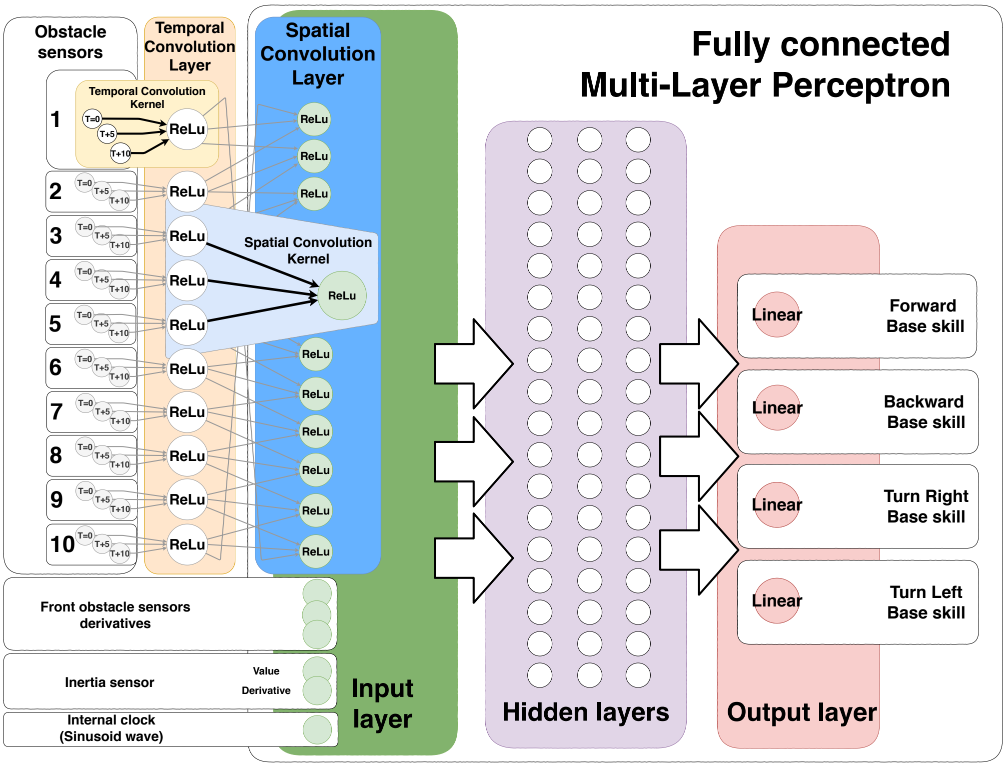


Figure 5.6: The Convolutional Neural Network used on the later iterations of the *Avoid* skill

5.2.3 Learning algorithm

To learn the tasks, we used a simple genetic algorithm. We chose this solution for its simplicity, exploratory properties and good performance with delayed rewards. By nature, there is no need to provide it with a description of how to achieve a goal (unlike methods which use backpropagation that requires an input-output training set) but only with a way to measure if the performance of an individual is better or worse than the performance of other individuals. Unlike other reinforcement algorithms where the reward signal modifies the behaviour, the quality of the behaviour is judged at the end of the life cycle of the individual. This allows the individual to get negative rewards if it will lead to a superior positive reward later on, thus circumventing the issue of delayed reward.

One of the major drawbacks of the genetic algorithm is its high computational cost, specifically the evaluation of the fitness function which requires running each individual, i.e. agents, in a simulated environment for a sufficient period of time. However, as the evaluation of each agent is completely independent, it can be run in parallel. This allows for the use of high performance computing solutions, bringing the real time evaluation of an entire generation down to the evaluation time of a single agent.

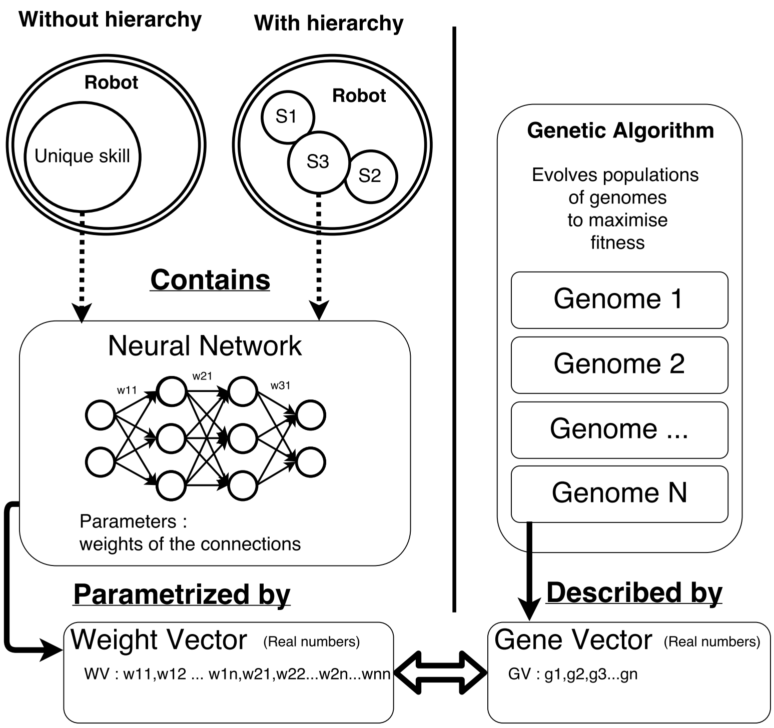


Figure 5.7: Relation between the skills and the genetic algorithm

In order to use a genetic algorithm on a neural network, the weights of the neural network's connections are ordered in a vector, which corresponds to the genome of the individual from the point of view of the genetic algorithm. The genome will be bred and mutated in accordance with the fitness function the environment provides as feedback. The relation between the skills and the genetic algorithm is illustrated in Figure 5.7.

The genetic algorithm works as follows (Russell and Norvig, 2009). An initial population is generated randomly (Alg.1 line 1). Each individual is evaluated in an environment

Algorithm 1: Genetic algorithm

Data:
current population: Pop_c
selected population: Pop_s
new population: Pop_n
population size: $PopSize$
iteration limit: LIM
all time best individual: $BestG$
Result:
$Pop_c \leftarrow generateRandomGenomes()$
while LIM *not reached* **do**
    $Pop_s \leftarrow \emptyset$
    $Pop_n \leftarrow \emptyset$
    foreach *genome* G *in* Pop_c **do**
        $evaluateFitness(G)$
    end
    $sortByHighestFitness(Pop_c)$
    if $getBest(Pop_c) > BestG$ **then**
        $BestG \leftarrow getBest(Pop_c)$
    end
    $Pop_s \leftarrow top30\%(Pop_c)$
    while $Pop_n \leq PopSize$ **do**
        $Pop_n \leftarrow Pop_n + crossbreed(Pop_s)$
    end
    $Pop_c \leftarrow Pop_n$
end
return $BestG$

Algorithm 2: CrossBreeding algorithm

Data:
input population: Pop_{in}
parent genome A: G_A
parent genome B: G_B
new genome: G_{New}
genome size: $GSize$
Result:
$G_A \leftarrow pickRandomGenome(Pop_{in})$
$G_B \leftarrow pickRandomGenome(Pop_{in})$
$G_{New} \leftarrow \emptyset$
$pivot \leftarrow random(1, GSize)$
while $iterator \leq GSize$ **do**
  if $iterator \leq pivot$ **then**
    $G_{New} \leftarrow G_{New} + getGene(iterator, G_A)$
  else
    $G_{New} \leftarrow G_{New} + getGene(iterator, G_B)$
  end
end
return G_{New}

related to the task to learn and is given a score by the fitness function (or reward function) of the environment (Alg.1 line 6). The individuals with the best scores are selected, using a simple truncation method (Alg.1 line 7 and 8) and their genomes mixed and mutated to generate a new population to be evaluated (Alg.1 line 10). The process is repeated until the given number of generations (LIM) is reached and ends by returning the best individual of all generations (De Jong, 1992).

Other learning techniques

Over the course of this thesis we improved the implementation of the basic genetic algorithm using, for instance, *tournament selection* instead of *truncation*. We also experimented with other learning algorithms, such as the NEAT algorithm (Stanley and Miikkulainen, 2002). NEAT, standing for NeuroEvolution of Augmenting Topologies, goes a step further than evolving the weights of the network, and also evolves the topology of the network, progressively adding neurons to fit the complexity of the function to represent. While it removes the need to define the topology of the network (and the use of tricks such as Eq.5.1), it adds significantly to the cost of learning. We found out it did fairly well on very simple tasks, by learning the simplest behaviour, however it tends to get stuck in local minimum when the topology should increase in complexity.

We also developed a technique of Neural imprinting, a simple technique based on Guided Policy Search (Levine et al., 2015b,a; Levine and Abbeel, 2014). This technique allows us to transfer various kinds of functions controlling the behaviour of the agent into a neural network, such as programmed functions, networks with different topologies or human operated behaviours. It can be seen as a "hand-over-hand" physical prompting (Hersen, 2005). This technique works as follows:

- The agent is placed in the environment and allowed to run using the source function.
- The inputs and outputs are recorded to form a training set for the resulting neural network.
- The neural network is trained with the training set and evaluated in the same environment.
- Repeat the process until the performance of the trained neural network is satisfactory.

Neural imprinting gave good results, with the constraint of needing a demonstrator. It can be a valuable tool for learning through imitation. Another possible use of neural imprinting, inspired by the prompt hierarchy in physical education (Winnick and Porretta, 2016), is the creation of an initial population for the genetic algorithm, using sub-optimal demonstrators. This initial population will be optimized by the genetic algorithm, speeding up convergence by narrowing the search around a set of sub-optimal but acceptable behaviours.

5.3 Learning process

To set up the experimental scenarios of the following chapters we have to define the capabilities of our agent, a simulated robot, and for each task, a set of reward functions to be used in the genome evaluation process. This section provides an overview of these elements, the specifics of each scenario will be detailed in the following chapters.

5.3.1 The simulated robot

The simulated robot is composed of two motors, one for each wheel, and a claw to grab the target object. It also has 18 sensors:

- 10 obstacle sensors (range finders) placed around the robot.
- 3 sensors giving respectively the orientation of the target object, the drop zone and the power supply (a zone to recharge the batteries).
- A sensor indicating if the target object is in range of the claw.
- 2 sensors indicating respectively if the robot is inside the drop zone and inside the power supply zone.
- A sensor indicating the remaining charge of the batteries.
- A simple inertial sensor indicating if any movement occurred.

Depending on the needs of the experiments the robot can also use:

- Sensors giving the orientation and distance of a different kind of object.
- Sensors giving the orientation and distance of a specific trigger zone.
- A sensor informing the robot it received a positive or negative reward.
- A constant sensor representing a unique ID of a robot within a group.
- Sensors giving the orientation and distance of the nearest robot of the same team.
- Sensors giving the orientation and distance of a different kind signals emitted by other robots.
- Multiple signal emitters.

The robot's software interfaces with the MIND implementation through what we call a *Sensor Module*. This module is able to compute the derivative and past values of any sensor input and provide the result as a virtual sensor.

The *Variable Modules* of MIND can be used to generate a sinusoidal wave, which are useful, in conjunction with the inertial sensor, to solve deadlock situations.

The motor command, issued by the *Motor Module* to its actuator as a real number between 0 and 1, is interpreted as a percentage of the actuator's capability (either power, speed, angular position, etc.). In the case of the wheels, 1 corresponds to full speed forward, 0 to full speed backwards and 0.5 to stopped. In the case of the claw, the number corresponds to its position (treated as binary): above 0.5 is closed, under 0.5 is opened.

5.3.2 Genome evaluation

To evolve a population to fit the desired behaviour, each genome of a generation must be evaluated and given a fitness score.

The genome to be evaluated sets the weights of a neural network using its gene vector (Fig. 5.7). The neural network is then placed inside the skill module corresponding to the task to learn. Finally, this skill and its instance of the MIND hierarchy take control of a simulated robot placed in an environment where it can run for a given time.

Figure 5.8 shows the environments for the *GoToObject*, *GoToDropZone* and *Avoid* tasks. As described in subsection 5.1.5, the red circle represents our robot, the green squares and borders represent obstacles. The different lines represent sensor information such as object orientation or obstacle sensors.

Notice the difference in size between the object and the drop zone, grabbing the object will require finer motor control, the target being smaller. Also, grabbing the object requires to align the front of the robot (where the claw is mounted) with the object.

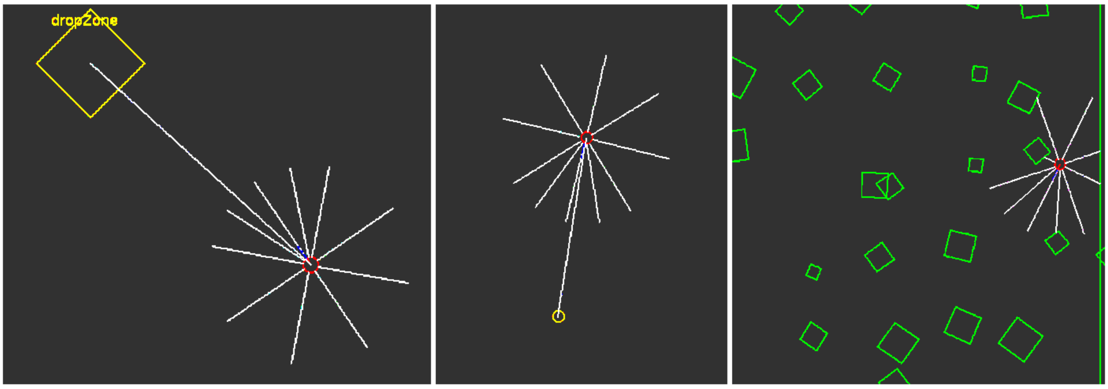


Figure 5.8: *GoToDropZone*, *GoToObject* and *Avoid* environments

5.3.3 Reward functions

At each tick of the simulation, a set of reward functions are evaluated and their result added to the current score. At the end of the evaluation, this accumulated score will represent the fitness score of the evaluated genome.

For instance, the *GoToObject* task has two reward functions:

1. *Closing on target*: $-\Delta_{DistanceTarget} * 0.001$, a very small reward given each tick that can be negative if the distance to the target increases.

2. *Reaching target*: $+0.5$, a large reward given each time the robot reaches the target. The target is then moved to a new place for the robot to reach (random placement with minimum distance constraint).

The small *Closing on target* reward helps to differentiate genomes that result in an attraction behaviour towards the target from the genomes that cause repulsion from the target. This helps to speed up the early stages of evolution. However, the reward remains small compared to the *Reaching target* reward to favour the outcome we desire over the process we think would lead to that outcome. It also helps in optimizing the final stages of evolution, favouring genomes that result in a sharper turn when facing the target.

Indirectly the *Reaching target* reward combined with the time limit also leads to optimization. Genomes that reach more targets within the time limit will have better scores. This leads to sharper turns, straight trajectories and moving at top speed.

The *GoToObject* task is a simple example, and yet we can already see subtle interactions between all the constraints and rewards. When designing the *Avoid* task we found out for ourselves what was meant in the Robot Shaping experiments:

> *This process is iterative, in that difficulties in finding, say, an appropriate shaping policy may compel us to backtrack and modify previous design decisions.*

Dorigo and Colombetti (1994)

> *[...] the designer needs to write the RP [Reinforcement Program], which could at first sight appear as difficult as directly coding the control program.*
>
> Dorigo and Colombetti (1998)

In complex behaviour where it is difficult to even picture an optimal solution out of many acceptable ones, learning from a hand crafted lesson (or reinforcement program) will quickly show its limits. Scenario 2 (sec. 6.2) will show indeed that there is no need to tune the reward functions to get the optimal behaviour, optimization can be achieved by simpler means.

Chapter 6

MIND Hierarchies

This chapter presents a series of experiments using the basic principles of MIND to demonstrate its suitability to cumulative learning. We show the ability of MIND to learn reactive behaviours, low level sensorimotor skills and complex coordination skills, and illustrate the effect of *influence* on behaviour.

We build coordination skills by making use of previously acquired skills without altering them. When the role of each skill is established through learning, we identify which aspect of the behaviour could be improved and focus the training on the corresponding skill.

Based on the hierarchy we established, we redefine the target behaviour by adding new constraints and sensors. We show that MIND provides the flexibility needed for continuous development enabling the agent to extend its functionality beyond its original purpose.

6.1 Scenario 1: Curriculum learning

6.1.1 Building a MIND hierarchy: Collect

In this first scenario, we aim to demonstrate that the MIND architecture is able to organize simple skills into complex behaviours, and is able to do this reliably even when using a simple genetic learning process.

6.1.2 Protocol

In this scenario the goal is to teach the robot a collecting task which consists in picking up an object and bringing it back to a drop zone without colliding with obstacles.

Each of the six skills needed to accomplish this task will be trained in 10 independent attempts. This will give us a fair sample of possible outcomes and convergence times, and allow us to draw conclusions in spite of the stochastic nature of genetic algorithms (Gen and Lin, 2007; Rudolph, 1994).

The scores given by the different sets of reward functions are used by the selection process of the genetic algorithm as a relative measure of performance. The result section will provide a benchmark for the final behaviour, however a complete evaluation and interpretation of such complex behaviours requires observation. Videos of the results are available at the following address:

Base skills :https://hal.archives-ouvertes.fr/hal-02572019v1

Complex skills :https://hal.archives-ouvertes.fr/hal-02572031v1

Curriculum and the MIND hierarchy

To create an initial hierarchy there are many possible ways to organize the different subskills, all of which are valid as long as they are able to learn from the curriculum.

We choose to divide the complex *Collect* task into a curriculum of five subtasks, three of which are base tasks to be learned by the corresponding base skills:

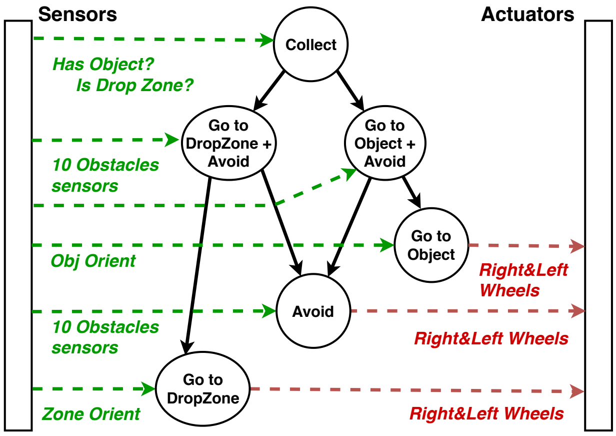


Figure 6.1: The complete hierarchy for the initial collection task, with sensor and motor information shown.

1. *GoToObject*: Going to the object in an empty environment. The agent receives a small reward for approaching the object and a large reward for reaching the object.

2. *GoToDropZone*: Going to the drop zone in an empty environment. The agent receives a small reward for approaching the drop zone and a large reward for reaching it.

3. *Avoid*: Avoiding collision while moving in an environment with obstacles. A collision ends the evaluation and gives the agent a large negative reward. A number of positive rewards are given (in order of importance): visiting new areas of the environment, keeping a straight path, keeping distance with obstacles and travelling forward.

On these base tasks we build two higher level complex tasks:

1. *GoToObject+Avoid*: going to the object while avoiding collision in an environment with obstacles. A collision ends the evaluation and a positive reward is given each time the agent reaches the object.

2. *GoToDropZone+Avoid*: going to the drop zone while avoiding collision in an environment with obstacles. A collision ends the evaluation, a positive reward is given each time the agent reaches the zone.

Finally, *Collect* is learned in the final environment containing an object, a drop zone and obstacles. A collision ends the evaluation, a positive reward is given for each object collected (picked up and brought to the drop zone).

Learning *Collect*, the largest complex task, is an interesting challenge. Its subtasks, *GoToObject* and *GoToDropZone*, require to use the motor functions in a completely opposite way and have to be performed sequentially and exclusively. Conversely, the *Avoid* task would be best used as a composition of vectors, by altering the set trajectory to smoothly avoid an obstacle.

6.1.3 Results

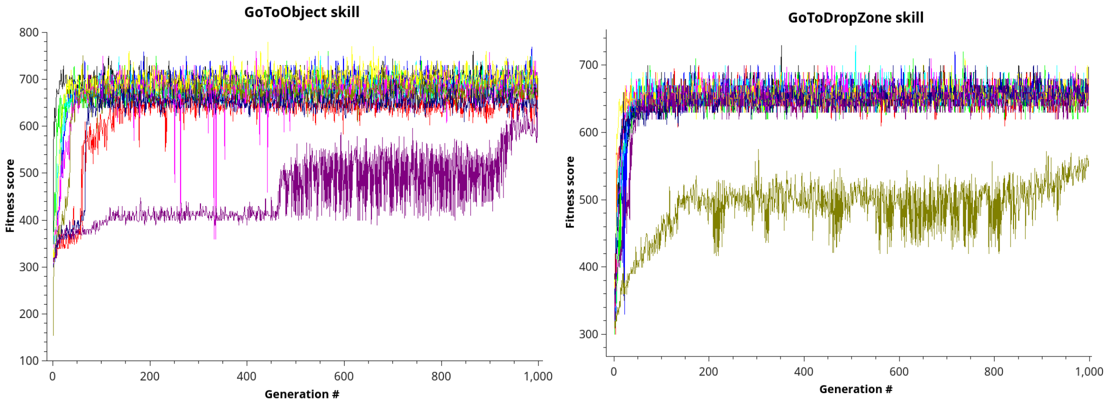


Figure 6.2: *GoToObject* and *GoToDropZone* base skills: best individual score for each generation, 10 separate attempts over 1000 generations.

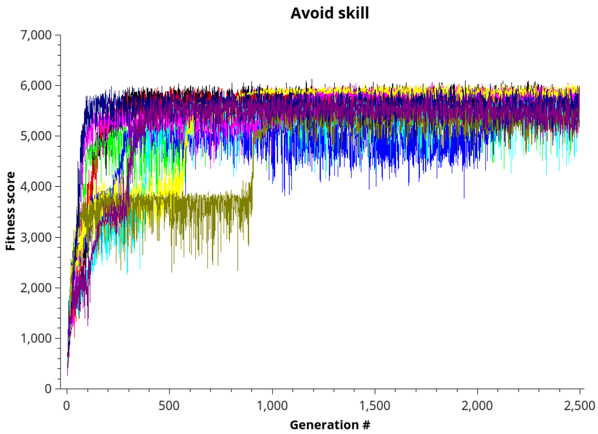


Figure 6.3: *Avoid* base skill: best individual score for each generation, 10 separate attempts over 2500 generations.

Figures 6.2 and 6.3 show that most attempts of the *GoToDropZone* and *GoToObject* skills reach their maximum value in 100 generations, whereas most attempts of the *Avoid* skill continue to evolve past 500 generations (the figure 6.3 shows the attempt over 2500 generations which we will discuss in section 6.2). This difference in convergence time is coherent with the complexity of the networks to train, *Avoid* has more than 10 inputs while the other base skills only 1.

We see that in both *GoToObject* and *GoToDropZone* skills, one attempt in ten did not reach a satisfactory result within the number of generations given. However, it is interesting to note that they are still improving with each generation, which is positive in the context of our work aimed at supporting open-ended and lifelong development of artificial agents. Setting aside the failed attempts, we selected successful base skills to learn the complex skills *GoToObject +Avoid* and *GoToDropZone +Avoid*. The *GoToObject +Avoid* is slightly more difficult to learn than *GoToDropZone +Avoid*, which can be explained by the fact that the object is smaller than the drop zone and requires more precision (the "claw" must be aligned with the object to grab it, see Sec. 5.3). Both skills still succeed on each of their respective 10 attempts within the given number of generations.

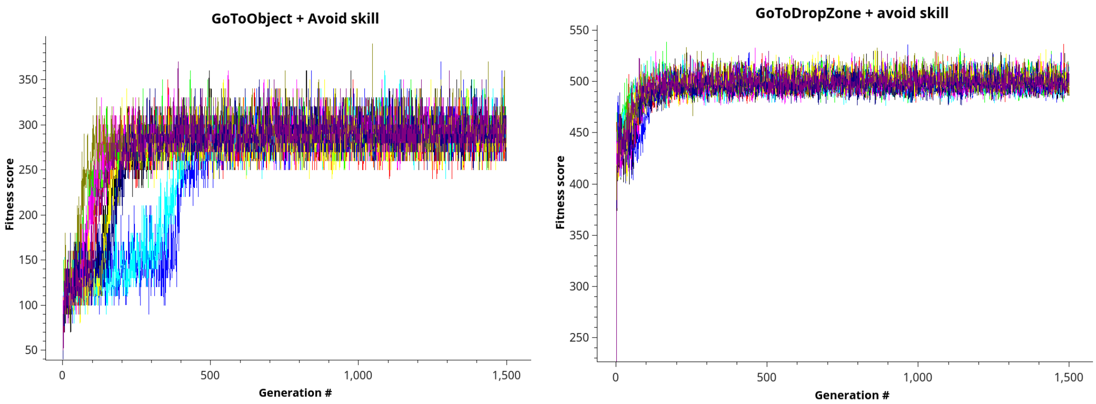


Figure 6.4: *GoToObject+Avoid* and *GoToDropZone+Avoid* complex skills: best individual score for each generation, 10 separate attempts over 1500 generations.

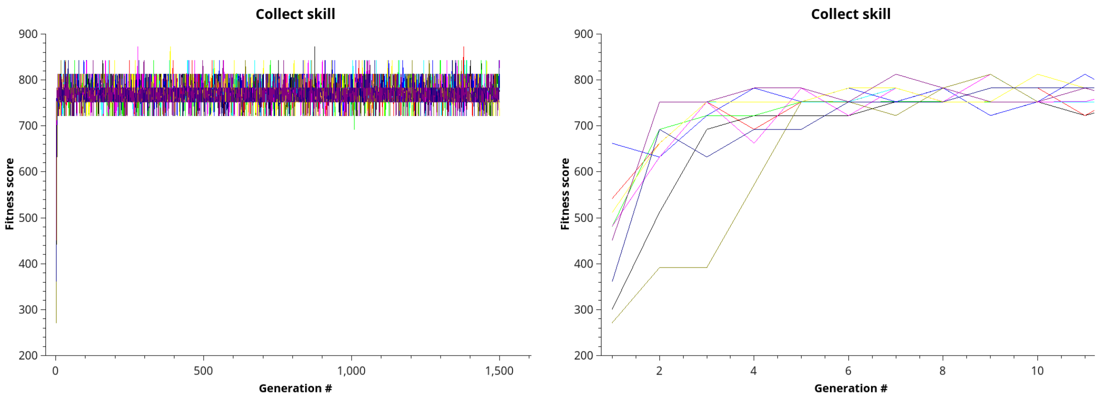


Figure 6.5: *Collect* master skill: best individual score for each generation, 10 separate attempts over 1500 generations. (Right: showing only the first 10 generations scores).

Finally, the master skill *Collect*, having a very simple network to train, succeeds on each of its 10 attempts in under 10 generations.

To illustrate the effect of influence we trained a similar hierarchy using a set of 4 constant base skills: *moveForward* always sets both wheels to full forward, *moveBackwards* always sets both wheels to full backwards, *turnLeft* and *turnRight* set one wheel to full forward and one wheel to full backwards. Using these base skills with the influence mechanism it is possible to obtain any combination of values for the wheels. Figure 6.6 presents our agent in a situation where it must avoid an obstacle to reach its goal. The right panel shows the environment and the left panel shows the state of the hierarchy. As described in subsection 5.1.5, the influence sum of each skill is displayed under its name, the influence links are represented by a gradient from black to green, for the 0 and 1 values respectively. The following describes the sequence shown in figure 6.6:

Step 1: Our agent is approaching an object to collect. The obstacle is not close enough to be considered a collision risk by *GoToObject+Avoid* thus it transmits its influence to *GoToObject* exclusively (1.0 x 1.0). *GoToObject* transmits an influence of 0.11 to *moveForward*. Since no other constant skill receives any influence, *moveForward* completely controls the output (both wheels are set to full forward).

Step 2: The agent came too close to the obstacle, *GoToObject+Avoid* transmits its influence to *Avoid* exclusively. *Avoid* transmits an influence of 1.0 to *moveForward* and *turnLeft*. In this simple case, both skills transmit a motor command of full forward to the right wheel, the commands being identical, their average result is full forward. In the case of the left wheel, *moveForward* transmits a motor command of full forward with the influence of 1.0 (weighted motor command: 1.0 x 1.0), *turnLeft* transmits a motor command of full backwards with the influence of 1.0 (weighted motor command: 0.0 x 1.0). According to the equation 4.3, the resulting motor command for the left wheel is:

$$\frac{(1.0 \times 1.0) + (0.0 \times 1.0)}{1.0 + 1.0} = 0.5 \tag{6.1}$$

0.5 corresponds to stop (being halfway between full forward and full reverse). With the right wheel full forward and the left wheel stopped, the resulting behaviour is a sharp left turn centred on the left wheel.

Step 3: As the agent recovers some distance with the obstacle, the influence to *Avoid* decreases while the influence to *GoToObject* increases. *GoToObject* transmits a high influence (1.0) to *turnRight* to make a sharp turn right to reach the object. But since *GoToObject* only has a relatively small influence (0.16) *turnRight* can at most receive as much as *GoToObject* (0.16 x 1.0). Since *moveForward* receives a much higher influence, the result is a trajectory slightly curved to the right.

Step 4: With the forward obstacle sensors clear of the obstacle, *Avoid* lowers its influence to *moveForward* which increases relatively the impact of *turnRight* on the wheels (even if the influence to *turnRight* decreased). The slight curve is now a deliberate turn around the obstacle.

Step 5: Eventually, our agent picks up the object. *Collect* sends its influence exclusively to *GoToDropZone+Avoid*. *GoToObject+Avoid* is still transmitting influence to its subskills, but since its own influence is 0.0, it has no effect. It is interesting to note that *GoToDropZone+Avoid* chooses to keep *Avoid* active at a distance from the obstacle where *GoToObject+Avoid* would have it inactive. This can be explained by the fact that *GoToObject+Avoid* may have to approach an obstacle in order to collect an object, while *GoToDropZone+Avoid* is more at risk of colliding with an obstacle when carrying the object (due to the object protruding in front).

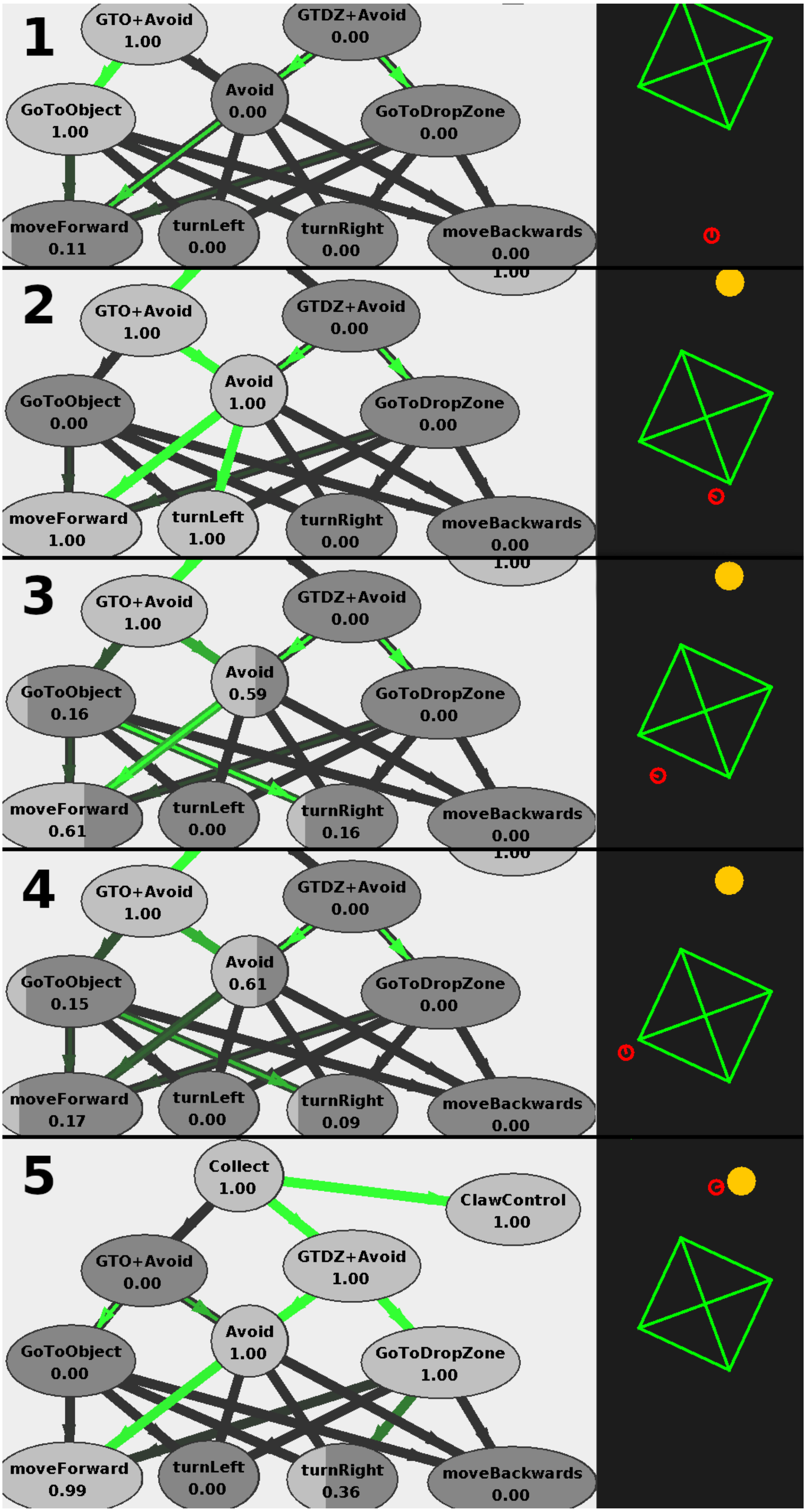


Figure 6.6: Five steps of avoiding an obstacle and reaching a goal using influence.

A videos of this behaviour is available at the following address:
https://hal.archives-ouvertes.fr/hal-02572034v1

6.1.4 Analysis

Our initial MIND hierarchy succeeded in quickly learning the curriculum we designed, and reached satisfactory scores on most of our attempts. Table 6.1 shows benchmark scores for this *Collect* hierarchy against a monolithic structure learning the same curriculum.

The monolithic structure is a single neural network which uses the same inputs and outputs used by our hierarchy (see Fig. 6.1), and 3 hidden layers, instead of the 2 used by each separate skills of the hierarchy. The monolithic structure uses 1610 parameters (i.e. neural network weights), while the sum of the parameters of all the skills of the hierarchy is 1510.

In the first benchmark (interruption on collision) 1 point is given for each collected object, any collision stops the simulation (the agent keeps its current score). The second benchmark (no interruption) does not stop in the event of a collision (but does not help the agent should it get stuck on an obstacle).

The higher score of the monolithic structure on the first benchmark is a consequence of its longer runtime. This shows that the monolithic structure is better at avoiding obstacles than our hierarchy. This can be explained by the fact that the monolithic structure can improve its avoid behaviour during the *Avoid* task, but also during both the *GoToObject+Avoid* and the *GoToDropZone+Avoid* tasks, thus benefiting from more generations and from in-context training. In our hierarchy the complex skills are able to use *Avoid*, but the training only affects the complex skills themselves and not its subskills. Compared to complete modularity which separates a complex skill from its subskills and requires each to learn its own internal function, a monolithic structure has the opportunity to share intermediate results. The structures described in Schrum and Miikkulainen (2015) show policies (subskill) and preference neurons (complex skill) sharing neural paths. We can reasonably assume that in the case of "how to avoid" (*Avoid*) and "when to avoid" (*GoToObject+Avoid*), both behaviours could share some analysis on the surrounding obstacles. This would however come at the cost of modularity.

These results raise the question of evaluating the resources needed to learn a task. The complexity of tasks can be difficult to assess, but it is easy to understand that the neural network assigned to the *GoToDropZone* skill, which uses two inputs, has far fewer connections, and thus fewer weights to adjust, than the neural network assigned to the *Avoid* skill, which uses more than 10 inputs. Figures 6.2 and 6.3 show how this difference in complexity impacts learning. Most attempts of the *GoToDropZone* and *GoToObject* skills reach their maximum value in 100 generations, whereas most attempts of the *Avoid* skill continue to evolve after 500 generations.

It is worth noting that while the complexity of the network to train has an impact on the learning time, so does the nature of the task. *Avoid* takes a longer time to learn, but all the attempts seem to reach an acceptable level of training. This can be explained by the fact that avoiding an obstacle has many solutions, all of which being valid. On the

Interruption on collision (average runtime)		
Structure	**Curriculum**	**Final task only**
Hierarchical	4.63 (54%)	0.06 (51%)
Monolithic	5.19 (68%)	0.31 (80%)

No interruption (equal runtime)		
Structure	**Curriculum**	**Final task only**
Hierarchical	9.55	0.18
Monolithic	7.70	0.39

Table 6.1: Top table: number of objects collected and percentage of runtime before the evaluation ends. Bottom table: number of objects collected when a collision does not end the evaluation. Results are given for our hierarchy and for a single skill monolithic structure. Each structure was trained using the curriculum and the final task only.

other hand *GoToObject* is represented by a much smaller network and takes less time to learn, but accomplishing this task has only two valid solutions:

1. The optimal solution: if the object is on my left, turn left; if the object is on my right, turn right.
2. The suboptimal solution: turn either always left or always right until the object is in front of me.

The suboptimal solution tends to lock the learning process in a local maximum.

The second column of table 6.1 shows the results of learning the whole curriculum as a single task, all reward signals cumulated in the final environment. This is of course impossible without a teaching entity analysing the context and prioritizing rewards, however the runtime scores show the agent managed to learn some form of *Avoid* behaviour. Learning as a single task using a hierarchical structure is a coevolutional approach similar to Whiteson et al. (2003), with the difference that a MIND hierarchy requires to learn the high level controllers. Without the given decision tree used in coevolution, which in effect only makes base skills evolve, no role is attributed to each skill and the decomposition of the task does not take place. For this method to succeed we would need a way to initiate the specialization of each skill, a process of reinforcement would then naturally take place.

The variation in quality of behaviours also led to the questions of whether it is relevant to continue the learning process, that is training the complex skills, using sub-optimal subskills. Would MIND be able to improve and retrain subskills after the whole hierarchy has been trained? The second scenario was designed to study these issues.

6.2 Scenario 2: Focused retraining

6.2.1 Learning with sub-optimal subskills, retraining and learning in broader context

In this scenario we will take advantage of the property of MIND that offers identifiable behaviour to work on the aspect of the curriculum that the agent had the most trouble assimilating.

After the Scenario 1 resulted in a trained and functional hierarchy, we observed that the *Avoid* skill was causing a bottleneck for the performance of the hierarchy. Drawing inspiration from the following quote we experimented with another learning strategy.

> *Hierarchical architectures are particularly sensitive to the shaping policy; indeed, it seems reasonable that the coordination modules be shaped after the lower modules have learned to produce the simple behaviours. [...] experiments [...] show that in fact good results are obtained by: shaping the lower CSs, then "freezing" them and shaping the coordinators, and finally letting all components free to go on learning together.*

Dorigo and Colombetti (1994)

6.2.2 Protocol

In this scenario we use the already trained hierarchy of Scenario 1 and try to improve its performance by retraining the *Avoid* skill which, by our observation of the behaviour, seems to be causing a bottleneck. We experimented with two different methods:

1. **Allocate more resources** to the original training of the *Avoid* skill and measure the performance improvement.

2. Use an alternative training method: **Learning in a broader context**. In this method we will retrain the *Avoid* skill from scratch while the agent is driven by the *Collect* skill and its hierarchy, using the final *Collect* task as the environment and reward function.

6.2.3 Results

Allocate more resources: We initially set up the learning process of each skill with 1000 generations. The resulting *Collect* skill did work, but still regularly failed due to collisions. Naturally, we considered allocating more resources to the *Avoid* task, expecting that its efficiency, and thus the efficiency of the *Collect* behaviour, would improve.

We retrained *Avoid* from scratch, increasing the number of generations from 1000 to 2500, and retrained all the higher level skills based on this new version. With 2500 generations, the final fitness score of the *Avoid* skill increased by 10% compared to the

Structure	score	runtime
Monolithic	5.19	68%
Avoid 1000 gen.	4.63	54%
Avoid 2500 gen.	5.31	56%
Avoid 1500 gen.+context	6.03	90%

Table 6.2: Number of objects collected and percentage of runtime before failure for the different versions of the *Avoid* skill.

Avoid skill trained with 1000 generations (Fig. 6.3). Consequently, the *Collect* skill based on the retrained *Avoid* skill increased its final benchmark score by 14%.

This result conforms with our observation of the behaviour and indicates that the *Avoid* skill is limiting the performance of the hierarchy under the control of the *Collect* skill.

Learning in a broader context: We then used our method of learning in a broader context, using the hierarchy of scenario 1 as a starting point. The *Collect* skill was set as master skill, the final *Collect* environment used, and the reward signal came from accomplishing the *Collect* task. In this context, we retrained the *Avoid* skill from scratch (ignoring the hand crafted *Avoid* reward function in favour of the *Collect* reward function). By only using 1500 generations on the *Avoid* skill with this method, the final benchmark score of the *Collect* skill improved by 30%.

6.2.4 Analysis

By allocating more resources to the *Avoid* skill (more than doubling it), the overall quality of the Collect task was only slightly improved. This leads us to question the quality of the *Avoid* learning task in the curriculum.

The choices made in the elements of the reward function are certainly in question. What we find reasonable to guide the development of a behaviour, it is the nature of evolution to exploit it to achieve the best score, regardless of the spirit of the law. For instance, in very early experiments the only reward for the *Avoid* skill was a negative one which the individual received upon colliding with an obstacle, and so the fittest individual simply did not move. Each subsequent laws added was in turn exploited until we found a set of laws which gave us a behaviour close to what was expected. This illustrates the problem of defining the reward function, which can be as difficult as simply programming the behaviour we want (Dorigo and Colombetti, 1994).

We needed an *Avoid* skill to build the hierarchy, but now that each skill has its own defined role in the hierarchy, instead of trying to improve the quality of *Avoid* on its own, we can train it in the final context of the *Collect* skill, as an element of the hierarchy.

The experimental success of the method of learning in a broader context suggests that there could be many more learning strategies to consider when teaching to a hierarchy, beyond a simple curriculum from basic to complex tasks. In the scope of our work, we found out that going back to further improve the subskills in the final context, once the whole curriculum has been roughly taught, can yield better results than trying to maximize each subskill before moving on to the next.

The initial progressive training of the hierarchy is still of great importance, even if each skill does not have to be trained to perfection, this first run through the curriculum is where each skill will specialize in its role within the hierarchy. When observing the resulting hierarchy learned without curriculum (hierarchical/monolithic in Tab.6.1) we noticed that most branches of the hierarchy were ignored and a single skill attempted to learn the complete task.

Having established each role in the hierarchy, a natural reinforcement process can take place, with the added benefit of being provided on-the-job training (or less constrained solution space). The benefits of this approach was described as "relaxing" the hierarchy in robot shaping (Dorigo and Colombetti, 1994) or as the coevolution mechanism in Whiteson et al. (2003).

The ability to focus the retraining on a particular aspect of the behaviour is one of the benefits of the property of identifiable behaviours (item 2 chapter 4), resulting from a modular approach.

6.3 Scenario 3: Flexibility

6.3.1 The modularity of MIND: Collect with power management

In this scenario we demonstrate that the MIND architecture is best suited to open ended development by adding new skills, and even new sensors, to expand the abilities of an already trained hierarchy. When using MIND, the acquisition of a new skill does not alter previously acquired skills, which remain available for other combinations. Our architecture is also able to integrate new inputs, which would require a monolithic structure to alter its topology and possibly lead to its retraining from scratch. This shows another advantage of MIND as an approach for open-ended development of agents over monolithic architectures.

6.3.2 Protocol

In this scenario we add to the initial collection task the energy consumption and the necessity to recharge the robot's battery. A sensor providing the current power level of the battery and sensors giving information about the direction of the power source are added to the robot. This scenario adds to the previous one the base task of going to the power source to recharge and the complex task of going to the power source without colliding with obstacles, re-using the previously learned *Avoid* task.

The benchmark for this scenario is the same as that in scenario 1 with the addition of power management. The battery of the robot discharges unless the robot stands in the power source area, in which case the battery quickly recharges. The benchmark is run for a given time frame or until the robot fails (if a collision occurs or the power level of the robot reaches zero). Each time the robot brings back the object to the drop zone, it scores one point and a new object is placed randomly in the environment.

We did not include a comparison to a monolithic structure, as the change in topology occasioned by the new sensors would require us to retrain the network from scratch, using the whole curriculum. Although a monolithic structure would certainly succeed in learning this new task, the computational cost and the drawbacks of having to keep the training material available for every subsequent extension of the agent's abilities leave this method out of our consideration for developmental purposes.

The hierarchies

To create an initial hierarchy there are many possible ways to organize the different subskills, all of which are valid, as long as they are able to learn from the curriculum.

When attempting to coordinate a new skill with an already existing complex skill in a hierarchy two possible ways can be considered:

2. Create a new complex skill that coordinates the new skill with the existing complex skill. The new complex skill will be the one that undergoes the training and not the pre-existing one (*Encapsulated variant*).

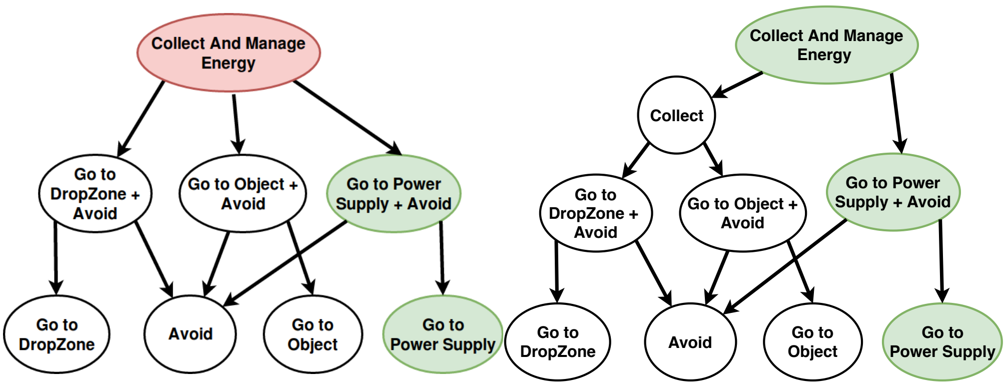


Figure 6.7: The hierarchy for the collection task with energy consumption. Left: Retrained variant: The old master skill is replaced. Right: Encapsulated variant: The old master skill becomes a subskill of the new master skill.

6.3.3 Results and analysis

A video of the resulting behaviour is available at the following address : https://hal.archives-ouvertes.fr/hal-02572023v1

Table 6.3 shows that both variants of the hierarchy (*Retrained* and *Encapsulated* (fig: 6.7)) obtain comparable scores, using the original and the retrained Avoid skill. As pointed out before, in this benchmark the robot can fail if a collision occurs or if its power level reaches zero.

Variant	**Original Avoid**	**Retrained Avoid**
	Score (average runtime)	
Retrained	1.73 (28%)	3.99 (69%)
Encapsulated	1.62 (36%)	3.73 (90%)

Table 6.3: Number of objects collected and percentage of runtime before failure for the two variants of the hierarchy.

No indication of what constitutes a low battery level was given to the robot, only its current energy level. The robot determined by itself when to abandon its current collection task and head for the power source based on its own experience with its battery discharge speed and the size and complexity of the environment.

Both variants of the hierarchy are valid, however we want to point out that the *Encapsulated variant* preserves the original *Collect* skill, making it available for future combinations, which is one of the desired properties of the MIND architecture (item 1 chapter 4).

Another property of MIND that is successfully demonstrated here, by the integration of the sensors related to power management into the hierarchy, is the flexibility of body (item 5 chapter 4). The local coordination of the added physical elements with the new behaviour did not negatively impact the previously acquired behaviours. The new sensors did not require altering the existing internal functions and their topology, the existing skills remain available for future combination and will not have to be learned again.

6.4 Conclusions on reactive hierarchies

This concludes our initial set of experiments aimed at demonstrating the merit of the MIND architecture and the Influence mechanism on simple reactive behaviour. MIND was able to learn a complex behaviour from scratch, by progressively building skills of higher complexity on top of previously acquired skills, and was able to deal with both sequential and simultaneous composition of skills.

Adding the advantages of vector composition (Arkin and Balch, 1997; Simonin and Ferber, 2000; Heess et al., 2016) while keeping the ability to form multi level hierarchies (Larsen and Hansen, 2005; Lessin et al., 2013) is a significant improvement on Robot Shaping techniques. Giving the higher level skills direct access to sensor data (Larsen and Hansen, 2005), including data from sensors not involved with the subskills, improves the decision process and helps decouple skills from each other (compared to RSH, we only use a unilateral control signal), at the cost of not sharing intermediate analysis between skills. The influence mechanism proves to be suited to direct neurocontrol (Lessin et al., 2013; Levine and Abbeel, 2014; Pérez et al., 2017; Devin et al., 2017) as well as motor primitives or schemas (Arkin and Balch, 1997), as evidenced by the use of programmed skills (*Claw control* or motor primitives from Fig. 6.6). The encapsulation of the skill internal function makes the choice of the controller independent from the use of MIND which is an advantage over comparable systems (Lessin et al., 2013, 2015), and suits a more controlled "shaping" approach.

The modular aspect of MIND is key to the progressive learning and accumulation of skills, and provides stable and identifiable skills, that is, define an area of responsibility for the skills within the global behaviour. We have seen the benefits of well defined skill identity in the focus retraining scenario (Sec.6.2) and the extension of the hierarchy using encapsulation (Sec.6.3), but beyond that it offers a great flexibility in combination and re-combination of modules in hierarchies. Other levels, for the most part, are not concerned with how a module accomplishes its function, and another module performing the same function can be substituted. for instance a skill relying on a *reach target* subskill is not concerned if the target is reached by flying of by swimming.

However, using a simple reactive hierarchy has limits: skills like *GoToDropZone* and *GoToObject* show that each new sensor tracking a different object requires a new separate skill, even when the behaviour are almost identical. A learning mechanism could include a trial phase of existing skills, using different inputs and simply duplicate the most suited, but in this case, maybe all navigation skills could be organized around a common concept? Since these skills both drive the agent towards a target, we could create a single skill such as *GoToTarget*, and have a higher level skill choose between the object or the drop zone to set as target.

In the next chapter we will experiment with the variable system of MIND that enables skills to store, retrieve and exchange information, and show how we can create a single navigation skill for all possible targets.

Note:
Over the course of this thesis, multiple implementations of MIND were made, the results presented here are from one of the first implementations. As we moved on to other experiments we refined the algorithms and added advanced neural networks as internal functions to improve convergence time. Each of the following experiments is based around a modified version of the collect task, adding in complexity and confronting MIND to a specific problem. Our initial trial of the MIND hierarchy shows 10 attempts to learn the collect task, but over the course of this thesis the collect task was leaned many more times in a variety of contexts using many variations of the internal functions (including NEAT networks, convolutional neural network and various fine-tuning of multi-layered perceptrons).
In each instance, the MIND hierarchy was able to learn this reactive behaviour.

Chapter 7

MIND Variables

This chapter presents the use of variables in MIND, the memory system which provides the skills with the ability to store, retrieve and share information. Providing a buffer for information between skills will change the way hierarchies are designed, intermediate analysis of the environment or results of a decision process can be centralized and distributed among skills, their meaning identified by the variable's name. But of course the crucial aspects of memory systems for developmental agents is to provide the means for internal representations. Memory is required to commit to a task and go beyond reactive behaviour, and the ability for an agent to keep an internal state makes it an individual. This will play an important part in social behaviour, with for instance the notion of role within a group.

Section 7.1 investigates one of the possible uses of variables in exchanging intermediate results between skills, providing a way to organize in-line structures: the output of one skill is the input of another. The named variable used as an intermediary identifies the nature of this information, in the same way a named skill identifies its purpose. This information module, the variable, is available as a source for all skills requiring this information and its value can be set by all skills able to determine this information, according to the rules of the influence mechanism.

Section 7.2 investigates the use of variables as memory, for keeping track of meaningful events or representing internal states. In order to rise above purely reactive behaviour, an agent must develop its own internal states. The interpretation of these states will drive its behaviour, making it a distinct individual animated by its own motives. The choice of representation and interpretation can vary for each individual, its validity is determined by its practical use and ability to extract relevant information from its interaction with the environment.

7.1 Scenario 4: Target Variable

7.1.1 Selecting between inputs

This scenario presents the use of variables for the centralization and exchange of information between skills. The result of a computation made by a skill is stored and made available as input for a different skill. The benefit of this approach is here demonstrated by having a single skill in charge of all navigation tasks, its goal position being provided by a variable set by higher level skills.

7.1.2 Protocol

The goal is to learn the same collect behaviour as Scenario 1, replacing the *GoToObject* and *GoToDropZone* skills by a single skill, *GoToTarget*, using a variable, *Target*, as its orientation input.

As such, the environments and reward functions will be the same as the *GoToObject* environment of section 6.1.

MIND hierarchy

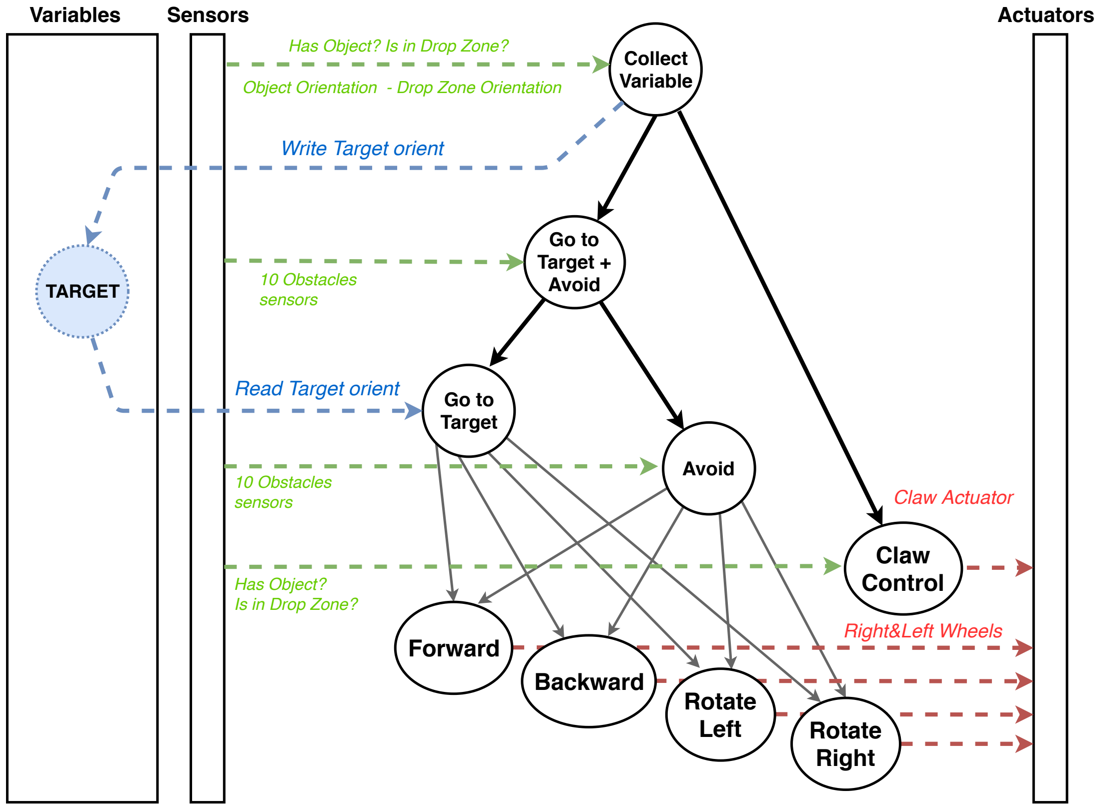


Figure 7.1: Collect hierarchy using a variable for the target

The *Collect Variable* hierarchy uses a different organization than the previous hierarchies at the base skill level. The 5 base skills use Java procedures as their internal functions. Four of the base skill controlling the wheels do not use any inputs and provide a constant behaviour:

1. *Forward*: Sets both wheels to full speed forward.

2. *Backward*: Sets both wheels to full speed backwards.

3. *RotateLeft*: Sets the right wheel to full speed forward and the left wheel to full speed backwards.

4. *RotateRight*: Sets the left wheel to full speed forward and the right wheel to full speed backwards.

5. *ClawControl*: Closes the claw when it senses an object in range, opens it when it is in the drop zone.

On these base tasks we build two higher level complex skills using the four movement base skills:

1. *Avoid*: Move while avoiding. Its inputs are the 10 proximity sensors, and also a movement sensor and a sine wave generator to solve deadlocked situations.

2. *GoToTarget*: Similar to *GoToObject* and *GoToDropZone*, the orientation to the target is given by a variable instead of the object or drop zone sensor.

GoToTarget + Avoid coordinates *Avoid* and *GoToTarget* using the proximity sensors, and is identical in function to the previously described *GoToObject + Avoid* and *GoToDropZone + Avoid* skills.

Finally, the *CollectVariable* skill coordinates *GoToTarget + Avoid* and the base skill *ClawControl*. It also outputs to the *Target* variable. Its inputs are the object and the drop zone presence sensors, in order to determine what must be done, but also both orientation sensors to the object and the drop zone in order to set the *Target* variable.

Remarks on the curriculum and the need for a Drive Module

The experimental setup, the environments and reward functions, remain the same as before, however a few adjustments in the curriculum had to be made to accommodate this particular hierarchy.

As usual, the curriculum is learned by starting from the lower level skills, building the higher levels on top of the previous ones. Figure 7.1 shows that *GoToTarget* uses the value of a variable that is set by the *CollectVariable* skill. Since *GoToTarget* is learned before *CollectVariable*, nothing sets the value of the *Target* variable during the initial training of the skill.

This issue was first solved using elements of the retraining methods explained in the previous part: *GoToTarget* is the learning skill, however another skill is used as a master skill. This new master skill uses a Java procedure as its internal function and simply copies the value of the object orientation sensor to the target variable and gives a constant maximum influence to the *GoToTarget* skill.

```java
public class VarFeeder extends HardCodedSkill{
  public void doStep(double[ ] in, double[ ] out)
  {
    // sets the influence to GoToTarget to 1.0
    out[0] = 1.0;
    // Copy the value of the object orientation to the Target variable
    out[1] = in[0];
  }
}
```

Using this hierarchy, *GoToTarget* is trained in a similar environment to the one used to train *GoToObject*.

However, this technique requires altering the hierarchy for learning and adding skills with no other purpose than “scaffolding” the learning process.

This need for direct control of higher level functions, either for learning or setting a goal for exploitation, leads to the development of the Drive Module described in 5.1.3.

7.1.3 Results and analysis

Videos of the results are available at the following address:
https://hal.archives-ouvertes.fr/hal-02924783

Figure 7.2 shows the trajectory of the agent collecting an object, the behaviour is similar to the original hierarchy presented in section 6.1.

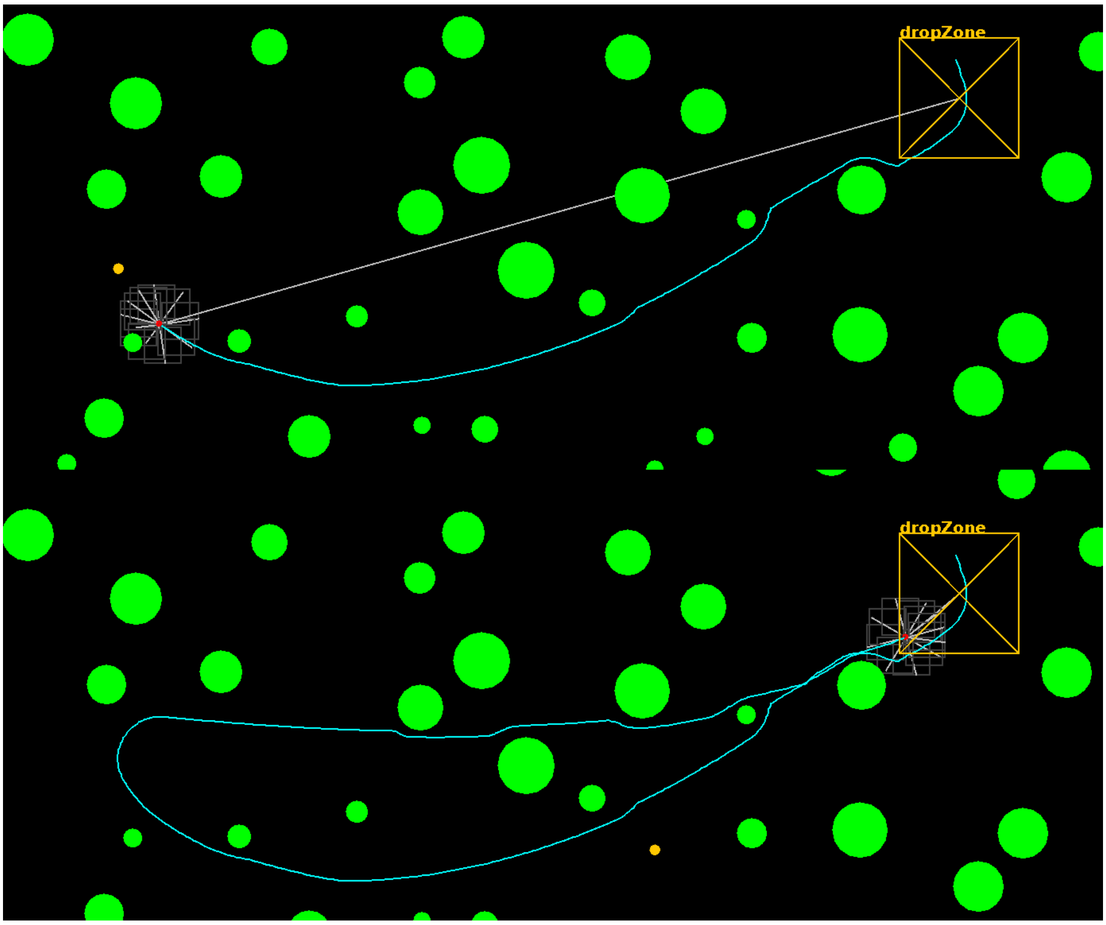


Figure 7.2: Trajectory of the *Collect Variable* skill collecting a single object.

During one of the attempts to learn the behaviour, the *Collect Variable* skill succeeded despite a malfunction of the sensor informing the agent of the presence of the object in the claw. It was able to recognize the case where it had to bring back the object by exploiting the fact that when the agent carries an object, the object is perfectly centred in front, being carried in the claw. When the object orientation sensor gives the information that the object is exactly in front, the agent interprets it as if it is carrying the object. Believing it is carrying the object, the agent starts to turn towards the drop zone. In the case where the object is held in the claw, the value of the orientation sensor would not change, but since it isn't carrying the object, the value changes and the agent corrects its course to reach the object.

The resulting behaviour was a slight zig-zag trajectory when reaching the object, then a straight line when bringing it back. This can only be observed when all obstacles are removed, the constant need to avoid collision masks this behaviour when obstacles are present.

This made us consider that when an agent brings an object to the drop zone, the object is always in front, and under these conditions the output of *GoToObject* would be moving straight forward. Because the Influence mechanism of MIND can perform vector composition, the *GoToObject* behaviour could remain active while the agent brings the object back to the drop zone without any adverse effect on the trajectory.

All further experiments on a *Collect* behaviour were conducted with multiple target objects and the added constraint that objects being carried could not register on orientation sensors. This configuration did not affect the original *Collect* hierarchy, however the *Collect Variable* hierarchy did not achieve optimal results as shown in figure 7.3.

Figure 7.3 shows the trajectory of the agent when collecting a group of 3 objects. Each row shows the collect of an object, the left panel shows the agent reaching the object and the right panel shows it reaching the drop zone.

When the agent brings back the second object it can be clearly seen that the orientation of the drop zone and the third object cause some confusion in the direction to follow (the unnecessary loop on the right side of the screenshot).

We suspect the use of a neural network as an internal function might not be suited to this specific task. The skill must perform a strict mutual exclusion, in one case setting the target variable to the object, and strictly the object, in the other case to the drop zone, and strictly the drop zone as the closest object detected isn't the one carried in front of the agent. In addition, this exclusive decision has to let one of the input signals (orientation of the object or orientation of the drop zone) through the network without distorting it.

The requirements for this internal function are very similar to what we can obtain using a MIND hierarchy: one higher level decision inhibits or lets through lower level signals without altering them. This lead to the creation of the influence neural network topology discussed in section 9.2.4, which we wish to evaluate in future research.

We experimented with various topologies which slightly improved the behaviour but specific combinations of inputs still cause erratic behaviour.

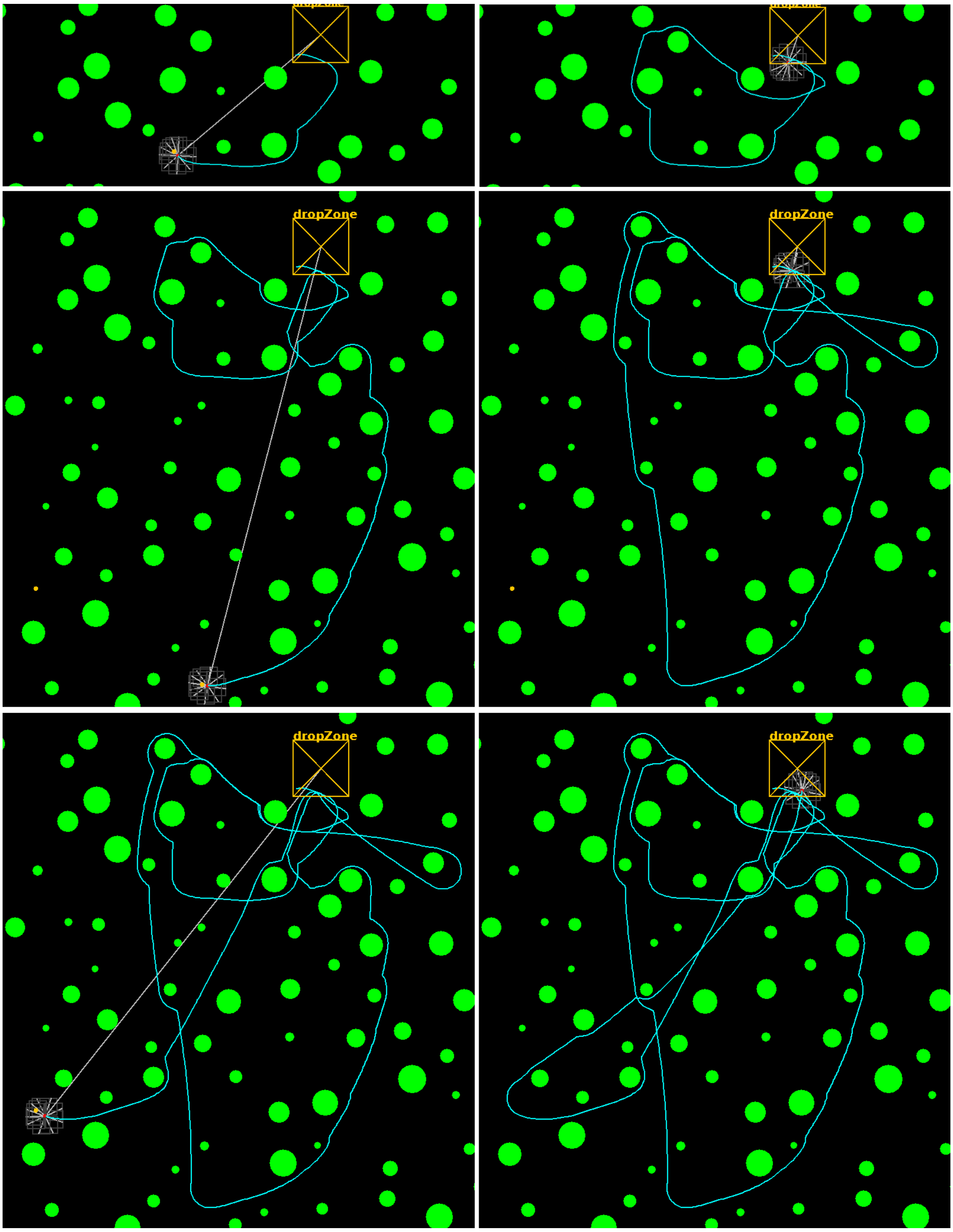


Figure 7.3: Trajectory of the *Collect Variable* skill collecting 3 objects.

7.2 Scenario 5: Counter Variable

7.2.1 Counting and memorizing

This scenario presents the use of variables as memory and internal representation.

The final desired behaviour of the agent is to perform a sequence of actions, without any indication in the environment of which step of the sequence the agent has already accomplished. By memorizing the current step of the sequence we depart from purely reactive behaviour and begin to investigate emergent memory representations and the process of evolving low level cognitive functions from the ground up.

We will use a protocol that respects the barrier of the interiority of the agent's mind by not allowing the teaching entity direct access to the memory modules. The learning process remains a genetic algorithm dependent on a reward function, which is determined by observing the behaviour of the agent in the simulation.

7.2.2 Protocol

The agent must collect an object twice, then activate a validation trigger by moving into a zone. This sequence is repeated indefinitely:

- Collect-Collect-Validate
- Collect-Collect-Validate
- Collect-Collect-Validate
- ...

A small reward is given for each object collected and a large reward is given for one full sequence. An incorrect sequence (3 object collected) ends the simulation.

MIND hierarchy

The *Count Objects* is built on the previous *Collect* hierarchy. The *Count Objects* skill uses the *Collect* skill and *Go To Valid + Avoid*, which drives the agent to the validation zone while avoiding obstacles.

The *Count Objects* skill uses the object, the drop zone and the validation zone presence sensors. The *Count* variable is used as input, to choose between the collect or the validation task, and used as an output to increment the sequence step and reset it.

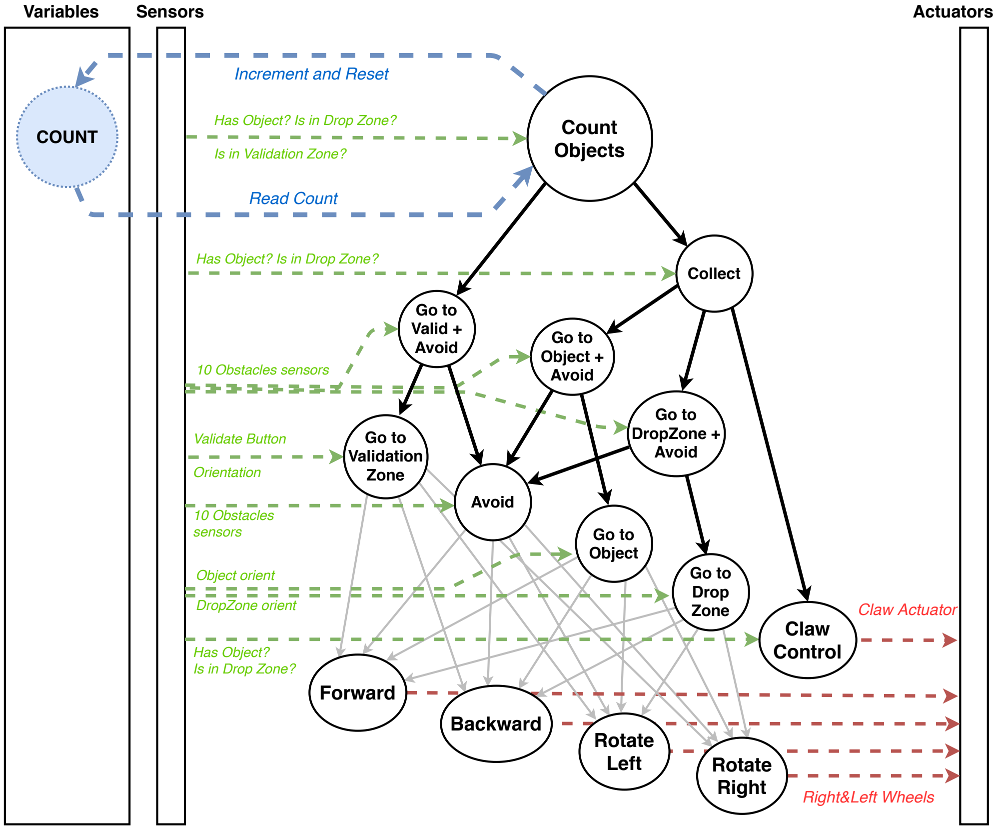


Figure 7.4: Collect hierarchy using a variable for the target

7.2.3 Results

Videos of the results are available at the following address:
https://hal.archives-ouvertes.fr/hal-02950608

The following figures (Fig. 7.5 and 7.6) show the resulting behaviour and state of the MIND hierarchy over 5 steps of the process.

- **Step 1:** The agent brings the first object to the drop zone, VAR_COUNT is at 0
- **Step 2:** The agent just dropped the first object, VAR_COUNT incremented by 0.1
- **Step 3:** The agent brings the second object to the drop zone.
- **Step 4:** The agent just dropped the second object, VAR_COUNT is incremented by 0.1, bringing it to 0.2. The agent is now headed to the validation zone.
- **Step 5:** The agent just reached the validation zone, VAR_COUNT is reset to 0. The agent is headed to the object to pick up.

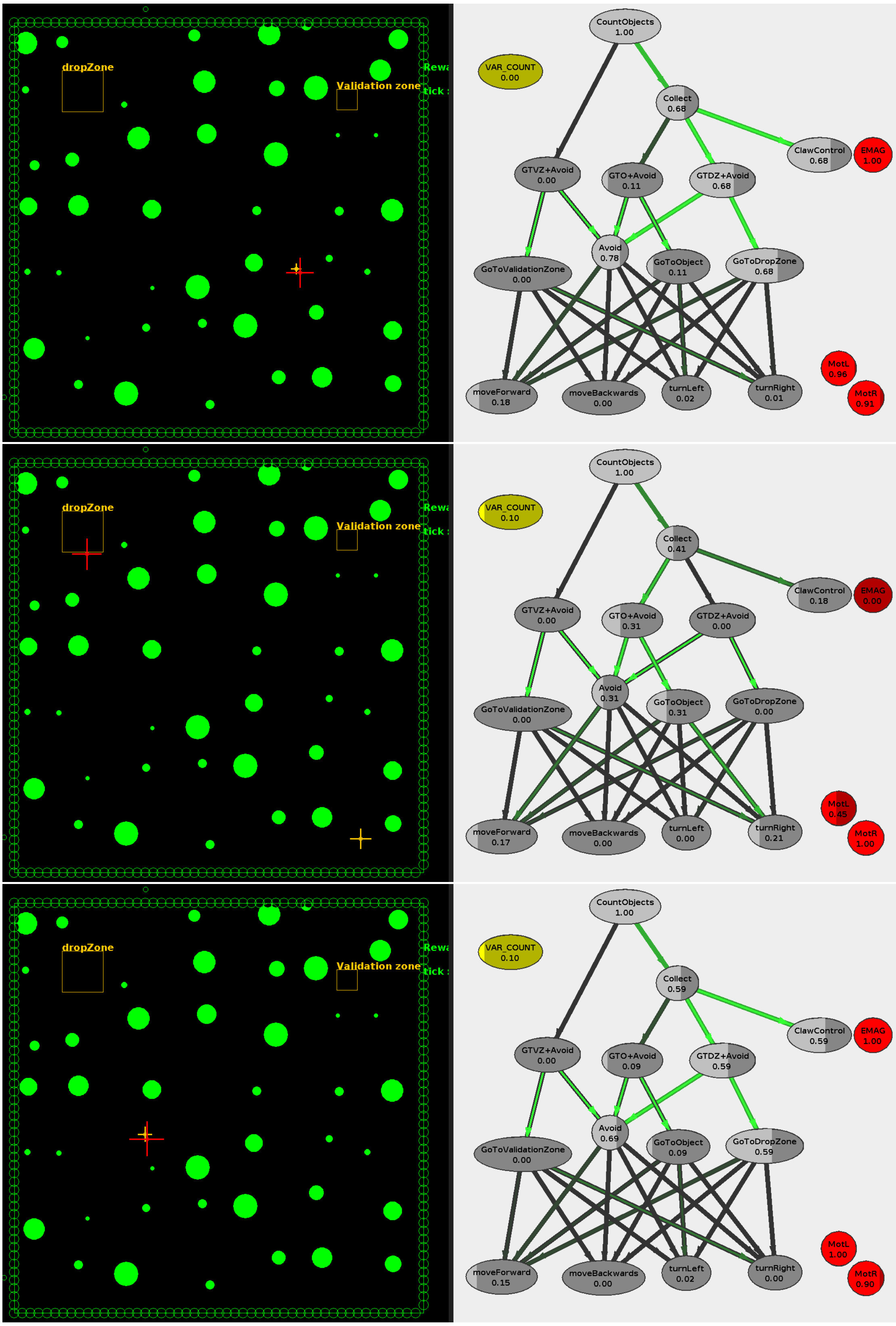


Figure 7.5: Steps 1 to 3. On the left the simulation, on the right the state of the MIND hierarchy

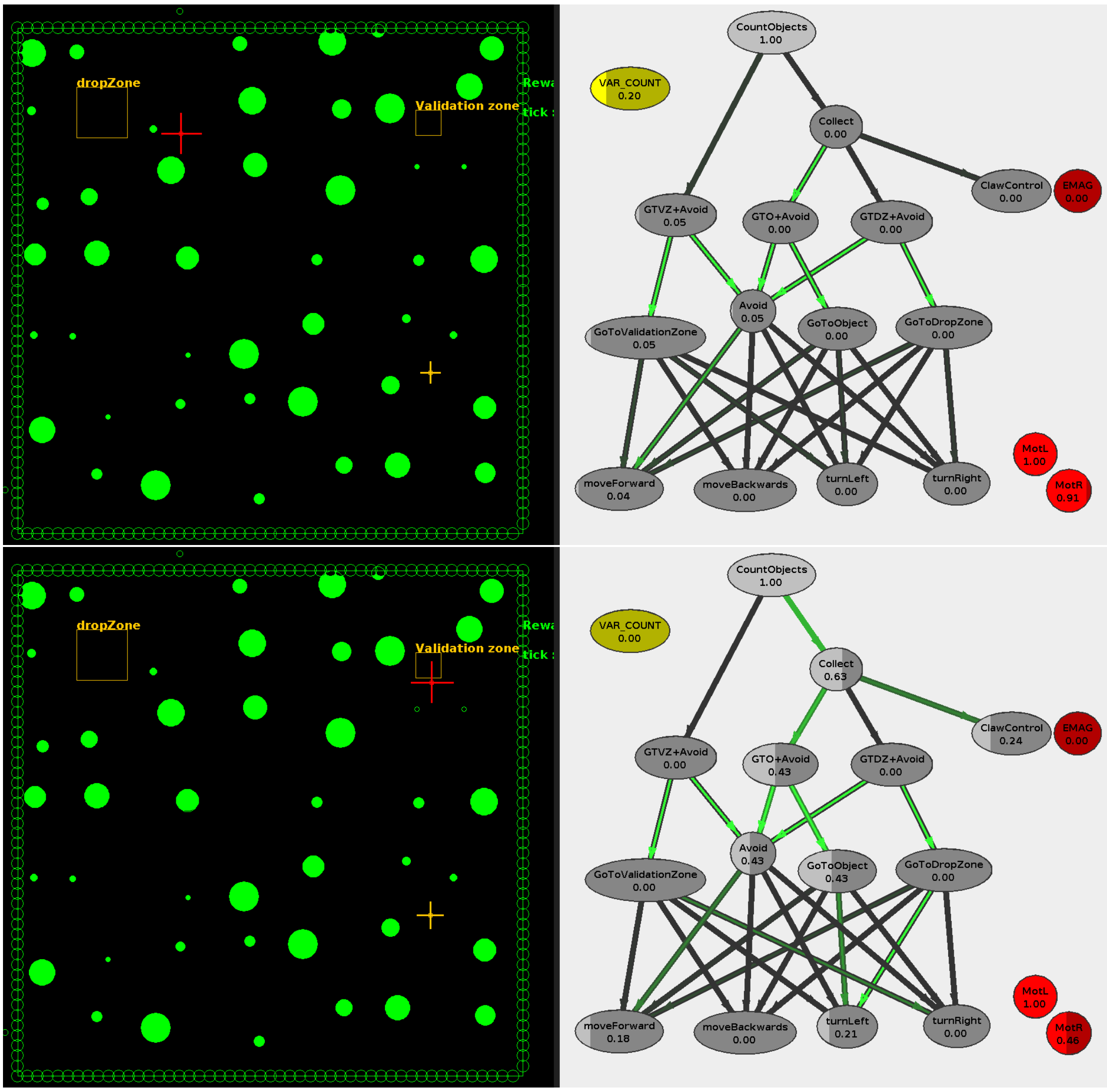


Figure 7.6: Steps 4 and 5. On the left the simulation, on the right the state of the MIND hierarchy

7.2.4 Analysis

The MIND hierarchy was able to learn to count through exposure to a (simulated) real world problem. The learning supervision did not have access to the internal state of the agent, nor did it teach a predefined symbol or memory representation.

Using MIND, our agent is able to go beyond simple reactive behaviour and integrate its internal state, persisting over time and independently of the environment, in its decision process. In turn, the decision process is able to affect its internal state.

The relation between internal states and decisions, and decisions and internal states is learned by the agent. Hence, the internal representation, the significant values of its internal state (the symbols representing them), is formed in an emergent process and grounded in experience.

In the results shown in figures (Fig. 7.5 and 7.6) the agent learn to count from 0, successively increment until 0.2 and reset the value to 0 when starting the sequence over. However, in other attempts at learning the same behaviour, the agent started at a value of 0.1, incremented to 0.3 and reset the counter to the value of 0.1. The resulting behaviour is exactly the same, but it is interesting to note the emergent aspect of the memory representation, in both cases being valid and optimal. The emergence of one or the other is due to the genetic process and the bias in the initial networks that were randomly generated.

In this experiment we set the number of objects to collect to two. In further experiments we could provide the agent with a variable number of objects to collect, through a sensor for instance, and try to learn a function that would match the count against the desired number of objects to collect to trigger the *GoToValidationZone* behaviour.

7.3 Conclusions on variables

Variable modules provide MIND with additional options for its memory systems and internal organisation. Skills are now able to exchange information, which will help generalizing behaviour around identified concepts. An obvious advantage of sharing information between skills is to generalize some behaviours, such as navigation as illustrated by section 7.1. The hierarchy used in the example of the target variable is a very simple one, but variable modules can be used as any sensor or actuator, which means several skills in the hierarchy can use the same variable as input and send concurrent commands to the same variable as output. A simple example would be *GoToTarget* and *FleeFromTarget* using the same target variable for different purposes. Since the target variable simply represents a heading, one could imagine a version of *Avoid* setting the heading of the agent to avoid collisions being concurrent with a *GoToObject* skill on the target variable.

Sensor modules already provided a low level memory system, an history of past samplings, but with the variable module, any form of processed information can be memorized offering the agent the ability to commit to a behaviour. Since this information is shared by skills with the ability to learn on both sides of the process (storing and retrieving), the meaning of a variable's values will have to be agreed upon through an emergent process. The example given in section 7.2 shows the emergence of the meaning of "enough" by subdividing the values of a variable into classes whose bounds are grounded in experience, and that different representations can emerge.

This series of experiments has given satisfactory results as a first step in the investigation of emergent memory representations. We are confident that MIND makes an excellent test bed for new research on evolving low level cognitive functions from the ground up, its connectionist and developmental approach gives a promising alternative to symbolic A.I.

With a means for internal representations, a MIND agent can now increase the complexity of its behaviour. As shown in chapter 3, one of the requirements to establish a reinforcement process in an agent is the ability to store and retrieve information, its internal state. In MetaCiv, this role is carried out by the cogniton. In the next chapter we will use MIND hierarchies in a multi-agent context, and using variables, we will attempt to learn a behaviour of social specialization through reinforcement.

Chapter 8

MIND Multi-Agent

In the previous chapters we experimented with each feature of the MIND architecture. The encapsulation of skills and progressive learning of complex behaviour, the flexibility and extensibility of a hierarchy, the memory system for internal states and information exchange.

In this chapter we present our final series of experiments on the use of MIND in a multi-agent context, covering the remaining aspects of the development of autonomous behaviours in a society of agents. The scenarios presented here involve complex behaviours, which include learning at both individual and collective levels, and should prove the robustness and potential of the MIND architecture.

In section 8.1, we first evolve a coordination behaviour, based on a reactive hierarchy. With communication through simple signals, identical agents evolve as a species to achieve a common goal in a manner similar to insect societies.

In section 8.2, we use the MIND variables to provide our agents with internal states. This degree of individuality allows them to represent their role within a social organization. In addition to learning how to find the appropriate role within a group on a species level, each agent will learn its own role during its lifetime. By doing so we will attempt to replicate our preliminary work on MetaCiv and social specialization in MAS (chapter 3).

8.1 Scenario 6: Foraging

8.1.1 Multi-agent coordination

This scenario presents simple multi-agent coordination on a foraging task.

The problem of foraging robots is a well-known problem in the field of artificial intelligence and multi-agent systems, whether from the point of view of modelling insect societies (Deneubourg et al., 1991) or developing autonomous robots for space exploration (Brooks et al., 1990). This problem involves issues of distributed problem solving and possibly inter-agent communication, and was used as experimental context for learning swarm robot control policies, requiring the coordination of many simple agents (Pérez et al., 2017).

We wish to learn a multi-agent foraging task based on the pre-existing *Collect* hierarchy (section 6.1) and show that it is possible to extend the capabilities of a MIND agent, in the developmental sense of the term, to new tasks requiring reactive multi-agent coordination behaviour. This multi-agent foraging behaviour close to insect behaviour remains in a purely reactive domain and requires only the exchange of simple signals (Simonin and Ferber, 2000).

8.1.2 Protocol

A group of four agents is placed in the environment. The agents are similar to the ones used in the previous experiments. Each agent has a sensor giving the orientation of the object, with a limited range of perception, a signal emitter and receiver giving the

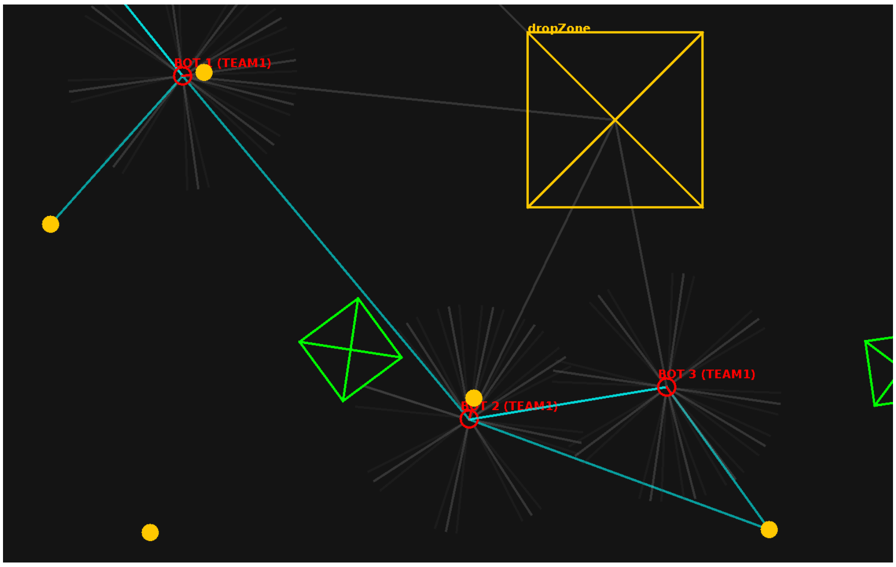


Figure 8.1: the multi-agent simulation environment

orientation of the nearest signal. The agents also have a sensor giving the orientation of the nearest agent.

The environment is a larger version of the collect environment, containing obstacles and a drop zone for the objects to collect. The objects are placed close to each other in the environment in groups of 8.

Agents can only carry one object at a time and their signal emitter has a much greater range than their object orientation sensor.

Since the agents have no information about their environment, the optimal expected behaviour is to look for objects to be collected where no other agents are present in order to obtain the largest observed area. When an agent finds the group of objects, it sends a signal to ask the other agents to come.

All agents in the simulation use the same MIND hierarchy, each with an instance processing the values of its own sensors. The overall response during the evaluation of the group of agents using the same genome for the internal function to be trained constitutes the score of this genome.

The complete foraging hierarchy is shown in figure 8.2. The skills in dotted line use a programmed internal function and the skills in blue are skills that require collective learning.

Initial hierarchy and mono-agent skills This hierarchy contains the sub-hierarchy *Collect* which was previously shown (see chapter 6). This sub-hierarchy was learned individually, with the collection task being performed alone.

Another skill present is *Reach Signal* which combines *Avoid* and *GoToSignal* and allows the agent to reach a signal emitted by another agent while avoiding obstacles.

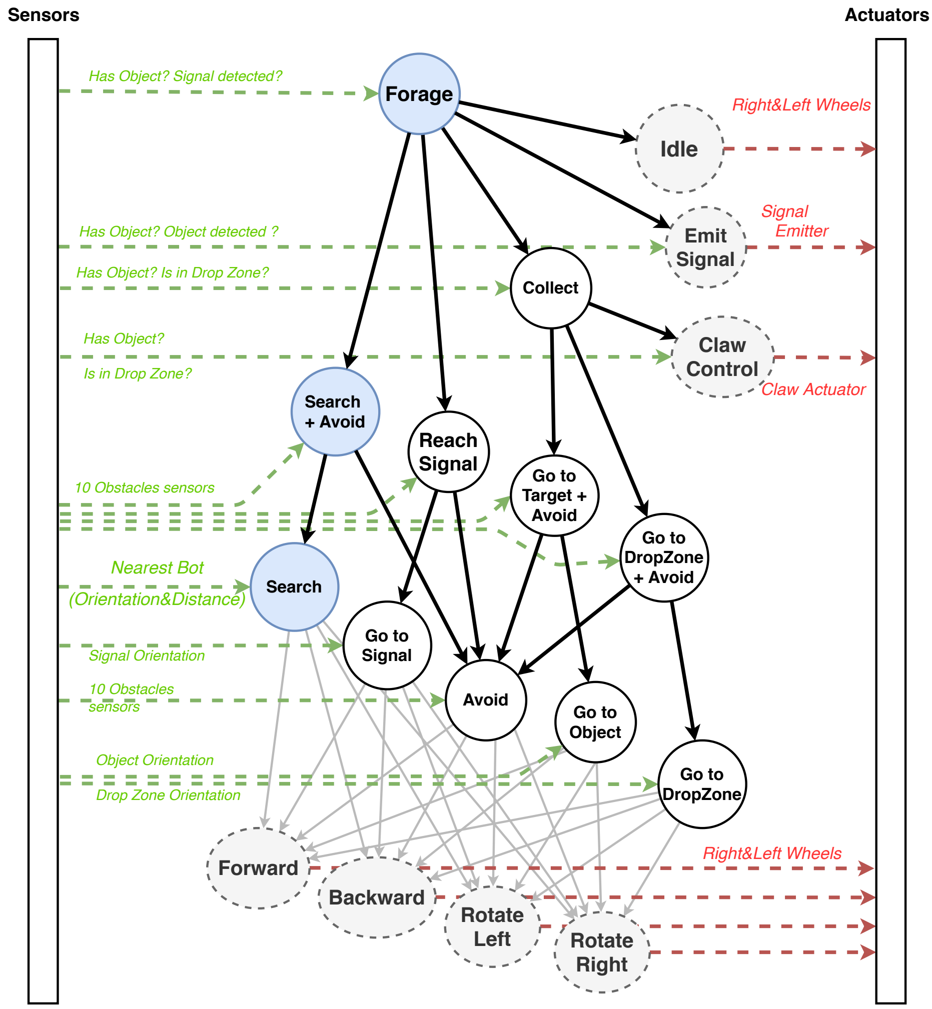


Figure 8.2: Foraging hierarchy

Although this skill implements signals, it has not been learned in a multi-agent context, the learning environment used is capable of simulating the emission of a signal.

Programmed skills Some skills are not learned but programmed, their function being trivial and not justifying the establishment of a curriculum. This is the case of *Emit Signal* which emits a signal as soon as an object is in the perception range. The programmed skills of the Collect hierarchy are also used: *Forward*, *Backward*, *Rotate Left*, *Rotate Right* and *Claw Control*. We add the *Idle* skill which stops both wheels.

Multi-agents skills *Search+Avoid* and its subskill *Search* are skills that are truly dependent on multi-agent operation, and can only be learned in a multi-agent context. An individual agent has no information which can help it establish an efficient search pattern, no memory of places already visited or pheromone/markers which it could deposit in the environment. It does not know the shape or dimension of the environment and cannot establish, for instance, a sweep pattern. However, as a group, agents can leverage the fact that they have several times the perception range of an individual agent and use the information on each other's position to coordinate into a formation that will cover the largest possible area.

Master skill Finally, *Forage* the master skill must coordinate *Search+ Avoid* to find the objects, *Emit Signal* to call other agents, *Reach Signal* to reach an agent who has found the objects before it, and *Collect* to collect the objects in its perception range. As the agent has no memory and no coordinate system to describe the position of the object group, the *Idle* skill can be used to immobilize the agent and wait for other agents to reach the position. The agent thus acts as a beacon marking the position of the object group (Goss and Deneubourg, 1992).

8.1.3 Results

Videos of the results are available at the following address:
`https://hal.archives-ouvertes.fr/hal-02924787v1`

The figures 8.3 and 8.4 present 8 successive steps of the resulting behaviour. The left panel shows the simulation environment (as described in subsection 5.1.5), the right panel shows the state of the MIND hierarchy for the agent **BOT 2**, whose name is displayed above the agent in the environment panel. The hierarchy view shows in red the actuators (placed close to the corresponding skills) and in blue the 4 sensors provided to the skill *Forage*: Is a friendly agent within sensor range? Is a signal active within sensor range? Is a target object within sensor range? Is the agent carrying an object?

The hierarchy corresponds to the one presented in the figure 4.2 with the skills in grey and the influence links showing the current influence transmitted, as described in subsection 5.1.5.

The following describes the successive steps of the behaviour of **BOT 2**, our agent of interest, situated in the bottom left of the environment in the initial state.

Step 1: The agent does not perceive objects, does not receive a signal. The master skill *Forage* activates *Search+Avoid* with maximum intensity (the very low relative value of *ReachSignal* has no impact on the behaviour).

Step 2: The agent perceives an object, the signal is activated, the skill collect is activated in majority, *Search+Avoid* remains partially active but does not disturb the collect behaviour. In this phase *Collect* activates the *GoToObject* branch which allows to reach the object.

Step 3: The agent has picked up the object, the corresponding sensor is active. In this *Collect* phase, the *GoToDropZone* branch is activated, which allows to reach the drop zone. *Forage* completely activates *Collect* and disables *Search+Avoid*. It can be seen that the other agents are attracted by the agent's signal.

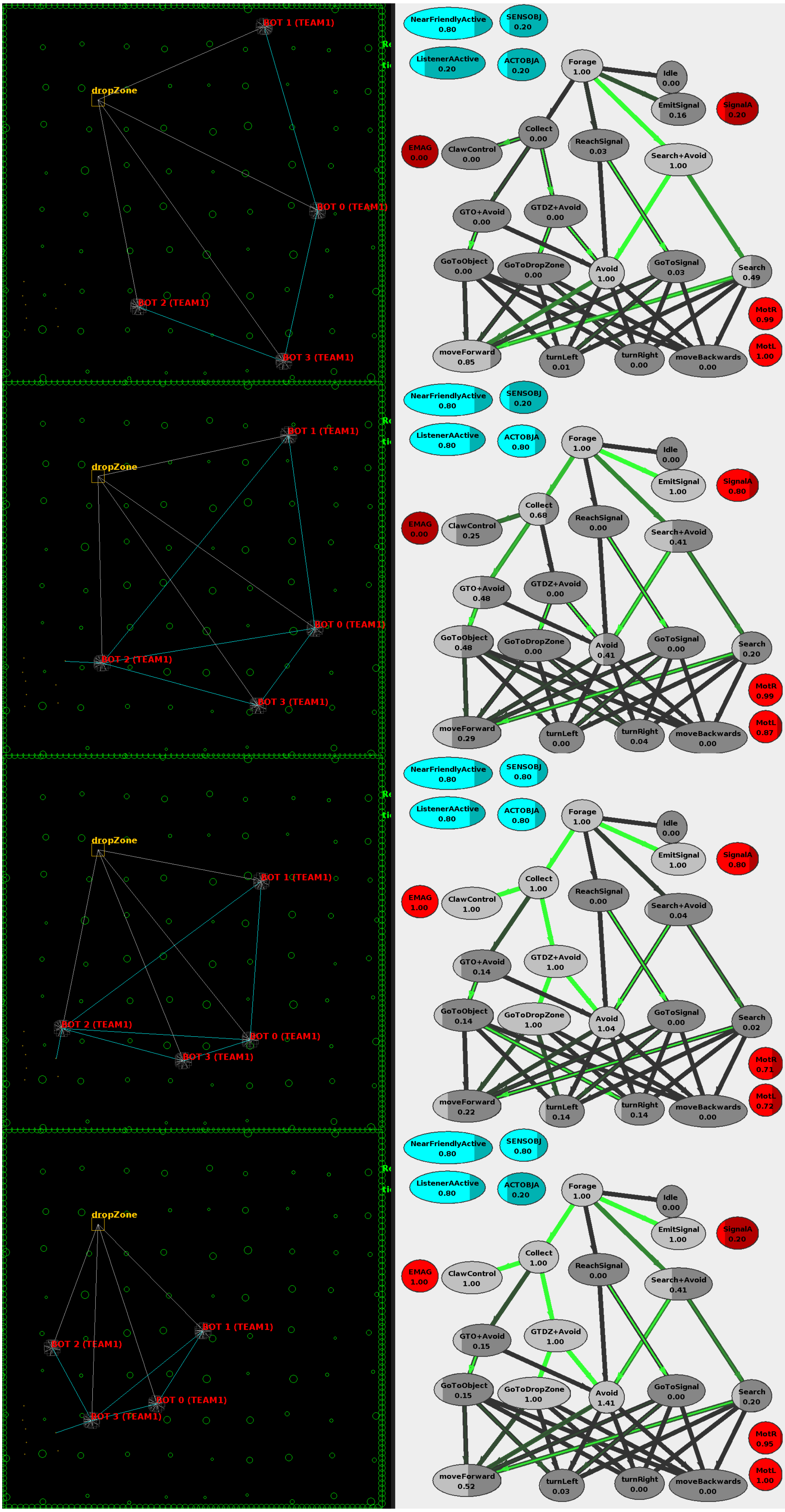


Figure 8.3: Steps 1 to 4

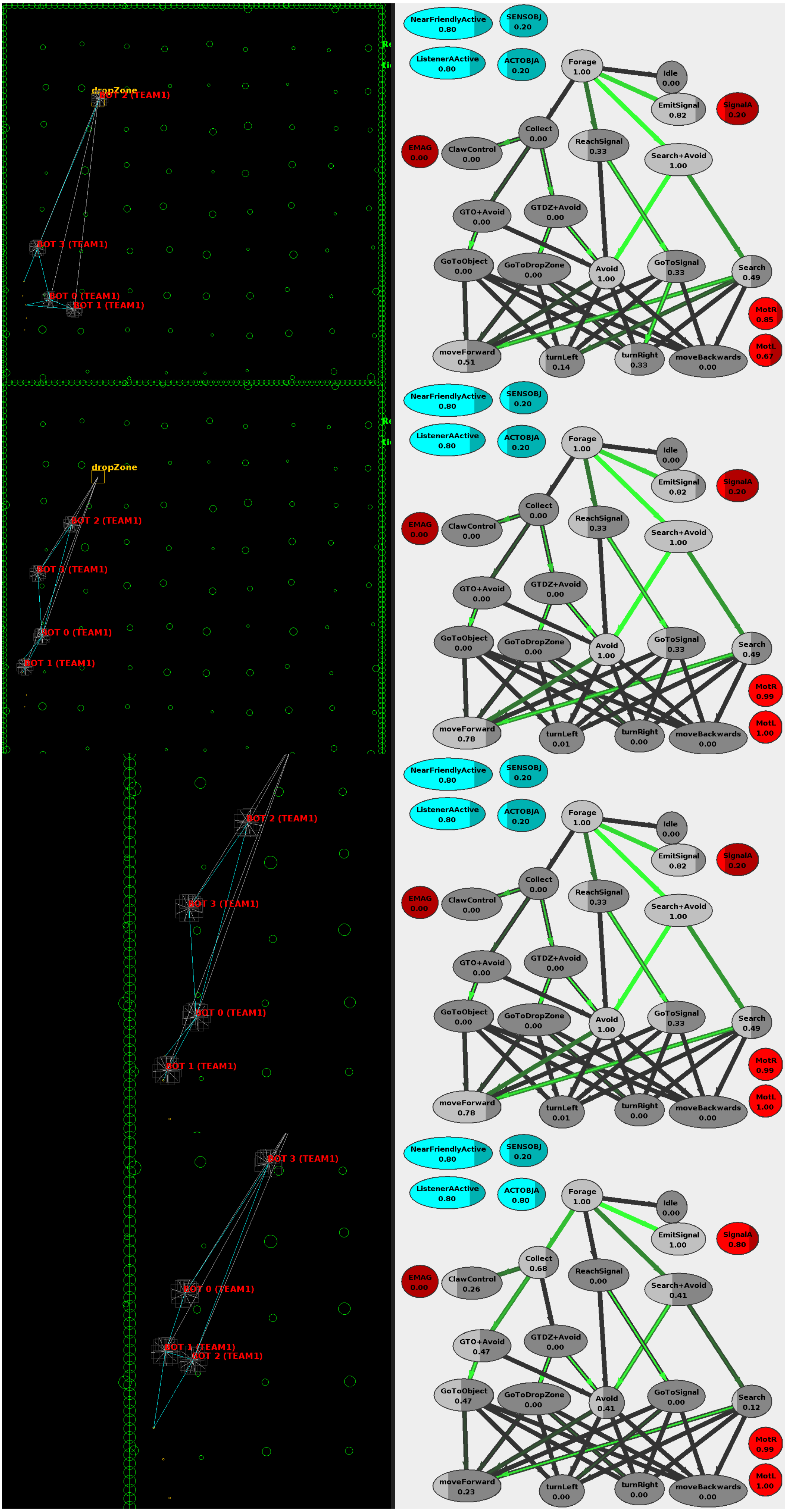


Figure 8.4: Steps 5 to 8

Step 4: The agent on his way to the drop zone goes out of perception range of the object, resulting in his signal turning off. The change in sensor values influences the hierarchy without disturbing the agent's behaviour. Although our agent has stopped emitting the signal, BOT 3 is now within perception range of the object and emits the signal to attract the other agents not yet carrying objects.

Step 5: The agent has deposited the object. *Collect* is disabled, *Search +Avoid* and *ReachSignal* are enabled. Even if *ReachSignal* receives less influence than *Search+Avoid*, we can see that the influence transmitted results in the majority of the behaviour required by *ReachSignal* being activated (i.e.: *turnRight*:0.33 versus *turnLeft*: 0.14).

Step 6 and 7: The turn towards the signal source is done, the agent moves in a straight line towards it.

Step 8: The agent is once again within perception range of a target object and repeats the collection process.

8.1.4 Analysis

The development of a MIND hierarchy for multi-agent coordination does not present any particular constraints or difficulties, other than respecting the point of view and philosophy of the Agent approach (Ferber, 1995). Coordination is done locally, implementing the simplest solutions by coordinating with the closest agents. Moreover, the genetic algorithm is extremely flexible regarding the composition of the skill provided. For instance, during a different experiment on the Collect task, an error in the parameters made the agent unable to grasp the object. Using the physics engine of the simulation, the agent simply pushed the object into the drop zone. We can see in our case the behaviour of the agent in step 5 or the fact that the *Idle* skill allowing the agent to wait for the other agents to arrive was not used.

The great difficulty, as Dorigo and Colombetti (Dorigo and Colombetti, 1994, 1998) noted, lies in the elaboration of the fitness function for the genetic algorithm. Adjusting the balance between several sources of rewards and finding the value to be assigned to each is a process that often requires several attempts to observe (and interpret) the impact of each setting. In addition to the tuning of reward value, many parameters will have an impact on the fitness of an agent and in some cases the relations between the parameters of the simulation and the genetic algorithm can be quite subtle, to the point of being affected by seemingly innocuous output encoding.

In Pérez et al. (2017) agents must use an output neuron of their ANN to display one of several possible colours. The possible colours are mapped to the output neuron covering its whole range, the transfer function of the neuron clamps the output within its range. In this case, keeping both extreme values of the range as possible choices will introduce a bias in the genetic algorithm, as the initial random weights are likely to provide an input on a wider range than the output. As a result, the genetic algorithm quickly converges towards solutions using both extreme values by benefiting from the reward from two easily separable values at the expense of difficult separation of all the values in the range. In this case, the output encoding could be tuned to help the genetic algorithm by adding two invalid colours which do not give rewards at both ends of the range. Doing so would disadvantage any configuration which tends to provide an input outside of the output range of the neuron, removing them at an early stage before they shape the general population.

In our case the challenge resides in finding the proper balance of the environmental parameters and understanding their interactions with the fitness function. For instance, in our results, the final behaviour does not use the *Idle* skill at all. When observing the behaviour of the agents we can see that in most cases the agent who finds the group of objects first rarely has time to leave the perception range before the other agents arrive; therefore it is not necessary to wait for them. It even saves time to start the *Collect* behaviour right away, which will give a chance to collect more objects in the allotted time. Here the fitness function is not directly at fault but simply the size of the environment which does not give the agents the opportunity to be far enough apart to need to use the *Idle* skill. On the other hand, the size of the environment, the dispersion of the group of objects and even the density of obstacles in the environment have an impact on the learning and use of a coordinated object search instead of a simple search through a random walk of the environment.

This experiment has shown that the MIND architecture is capable of performing a coordination task for a homogeneous group of reactive agents. It was able to combine behaviours acquired individually with new strictly collective behaviours, without any major change in the learning techniques employed. The only impact on the cost in computing resources comes from the management of signals in the simulated environment. The modular aspect of MIND has once again shown its advantage since it has been possible to add not only new behaviours, but also new sensors and actuators without any modification to the original hierarchy. Taken individually, the *Collect* behaviour is still capable of operating autonomously.

8.2 Scenario 7: Foraging role

8.2.1 Social specialization

In this final scenario we attempt to replicate the work on social specialization presented in chapter 3, where agents learn their role within a group through reinforcement. In addition to learning their role from an individual perspective over the course of a simulation, we use MIND to learn the reinforcement mechanism itself, from a species perspective. By doing so we will have covered the development of agents, from sensorimotor skills to social behaviour, learning and adapting on different scales and in different scopes.

8.2.2 Protocol

The goal for the group of agent is to collect two different kinds of objects, referred to as **A** and **B** (respectively, yellow and purple in the following figures), to match a given ratio: 1 object **A** for two object **B**.

All agents are created equal, no special initialization is done or random values assigned, aside from having slightly different starting positions as two agents cannot occupy the same space at the same time.

The agents are similar to the ones used in the previous experiments. Each agent has sensors tracking the two different kind of objects. These sensors have unlimited range of perception, as exploration isn't the focus of this experiment. The agents also have a *payoff* sensor giving information to the agent about the "worth" of the object collected. This worth is a function of the quantities of each objects collected to the desired ratio.

Agents have no means of communication, their relation to the group is done through the "payoff" for their individual labour compared to the need and total labour of the group.

The environment is a version of the collect environment, containing obstacles and a drop zone for the objects to collect. The objects are placed away from the drop zone, each kind on one side of the environment, with slightly more object of each kind than the number of agents.

A distance between the objects and the drop zone will help agents settle in a role. The agent should not react instantly to the "payoff" for its work as the situation will evolve between two collect: other agents will have brought back objects which will affect the collect ratio.

Figure 8.5 shows the complete hierarchy for this task. **Collect A** and **Collect B** are duplicates of the **Collect** skill whose sub-hierarchies use the sensors for the **A** and **B** objects.

MIND hierarchy

Collect Select This skill select either **Collect A** or **Collect B**, depending on the **ROLE** variable. It also has access to the distance information to object **A** and **B**. Should

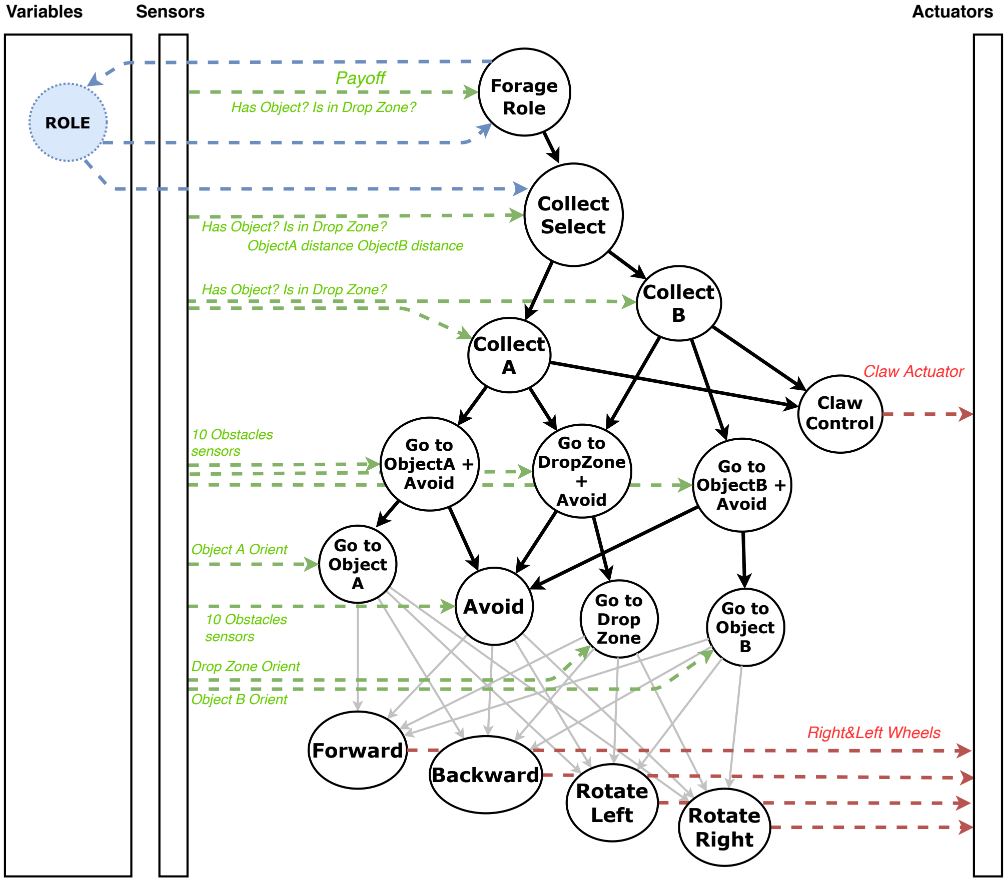


Figure 8.5: Foraging with roles

the **ROLE** variable be ambiguous, the agent could choose the closest object thus starting the reinforcement process using the chaos of the simulation as random initialization.

Forage Role This skill has access to the last payoff information, the status of the collect task and the current role in order to set the value of the **ROLE** variable.

8.2.3 Results and analysis

Videos of the results are available at the following address:
https://hal.archives-ouvertes.fr/hal-02950611v1

Although the problem presented here is more abstract than the CogLogo experiment on farmer-artisan social specialization (chapter 3), from a modeller's perspective the fundamental challenges remain the same.

What our CogLogo model prepared us to expect is that even when the model consistently leads to a stable emergent organization, chances and accidents give rise to different path towards organization. In order to properly evaluate genomes we decided to increase the number of repetition of the evaluation of each genome. This would orient the evaluation towards consistency over performance.

In the case of learning through reinforcement, and in particular in learning a social role in an unstable environment, a major concern is finding a system with a proper balance of stability and adaptation to changes. The specialization into a role must have a certain inertia to smooth the high frequency fluctuation of the social demand, while at the same time remain able to react to a general tendency of the social environment.

In both the social model and our developmental experiment, a part of the solution resides in the model of the social demand, or "market". In section 3.3.3, we briefly explain how commodities are regulated: the wheat produced by farmers is a universal need, which must be continuously produced and consumed and cannot be accumulated beyond a point (due to a rule of decay). The tools produced by the artisans are a secondary need of the farmers, which are only consumed when used and can be accumulated. Without entering into details of the model (such as the conversion of perishable commodities into accumulation of non-perishable wealth), wheat is the primary (universal/essential) product of the society, tools being produced *as a function* of the primary product. A similar rule of decay was applied to the objects collected. Both counts were divided by a constant over time, with the effect of lessening the impact of past states of the "market" in favour of its current state. This helps reduce the "pendulum" effect, where the reaction to an imbalance create the opposite imbalance due to inertia. When setting up a model for simulations involving reinforcement, the parameters of the initial state must be considered with regard to the mechanisms involved, it is a good general rule to remember that *a world has existed before*. Starting from an imbalanced state will drive the agent to a behaviour leading to the opposite extreme.

In our experiment, we initialized the number of collected objects to values fitting the desired ratio. High initial values are preferable, adding 1 object to a 20:10 count will have a smoother impact on the ratio than to a 2:1 count. Since this count is subjected to the rule of decay we established, the initial values will fade out in favour of the real collect rate once the initial unstable period is over.

The other aspect of such reinforcement models is evidently the reinforcement mechanism itself. While in simulations, such as our CogLogo model, the reinforcement mechanism is programmed, the challenge of this experiment was to learn this reinforcement mechanism. Programming a reinforcement mechanism involves three aspects. Storing the calculated value for future use, this is the purpose of the roles Cognitons in CogLogo. Conversion from observation to the reinforcement value, how much the measure of success/failure impacts the state of the agent. Finally, a possible regulation mechanism, for instance a mechanism comparable to the rule of decay, where in the absence of reinforcement the system slowly reverts to a neutral state.

Storing the value of the role and making it available to the decision process is accomplished through the role variable and its links to different skills. Part of the observation is given through the calculation of the value of the payoff sensor, a sufficiently large range of possible collect ratios expressed within the 0.0/1.0 bounds of our input-output model.

However, the effect of this observation on the role variable and the possible regulation mechanism are controlled by the internal function of the skill and can only be learned. The implementation of the reinforcement model is replaced, in this context, by setting up the proper learning environment and reward function.

In experimenting with different settings we obtained very different results, some of which parallel the outcome of programmed models. First of all, we encountered the usual result of “circumventing the problem” (or outsmarting the designer) genetic algorithms are notoriously good at attaining. When setting the collection ratio to 1:1 (one object A for one object B), our agents would all start with one role, switch to the other role upon collecting an object, switch back to the other role upon collecting the next object ... no reinforcement mechanism is needed to achieve this behaviour, and on average lead to an equal ratio of object collected. For the following attempts, the ratio was set to 2:1.

In setting the values for reward functions a delicate balance must be found. When setting the reward function with no punishment, and a higher reward for keeping close to the ratio, the outcome was a tendency towards inertia and stabilizing role distribution on ratios between 2:1 and 1:1. Adding punishment for incorrect ratios produced instability of roles.

After many trials and fine-tuning of the all the parameters of the model, the environment, the reward functions, and trying alternative hierarchies and variable types, we were not able to get the results we were expecting.

However, in the typical fashion of genetic algorithms, our agents did learn a behaviour that accomplishes the goals we set, but not in the way we wished. This behaviour exploits extreme imbalances and collapses of the system to draw roles for the agents randomly. If the random distribution of role does not give the desired ratio, the minority switches back to the majority role, causing a new extreme imbalance and a new random distribution takes place. In a way this can be seen as an “accelerationist” policy. Figures 8.6 and 8.7 illustrate this behaviour.

Step 1: In the initial stage, all the agents set their role to 1, collecting B objects in purple. This will rapidly lead to an incorrect ratio and a decreased payoff.

Step 2: As the payoff sink very low from the extreme imbalance in role distribution, almost all agents change their role (0 corresponding to A objects, in yellow).

Step 3: The high payoff of the new role will rapidly drop, causing another migration, however the first few to switch roles will affect the payoff before the rest of the group has time to collect their object.

Step 4: A few agents will remain in the old role, this selection is due to the chaos of the simulation, and there is no guarantee that the ratio will be correct. In this case 2 agents remained in the role A and 7 went back to the role B. A 1 to 2 ratio requires 3 agents in the role A and 6 agents in the role B.

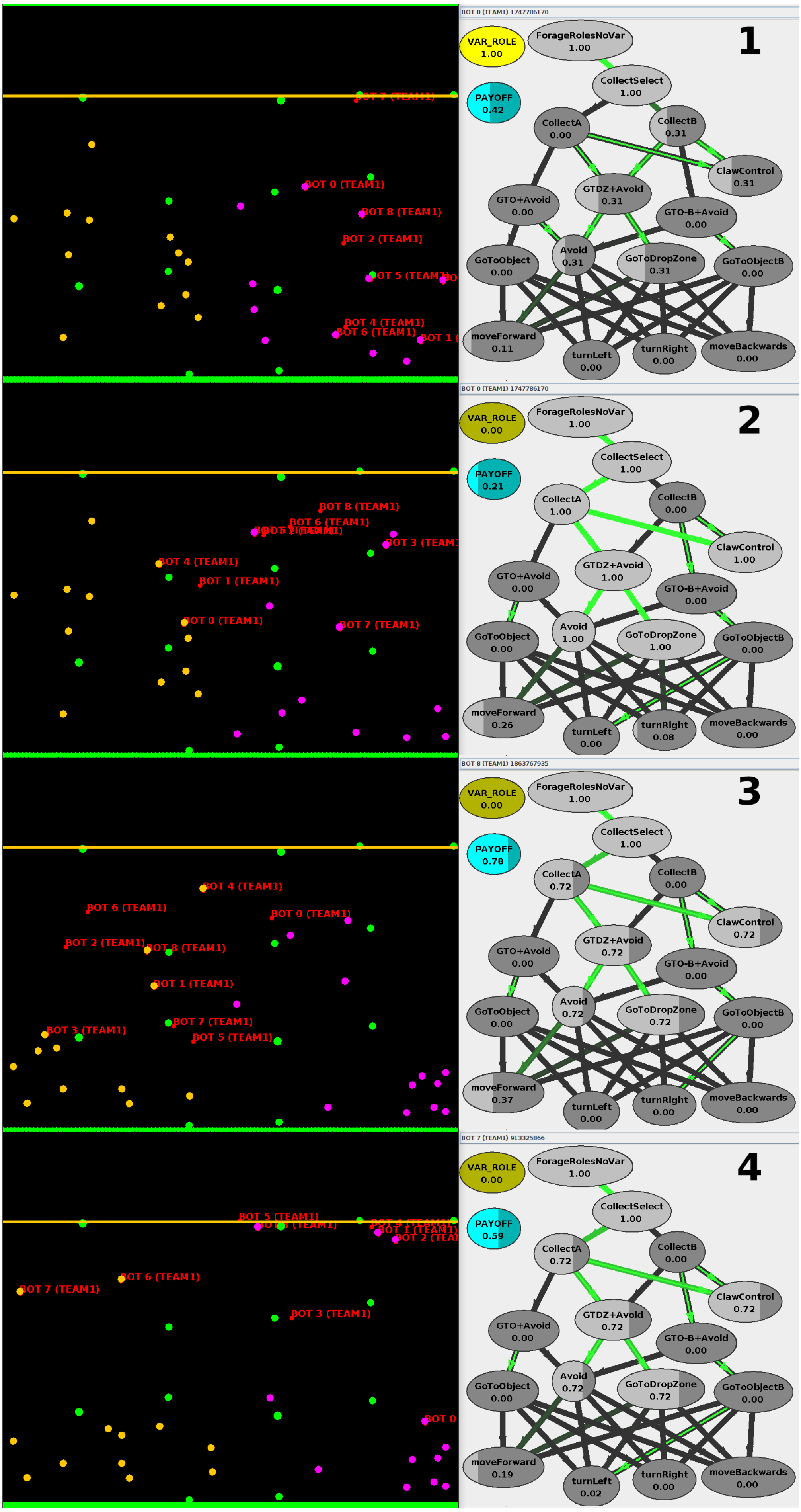


Figure 8.6: Foraging with roles

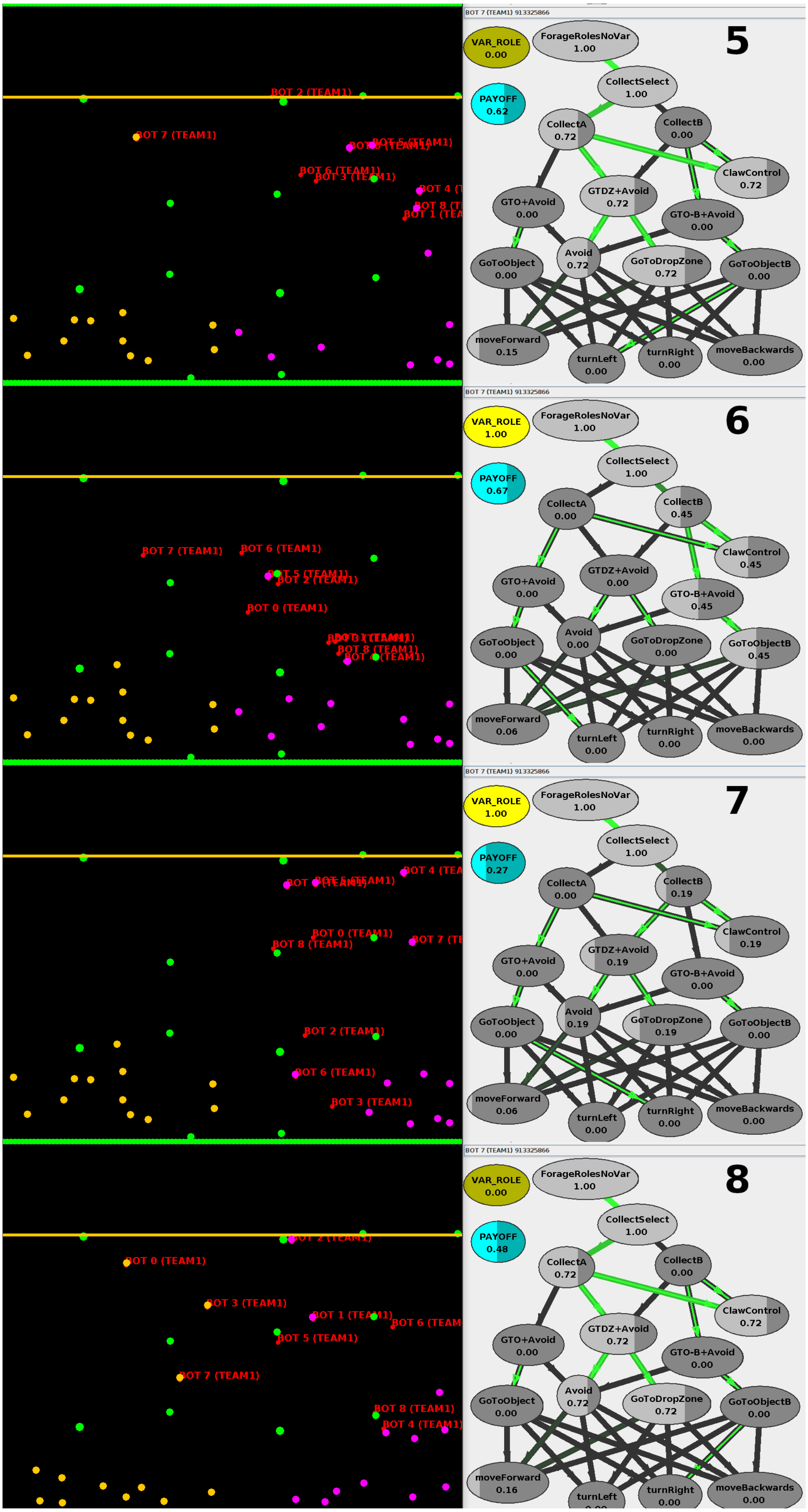


Figure 8.7: Foraging with roles

Step 5: As the payoff increases in the minority group, some agents start to switch back to the majority group.

Step 6: The last agent switched back to role B.

Step 7: The payoff for all agents sinks again, the system will collapse again.

Step 8: After a few cycles, the correct distribution of roles is found by accident and the systems remains stable indefinitely.

8.3 Conclusions on multi-agent applications

This concludes our work with the MIND architecture, covering all aspects of the development of autonomous behaviours in a society of agents, from low level reactive behaviour to social behaviour. The suitability of MIND for developmental agents is made obvious by the fact that, even in this section, the hierarchies are still built upon the original collect hierarchy of the first scenario.

Learning coordination in a homogeneous group of agents means all the agents in the group use the same genome during the evaluation of a population. One of the benefits is that at any given moment the same genome is evaluated as many times as the number of agents present, with multiple points of views. For learning coordination specifically, it also means the same genome is evaluated on both sides of an interaction. This simplifies the evaluation because it is no longer a question of whether each agent has performed its role in the interaction, or even what the roles and form of the interaction are, but simply of whether or not an overall positive result has occurred. This would not be the case with a heterogeneous population, where a failure leads either to punishing both genomes involved in an interaction or to having to determine which of the two caused the failure. It is possible to imagine a competitive task where a competing evaluation is simpler to establish, for instance a fight with elimination.

This question of heterogeneity, especially in the capacities and motivations of agents is an important point for our developmental agent project. Would it be possible to achieve coordination in a heterogeneous group of reactive agents? In the context, for example, of a competing assessment of genomes from different populations in a prey-predator context. Or lead to the emergence of a symbiotic relationship in a cooperative context where heterogeneity induces an initial bias towards the development of a specific role.

In the first scenario, multi-agent coordination is based on reactive agents exchanging simple signals. In the second scenario, agents are still able to form a social organisation without using communication. This is accomplished by using a variable containing the internal representation of their role within the group, a sense of their individuality. Their choice of role comes through observation of the environment, the task in common and their own particular situation. Even if we did not obtain the gradual reinforcement process we expected, the resulting behaviour somehow still achieves a balanced distribution of roles, which could not be attained without individual internal states.

Using MAS allowed us to demonstrate the ability of MIND to organise complex high-level behaviour. Without the use of variables, the highest social behaviour we could conceive was coordination on the level of insect societies. The interaction between these reactive agents is based on their homogeneity, all participants expect to be identical at all time, their role is determined at a species level. Introducing variables allows for more complex social behaviour, the collaboration between individuals. Each individual must determine on his own, during its existence, which role to adopt in order to function within a society. It should now be evident that such high level behaviour is entirely dependant on internal states, and thus justifies the need for variables in the MIND architecture.

Chapter 9

Conclusions

Set in the field of developmental robotics, our work was aimed at bringing together the different components and techniques related to embodied artificial intelligence. Our approach was to provide a modular architecture with a coordination method that is generic and has minimal constraints on the components provided.

We reviewed the domains related to developmental robotics, and the solutions they offer, and given an idea of how these domains are related. Learning techniques and structures, based on machine learning, able to maximize a reward for a task. Motivational systems evaluating benefits of behaviour and driving development. Behavioural hierarchies structuring development by building on previously acquired skills. Memory systems acting at all levels, from low level temporal sequences to high level representation for planning and emergent communication. Social interaction for cooperation, emergent specialization, communication and social learning.

Our preliminary work on multi-agent systems allowed us to evaluate a number of systems involved in emergent social organization of high level agents. Extrinsic motivation was given through a rudimentary system of trade between agents, the social specialization into productive role involved learning through a reinforcement mechanism affecting a simple memory system. This simulation showed the emergent aspect of development, however as a simulation of social behaviour its purpose is to validate given social models, not to generate intelligent behaviour.

Therefore, we introduced MIND, Modular Influence Network Design, an architecture dedicated to developmental agents. The MIND architecture encapsulates sub behaviours into modules and combines them using a generic control signal, the Influence, to form a multi layered hierarchy reflecting the modular and hierarchical nature of complex tasks. This modularity fits many requirements of Ongoing Emergence and facilitates the integration of many learning structures, sensors, actuators and memory systems into a single structure.

In order to demonstrate the potential of our architecture, we implemented MIND into a simulation software, and designed experiments in a 2 dimensional environment. We chose simple multi-layer perceptron as the learning structures for our skills, and trained them using a genetic algorithm, defining our curriculum by designing training environments and carefully crafted fitness functions.

The first series of experiments was designed to test the basic principles of MIND. Learning complex skills requiring simultaneous composition of subskills as well as their mutual exclusion. The advantages of the modularity of MIND were demonstrated by the focused retraining of a skill benefiting the global behaviour, and its ability to extend an existing hierarchy to achieve new goals.

The next step was to experiment with the Variable system of MIND. Storing and retrieving information in the context of sensory input selection, and persistence of information enabling commitment to a task beyond simple reactive behaviour. This mechanism is necessary for the formation of internal representations and the notion of individual, for instance one's role within a group.

We concluded with experiments on social organization, by learning reactive coordination through simple signals and then social specialization through a productive role.

Using the MIND hierarchy with a carefully crafted curriculum and a simple memory system, we made a first attempt at evolving from scratch the social model given in our preliminary multi-agent experiments.

9.1 Contributions

Although this thesis was aimed at creating an architecture supporting agent development, it also helped in validating the MetaCiv meta-model for multi agent simulations, created open source software for the scientific and educational community, and provides a new perspective for future works on machine learning.

9.1.1 Multi agent systems: CogLogo

Our work on the CogLogo plug-in provides an implementation of the MetaCiv meta-model. The main aspect of modelling agent behaviour involves mapping beliefs to action as is the case in architectures such as GOAL (Hindriks, 2009), but with the addition of a stochastic decision process. Other architectures have developed systems for group organisation, using mechanisms such as reputation (Hübner and Vercouter, 2008). Our model uses a simpler mechanism of participation to determine the impact of social organisation on the decision process of individual agents, the degree of participation is left to the judgement of the agent itself. Another benefit of the CogLogo plug-in is its graphical interface used to define models: relations between element are established by links, no modelling language is added and the intimidate graphical representation of elements during and after the modelling process gives a clear view of the model and the relationships between its elements.

The MetaCiv meta-model (Ferber et al., 2014) saw its publication limited by the lack of experiments and in-depth analysis of its practical application. CogLogo was used to validate the MetaCiv meta model by showing its application a social model reliably lead to emergent organization. Multiple run of the same simulation lead to the same balanced organization, but the variety of approaches to equilibrium (initial over-production or under-production of commodities) demonstrate the organic qualities of the model. Through this work, further refinements were made to MetaCiv, such as the reinforcement link mechanism that makes the influence of actions on the mental state explicit in the model.

The CogLogo plug-in for NetLogo itself is open source software distributed under GPL 3 licence, and is available to the community for review or implementation of social models. It is provided with documentation and examples, in the hope to create interest in MetaCiv as tool for sociological research.

CogLogo is available at:
https://gite.lirmm.fr/suro/coglogopublic
https://github.com/suroFr/CogLogo

9.1.2 Developmental agents: MIND

The main contribution of this thesis is the MIND architecture designed specifically for developmental agents. MIND allows the creation of functional hierarchies through composition of behaviour. The modularity of MIND allows evolution of the hierarchy and integration of heterogeneous elements through the encapsulation into modules. Sensors, actuators and memory systems are integrated with skills by the use of the *influence* mechanism. Contrary to other architectures, the *influence* handles all aspects of composition of behaviour, there is no need to specify combinators or particular properties between linked modules. MIND can accommodate any learning technique and is independent of their structure, handcrafted behaviours and programming procedures can be used with various forms of learning structures inside the same MIND hierarchy. MIND can be used for completely handcrafted agent behaviour, offering good software evolutivity by taking advantage of its modular nature, the Influence mechanism making it a signal oriented programming paradigm.

The mechanism of *variables* and its close proximity to the skill hierarchy allow for low level internal representations and memory systems. These variables allow MIND to go beyond simple reactive behaviour by allowing the agent to develop his own internal state, existing independently of the environment. The close proximity of the variables to the skills, and the deliberate choice of a low level signal approach, means variables are an integral part of the skill learning process, which leads to emergent representation developing for these variables. The initial success of the variable system indicates the possibility to use MIND as a structure for the development of cognitive behaviour.

MIND has proven itself as a robust architecture capable of handling multi-level hierarchies, both wide and deep, and able to perform well on complex tasks involving multi-agent behaviours.

The EvoAgents platform itself is open source software distributed under GPL 3 licence, and is available to the community for review or implementation of MIND hierarchies. EvoAgents provides the machine learning methods of the Encog library, both 2D and 3D physics simulation environments, multi-agent support and network socket connection for external programs or remote controlled robots.

EvoAgents is available at:
`https://gite.lirmm.fr/suro/evoagents2020`

9.2 Perspectives

9.2.1 The future for MetaCiv and CogLogo

The results offered by MetaCiv and the CogLogo plug-in are promising, and warrants further trials on social models of greater complexity, closer to actual requirements of the research field.

Another concern for high level multi agent models is the resolution of interactions such as combat or political and trade negotiations. Branching from MetaCiv and its cogniton architecture, we started the development of the Habiliton architecture. *Habilitons* are units of ability or technical knowledge to quantify the ability of an agent. A model is described for an interaction involving an *Instigator*, a *Recipient* and a *Location*, and gives the relations between outcomes and the *Habilitons* of both *Instigator* and *Recipient*. A stochastic decision process determines the outcome in a way similar to selection of a plan in MetaCiv.

HabiLogo is a NetLogo plug-in under development that implements the Habiliton architecture. It can be used alone or in conjunction with CogLogo in any NetLogo models.

9.2.2 Diversification for MIND

We plan to expand the capabilities of MIND, mostly through additions and improvements of the implementation, and confront it to a variety of applications.

We already carried a preliminary experiment on a real world agent, a wheeled robot of a design similar to the on used in the simulated experiments. We designed a simple task using the two base skills *GoToObject* and *Avoid*, and the complex skill (in this case also the master skill) *GoToObject+Avoid*. Our robot used a Single Board Computer[1] running Linux and EvoAgents. Sensory-motor equipment was made from plug and play hobby kit elements[2] and included 10 ultrasonic range finders, a dual motor control board, a 9dof sensor, a basic switch, a servo control card to control a twin servo pan and tilt, a basic webcam and two servos to control the claw. The pan and tilt arm and webcam were used, with the OpenCV computer vision library[3], to search and track a luminous ball. The position of the pan arm was converted to a sensor information replicating the orientation sensor used in the simulation.

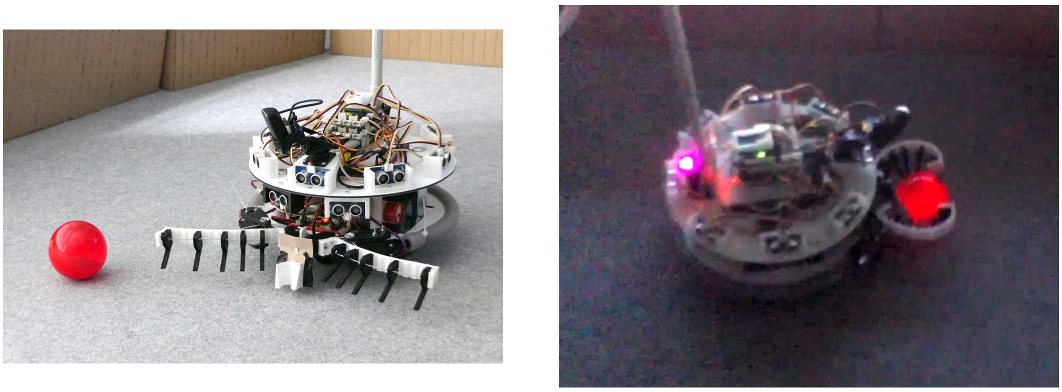


Figure 9.1: Our crude robot, performing the GoToObject + Avoid task

Given that the MIND hierarchy learned in simulation is entirely reactive, and does not require feedback from the actuators as is the case with methods involving planning, we did not anticipate any problem using it on a real world application. Despite the noisy

[1]Raspberry PI3: www.raspberrypi.org/products/raspberry-pi-3-model-b-plus
[2]Grove Pi: www.dexterindustries.com/grovepi
[3]OpenCV computer vision library: www.opencv.org

sensor inputs, the acquisition delays and the imprecision of the actuators, our crude robot demonstrated the expected behaviour using the hierarchy and skills learned in a simulation that was not calibrated in any way to fit the robot or real world environment.

A video of this preliminary experiment is available at the following address :
https://hal.archives-ouvertes.fr/hal-02594407v1

From these preliminary results, we would like to conduct more experiments with robots of higher complexity, which brings us to another challenge for MIND: its application to complex movement, such as humanoid walk. How does a skill involving dozens of actuator reacts to coordination with another skill? Can this complex movement be subdivided into skills, for instance reactively activated to regain balance. Another possible solution is to branch a basic behaviour to specific circumstance by encapsulating this behaviour in a complex skill that applies a correction to the actuators to fit the situation. For instance, a flat terrain walk skill can be the subskill of an uphill walk skill, which will simply apply the corrections needed for the climbing posture.

We also plan to interface MIND with an internal spatial representation, a model of the real world. The objective is twofold: the model can be used to memorize locations and map the environment in a traditional fashion. A number of spatial algorithms can be applied on such a model to improve the performance of the agent, pathing algorithms for instance, or physics prediction of the behaviour of external objects leaving perceptual range. But the main benefit of this internal model is to allow the agent to perform simulations allowing him to improve his skills using the same learning techniques presented in this thesis. Making this model accessible gives an instructor the ability to define reward functions inside the model to guide the agent learning, or even design an entire curriculum in a virtual model based on recorded existing situations.

Another aspect we would like to investigate is how far can we develop the use of the MIND memory systems. Teaching a knowledge representation without allowing the teaching entity direct access to the memory modules immediately benefits the instructor by providing a method that is not dependent on the configuration of the agent being taught. But most importantly, we believe that respecting the barrier of the interiority of the agent's mind will lead to emergent memory representations. The use these memory modules in a connectionist fashion on a situated agent having learning capabilities and emergent memory representation, makes an excellent test-bed for new research on evolving low level cognitive functions from the ground up. This developmental approach could be the long sought for alternative to symbolic A.I.

Finally, inspired the *Culturon* model of MetaCiv, we would like to implement a modified version of the memory system able to automatically synchronize between all agents and experiment with the AGR model on learning MIND Agents. Instead of message exchanges, a shared collective value can be entered as a direct input of a skill, agents experiencing a form of collective consciousness and acting according to their role within the group.

9.2.3 Open ended development

In order to make MIND an efficient tool for open-ended and lifelong development through curriculum learning our next step is the automation or streamlining of the different steps required by the MIND approach. In the first place we will simplify the process for human supervision. From the instructor's point of view, what languages and strategies in formulating a lesson would help the agent understand what is expected of him while at the same time remain natural and simple to define for the human instructor?

Another point is the self organization of the MIND hierarchy, and the automatic definition of skill modules. For a new lesson given by the instructor, with a well-defined goal, an agent using MIND should be able to determine if it requires the creation of a new skill module or the improvement of an existing one. If a new skill module must be created, the agent could either create it from scratch or duplicate an existing one, or use transfer learning methods from several sources. The agent should also be able to determine what are the relevant sensor inputs and actuator outputs or subskills that should be used by the skill to accomplish the given task. Other works already suggested sensori-motor babbling, by randomly activating the actuators, one by one or in groups, the agent can find their relation to the sensors by observing their changes. It might be possible to extend this principle to subskill babbling, and observing relation to relevant sensors, feedback and reward signals.

Finally, if MIND is to be use for open-ended and lifelong development, we would like to study how it handles very large hierarchy. We anticipate that such a developmental agent, being a general purpose AI, would have a wide hierarchy, that is, many skills to be used under different circumstances, and a comparatively shallow depth. While wide hierarchy pose no problems, there could be some concern with deep hierarchies where the successive transfer of influence could result in a form of noisy motor output caused by several unrelated subskill receiving a very small amount of influence. In the context of our experiments, the motor tasks do not require extreme precision, and the hierarchy is not deep enough to show any hint of this phenomenon.

The reason this problem could arise is mainly due to the use of genetically trained artificial neural networks as skill internal functions giving a "good enough answer" (for a given input that should result in a left turn, if the output is 0.99 for left turn and 0.01 for right turn, the impact of right turn is not noticeable). If an influence command is transmitted from master skill to base skill is 1.0 with all other influences remaining at 0.0, then no matter how many complex skills are involved the output will be the exact command of the base skill. What it means in practice is that if noisy outputs become noticeable, it is due to imprecision in the neural networks that should be finely retrained (as the depth of the hierarchy increases, the acceptable imprecision of the internal functions decreases). Should noisy output become a problem and finer training of the internal function be costly and impractical, we would investigate ways to reduce noise in large networks such as filters, threshold layers, or logit functions (invert sigmoid).

9.2.4 Generalization to machine learning: Influence Neural Networks

Following the no free lunch theorem, many improvements to neural networks have been made by making assumption on the nature of the data. For instance convolutional neural networks used for image processing use the spatial proximity of pixel as a basis for the topology of the network.

The module and influence links in the MIND hierarchy are designed for the needs of developmental agents to subdivide tasks into subtasks. However since the inputs, outputs and the influence are all signal in nature, this structure can be generalized as a neural network topology.

The Influence Neural Network (Fig. 9.2) adds the influence operation to neural networks, in effect it is a variable transfer function that feeds from the source input and is trained with the network.

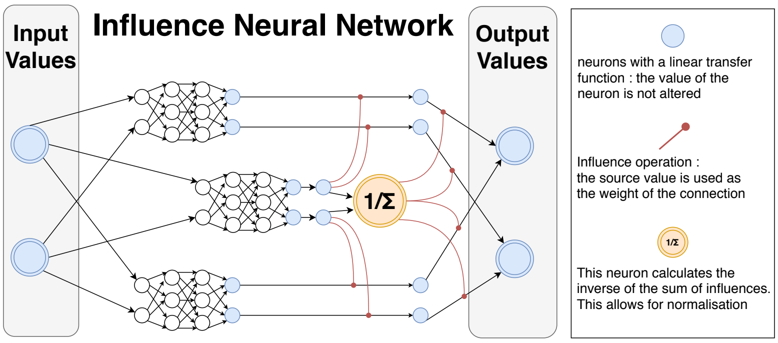


Figure 9.2: Influence Neural Network

An example of application is given in Fig. 9.3 where data is arranged into clusters and sub clusters, with possible overlap. Here the high level network estimates the font used, while each sub network estimates the letter. When a part of the letter could introduce confusion between two fonts (the curve of the B is similar to the curves on the A of the other font), the high level network helps discriminate at the font level (few curves, font 1, many curves font 2).

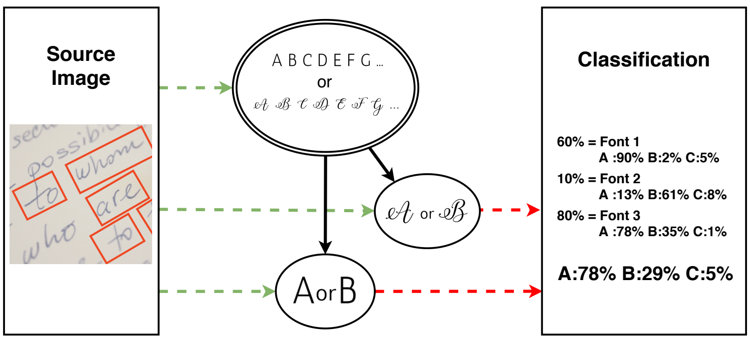


Figure 9.3: A possible use case of the INN

Further investigation of the influence neural network topology is required, specifically around the learning methods suited to this topology.

During early experiments with the MIND hierarchy we tried letting all the skills of an established hierarchy learn (that is, modify their internal functions) at the same time. The outcome was that most of the hierarchy was set to be inactive by not receiving signal from the higher level skills, and only one base skill was active trying to learn all the tasks to perform on its own. As explained in the focus retraining experiment, there needs to be an established bias towards one subtask to perform to start the specialization process of the subskill. This is solved by building complex skills on top of existing base skills.

Understanding this need to create an initial bias in such topology will help adapt the backpropagation algorithm to the Influence Neural Network, probably by initiating a first pass only modifying each subnetwork, thus creating a bias towards a subset of the data, before running a general backpropagation on the whole network, taking into account the influence method in the error propagation.

9.2.5 On the developmental approach to general purpose artificial intelligence

In this thesis, we have presented an underlying architecture using a simple and generic coordination mechanism, independent from learning structure and strategies. This simple system is designed for the accumulation of knowledge, with the goal of not only gain a large repertoire of increasingly complex behaviour, but to lead to the emergence of abstract level operation, following the suggestion of Turing (Turing, 1950).

For any system to behave in a manner similar to human intelligence, it is important to understand for what purpose and under which conditions this intelligence came to be. Embodiment, individuality and social interaction are key elements in shaping this intelligence. Being subjected to physical constrains and fundamental laws of the environment, time, space and causality, perceiving the world through a point of view, having an entirely subjective experience which can only be shared through communication and symbols: all these constraints are the ground which contributed to the emergence of abstract operation and reasoning.

We wish to position ourselves among embodied and developmental research on artificial intelligence, following the development of artificial creatures(Lessin et al., 2015) shaping their behaviour (Dorigo and Colombetti, 1994) or evolving their own language (Varshavskaya, 2002). We set for ourselves an the ambitious long term research project: creating an autonomous artificial agent capable of learning to perform tasks which cannot be anticipated by the designer itself. Much fundamental research is needed in order to achieve this intellectually rewarding goal which of course can never be undertaken by aiming at improvements in existing techniques requested by industrial interests[4].

[4] The class which has the means of material production at its disposal, has control at the same time over the means of mental production [...] as they rule as a class and determine the extent and compass of an epoch, it is self-evident that they do this in its whole range, hence among other things rule also as thinkers, as producers of ideas, and regulate the production and distribution of the ideas of their age: thus their ideas are the ruling ideas of the epoch. (Marx and Engels, 1848)

Contrairement aux rêveries des spectateurs de l'histoire, quand ils essaient de s'établir stratèges à Sirius, ce n'est pas la plus sublime des théories qui pourrait jamais garantir l'évènement ; tout au contraire, c'est l'évènement réalisé qui est le garant de la théorie. De sorte qu'il faut prendre des risques, et payer au comptant pour voir la suite.

Contrary to the daydreams of the spectators of history, when they try to establish themselves strategists at Sirius, it is not the most sublime of theories that could ever guarantee the event; on the contrary, it is the realized event that is the guarantee of the theory. So that one has to take risks, and pay cash to see what happens next.

G. Debord

Bibliography

Gary An, Qi Mi, Joyeeta Dutta-Moscato, and Yoram Vodovotz. Agent-based models in translational systems biology. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine*, 1(2):159–171, 2009.

Ronald C Arkin and Tucker Balch. Aura: Principles and practice in review. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):175–189, 1997.

Andrew G Barto, Satinder Singh, and Nuttapong Chentanez. Intrinsically motivated learning of hierarchical collections of skills. In *Proc. 3rd Int. Conf. Development Learn*, pages 112–119, 2004.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 41–48. ACM, 2009.

Nikolai Bernstein. *The Co-ordination and Regulation of Movements*. 1967.

Gregory Beurier, Olivier Simonin, and Jacques Ferber. Model and simulation of multi-level emergence. In *Proceedings of IEEE ISSPIT*, pages 231–236, 2002.

François Bousquet and Christophe Le Page. Multi-agent simulations and ecosystem management: a review. *Ecological modelling*, 176(3-4):313–332, 2004.

Valentino Braitenberg. *Vehicles: Experiments in synthetic psychology*. MIT press, 1986.

Rodney A Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

Rodney A Brooks, Pattie Maes, Maja J Mataric, and Grinell More. Lunar base construction robots. In *EEE International Workshop on Intelligent Robots and Systems, Towards a New Frontier of Applications*, pages 389–392. IEEE, 1990.

Davide Calvaresi, Mauro Marinoni, Arnon Sturm, Michael Schumacher, and Giorgio Buttazzo. The challenge of real-time multi-agent systems for enabling iot and cps. In *Proceedings of the international conference on web intelligence*, pages 356–364. ACM, 2017.

Nicolas Carlési. *Coopération entre véhicules sous-marins autonomes : une approche organisationnelle réactive multi-agent*. Theses, Université de Montpellier 2, 2013. URL https://hal.archives-ouvertes.fr/tel-01213353.

Dongkyu Choi and Pat Langley. Evolution of the icarus cognitive architecture. *Cognitive Systems Research*, 48:25–38, 2018.

Hoang Nam Chu, Arnaud Glad, Olivier Simonin, Francois Sempe, Alexis Drogoul, and François Charpillet. Swarm approaches for the patrolling problem, information propagation vs. pheromone evaporation. In *19th IEEE international conference on tools with artificial intelligence (ICTAI 2007)*, volume 1, pages 442–449. IEEE, 2007.

Jerome T Connor, R Douglas Martin, and Les E Atlas. Recurrent neural networks and robust time series prediction. *IEEE transactions on neural networks*, 5(2):240–254, 1994.

Kenneth A De Jong. Are genetic algorithms function optimizers? In *PPSN*, volume 2, pages 3–14, 1992.

Jean-Louis Deneubourg and Simon Goss. Collective patterns and decision-making. *Ethology Ecology & Evolution*, 1(4):295–311, 1989.

Jean-Louis Deneubourg, Simon Goss, and al. The dynamics of collective sorting robot-like ants and ant-like robots. In *1st international conference on simulation of adaptive behavior on From animals to animats*, pages 356–363, 1991.

Coline Devin, Abhishek Gupta, Trevor Darrell, Pieter Abbeel, and Sergey Levine. Learning modular neural network policies for multi-task and multi-robot transfer. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2169–2176. IEEE, 2017.

Razvan Dinu, Tiberiu Stratulat, and Jacques Ferber. A formal model of agent interaction based on MASQ. In *International Workshop on Agent-based Modeling for Policy Engineering (AMPLE 2012)*, pages 66–74, 2012.

Marco Dorigo and Marco Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial intelligence*, 71(2):321–370, 1994.

Marco Dorigo and Marco Colombetti. *Robot shaping: an experiment in behavior engineering.* MIT press, 1998.

Jacques Ferber. *Les Systèmes Multi-agents, Vers une intelligence collective.* InterEditions, Paris, 1995.

Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations: an organizational view of multi-agent systems. *Agent-Oriented Software Engineering (AOSE) IV*, 2004.

Jacques Ferber, Fabien Michel, and José-Antonio Báez-Barranco. {AGRE} : Integrating Environments with Organizations. In *Environments for Multi-agent Systems*, volume 3374 of *Lecture Notes in Computer Science*, pages 48–56. Springer, 2005. ISBN 3-540-24575-8. doi: 10.1007/b106134.

Jacques Ferber, Tiberiu Stratulat, and John Tranier. Towards an integral approach of organizations in multi-agent systems: the MASQ approach. *Multi-agent Systems: Semantics and Dynamics of Organizational Models, IGI*, 2009.

Jacques Ferber, Julien Nigon, Gautier Maille, Tristan Seguin, Sven Holtz, and Tiberiu Stratulat. De masq à metaciv: un cadre générique pour modéliser des sociétés humaines dans une approche transdisciplinaire, 2014.

Leon Festinger. Cognitive dissonance. *Scientific American*, 207(4):93–106, 1962.

Francesco Foglino, Christiano Coletto Christakou, and Matteo Leonetti. An optimization framework for task sequencing in curriculum learning. In *2019 Joint IEEE 9th International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, pages 207–214. IEEE, 2019.

Mitsuo Gen and Lin Lin. Genetic algorithms. *Wiley Encyclopedia of Computer Science and Engineering*, pages 1–15, 2007.

Simon Goss and Jean-Louis Deneubourg. Harvesting by a group of robots. In *Proceedings of the First European Conference on Artificial Life*, pages 195–204, 1992.

Çaglar Gülçehre, Marcin Moczulski, Francesco Visin, and Yoshua Bengio. Mollifying networks. *CoRR*, abs/1608.04980, 2016. URL http://arxiv.org/abs/1608.04980.

Olivier Gutknecht. *Proposition d'un modèle organisationnel générique de systèmes multi-agents et examen de ses conséquences formelles, implémentatoires et méthologiques.* PhD thesis, Université Montpellier II-Sciences et Techniques du Languedoc, 2001.

Olivier Gutknecht, Jacques Ferber, and Fabien Michel. Madkit : Une expérience d'architecture de plate-forme multi-agent générique. In Sylvie Pesty and Claudette Sayettat-Fau, editors, *Systèmes multi-agents : Méthodologie, technologie et expériences - JFIADSMA 00 - huitième journées francophones d'Intelligence Artificielle et systèmes multi-agents*, pages 223–236. Hermès Lavoisier Editions, 2000. URL http://www.lavoisier.fr/notice/fr2746201760.html.

Michael Alexander Kirkwood Halliday. Learning how to mean. In *Foundations of language development*, pages 239–265. Elsevier, 1975.

Nicolas Heess, Gregory Wayne, Yuval Tassa, Timothy P. Lillicrap, Martin A. Riedmiller, and David Silver. Learning and transfer of modulated locomotor controllers. *CoRR*, abs/1610.05182, 2016.

Michel Hersen. *Encyclopedia of Behavior Modification and Cognitive Behavior Therapy: Volume I: Adult Clinical Applications Volume II: Child Clinical Applications Volume III: Educational Applications.* Sage Publications, 2005.

Todd Hester and Peter Stone. Real time targeted exploration in large domains. In *2010 IEEE 9th International Conference on Development and Learning*, pages 191–196. IEEE, 2010.

Todd Hester and Peter Stone. Intrinsically motivated model learning for a developing curious agent. In *2012 IEEE international conference on development and learning and epigenetic robotics (ICDL)*, pages 1–6. IEEE, 2012.

Todd Hester and Peter Stone. Intrinsically motivated model learning for developing curious robots. *Artificial Intelligence*, 247:170–186, 2017.

Koen V Hindriks. Programming rational agents in goal. In *Multi-agent programming*, pages 119–157. Springer, 2009.

John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

Bing-Qiang Huang, Guang-Yi Cao, and Min Guo. Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance. In *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, volume 1, pages 85–89. IEEE, 2005.

Clark Leonard Hull. Principles of behavior: An introduction to behavior theory. 1943.

Jomi Hübner and Laurent Vercouter. Instrumenting multi-agent organisations with reputation artifacts. *AAAI Workshop - Technical Report*, 01 2008.

Kinjal Jadav and Mahesh Panchal. Optimizing weights of artificial neural networks using genetic algorithms. *Int J Adv Res Comput Sci Electron Eng*, 1(10):47–51, 2012.

George Konidaris. Value function approximation in reinforcement learning using the fourier basis. 2008.

George Konidaris and Andrew G Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in neural information processing systems*, pages 1015–1023, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Yasuo Kuniyoshi, Yasuaki Yorozu, Masayuki Inaba, and Hirochika Inoue. From visuomotor self learning to early imitation-a neural architecture for humanoid learning. In *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, volume 3, pages 3132–3139. IEEE, 2003.

John E Laird. Extending the soar cognitive architecture. *Frontiers in Artificial Intelligence and Applications*, 171:224, 2008.

Pat Langley and Dongkyu Choi. A unified cognitive architecture for physical agents. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1469. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

Pat Langley, Dongkyu Choi, and Seth Rogers. Acquisition of hierarchical reactive skills in a unified cognitive architecture. *Cognitive Systems Research*, 10(4):316–332, 2009.

Tobias Larsen and Søren Tranberg Hansen. Evolving composite robot behaviour-a modular architecture. In *Proceedings of the Fifth International Workshop on Robot Motion and Control, 2005. RoMoCo'05.*, pages 271–276. IEEE, 2005.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436–444, 2015.

François Legras, Arnaud Glad, Olivier Simonin, and François Charpillet. Authority sharing in a swarm of uavs: Simulation and experiments with operators. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 293–304. Springer, 2008.

Paulo Leitão, José Barbosa, and Damien Trentesaux. Bio-inspired multi-agent systems for reconfigurable manufacturing systems. *Engineering Applications of Artificial Intelligence*, 25(5):934–944, 2012.

Dan Lessin, Don Fussell, and Risto Miikkulainen. Open-ended behavioral complexity for evolved virtual creatures. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 335–342, 2013.

Dan Lessin, Don Fussell, Risto Miikkulainen, and Sebastian Risi. Increasing behavioral complexity for evolved virtual creatures with the esp method. *arXiv preprint arXiv:1510.07957*, 2015.

Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *arXiv preprint arXiv:1504.00702*, 2015a.

Sergey Levine, Nolan Wagener, and Pieter Abbeel. Learning contact-rich manipulation skills with guided policy search. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 156–163. IEEE, 2015b.

Weiwei Lin, Ziming Wu, Longxin Lin, Angzhan Wen, and Jin Li. An ensemble random forest algorithm for insurance big data analysis. *Ieee Access*, 5:16568–16575, 2017.

Manuel Lopes and Pierre-Yves Oudeyer. The strategic student approach for life-long exploration and learning. In *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, pages 1–8. IEEE, 2012.

Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

Max Lungarella and Luc Berthouze. On the interplay between morphological, neural, and environmental dynamics: A robotic case study. 2002.

Max Lungarella, Giorgio Metta, Rolf Pfeifer, and Giulio Sandini. Developmental robotics: a survey. *Connection Science*, 15(4):151–190, 2003.

Douglas C MacKenzie, Ronald C Arkin, and Jonathan M Cameron. Multiagent mission specification and execution. In *Robot colonies*, pages 29–52. Springer, 1997.

Karl Marx and Friedrich Engels. *The German Ideology.* 1848.

Andrew N Meltzoff and M Keith Moore. Explaining facial imitation: A theoretical model. *Infant and child development*, 6(3-4):179–192, 1997.

Ghezlane Halhoul Merabet, Mohammed Essaaidi, Hanaa Talei, Mohamed Riduan Abid, Nacer Khalil, Mohcine Madkour, and Driss Benhaddou. Applications of multi-agent systems in smart grids: A survey. In *2014 International conference on multimedia computing and systems (ICMCS)*, pages 1088–1094. IEEE, 2014.

Fabien Michel. *Approches environnement-centrées pour la simulation de systèmes multi-agents. Pour un déplacement de la complexité des agents vers l'environnement.* PhD thesis, Université de Montpellier, 2015.

Marvin Minsky. *Society of mind.* Simon and Schuster, 1988.

Jose Luis Morales, Pedro Sánchez, and Diego Alonso. A systematic literature review of the teleo-reactive paradigm. *Artificial Intelligence Review*, 42(4):945–964, 2014.

Jean-Pierre Müller. Emergence of collective behaviour and problem solving. In *Engineering Societies in the Agents World IV*, pages 1–20. Springer, 2004.

Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 566–574. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

Nils Nilsson. Teleo-reactive programs for agent control. *Journal of artificial intelligence research*, 1:139–158, 1993.

Andrew M Nuxoll and John E Laird. Extending cognitive architecture with episodic memory. *Ann Arbor*, 1001:48109–2121.

Pierre-Yves Oudeyer. Developmental robotics. In *Encyclopedia of the Sciences of Learning*, pages 969–972. Springer, 2012.

Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? a typology of computational approaches. *Frontiers in Neurorobotics*, 1(6), 2007.

Iñaki Fernández Pérez, Amine Boumaza, and François Charpillet. Learning collaborative foraging in a swarm of robots using embodied evolution. In *Artificial Life Conference Proceedings 14*, pages 162–161. MIT Press, 2017.

Jean Piaget. *The construction of reality in the child.* Basic Books, New York, 1954.

Jean Piaget and Eleanor Duckworth. Genetic epistemology. *American Behavioral Scientist*, 13(3):459–480, 1970.

Christopher Prince, Nathan Helder, and George Hollich. Ongoing emergence: A core concept in epigenetic robotics. 2005.

Mitchel Resnick. Learning about life. *Artificial life*, 1(1_2):229–241, 1993.

Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH computer graphics*, volume 21, pages 25–34. ACM, 1987.

Matthew Robbins. Neural-network. https://github.com/matthewrdev/Neural-Network, 2014.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Günter Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE transactions on neural networks*, 5(1):96–101, 1994.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 0136042597, 9780136042594.

R Keith Sawyer. Artificial societies: Multiagent systems and the micro-macro link in sociological theory. *Sociological methods & research*, 31(3):325–363, 2003.

Jacob Schrum and Risto Miikkulainen. Discovering multimodal behavior in ms. pac-man through evolution of modular neural networks. *IEEE transactions on computational intelligence and AI in games*, 8(1):67–81, 2015.

Olivier Simonin and Jacques Ferber. Modeling self satisfaction and altruism to handle action selection and reactive cooperation. In *Proceedings of the 6th International Conference on the Simulation of Adaptive Behavior*, volume 2, pages 314–323, 2000.

Justin Sirignano, Apaar Sadhwani, and Kay Giesecke. Deep learning for mortgage risk. *arXiv preprint arXiv:1607.02470*, 2016.

Burrhus Frederic Skinner. *Science and human behavior.* Number 92904. Simon and Schuster, 1965.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Peter Stone and Manuela Veloso. Layered learning. In *European Conference on Machine Learning*, pages 369–381. Springer, 2000.

Tiberiu Stratulat. *Systèmes d'agents normatifs: concepts et outils logiques*. PhD thesis, Université de Caen, 2002.

Tiberiu Stratulat, Jacques Ferber, and John Tranier. MASQ: towards an integral approach to interaction. *Proc of 8th Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 813–820, 2009.

F. Suro. Coglogo. https://github.com/suroFr/CogLogo, 2017. SMILE: Système Multiagent, Interaction, Langage, Evolution., Laboratoire d'informatique, de robotique et de microélectronique de Montpellier., France.

François Suro, Thibaut Castanié, and Vincent Bazia. Project robot-sapiens. https://github.com/Ooya/Robot-Sapiens, 2015.

Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2011.

Alan M Turing. Computing machinery and intelligence. 1950.

Annelinde RE Vandenbroucke, Ilja G Sligte, Adam B Barrett, Anil K Seth, Johannes J Fahrenfort, and Victor AF Lamme. Accurate metacognition for visual sensory memory representations. *Psychological science*, 25(4):861–873, 2014.

Paulina Varshavskaya. Behavior-based early language development on a humanoid robot. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 2002.

Yibo Wang and Wei Xu. Leveraging deep learning with lda-based text analytics to detect automobile insurance fraud. *Decision Support Systems*, 105:87–95, 2018.

Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks, 2014.

Steven D Whitehead and Long Ji Lin. Reinforcement learning in non-markov environments. *Artificial Intelligence. Submitted*, 1993.

Shimon Whiteson, Nate Kohl, Risto Miikkulainen, and Peter Stone. Evolving keepaway soccer players through task decomposition. In *Genetic and Evolutionary Computation Conference*, pages 356–368. Springer, 2003.

Bernard Widrow and Michael A Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

Ken Wilber. *A Theory of Everything: An Integral Vision for Business, Politics, Science and Spirituality.* Shambhala Publications, 2001.

U. Wilensky. Netlogo. http://ccl.northwestern.edu/netlogo/, 1999. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

Joseph Winnick and David L Porretta. *Adapted physical education and sport.* Human Kinetics, 2016.

Xiaolin Wu and Xi Zhang. Automated inference on criminality using face images. *ArXiv*, abs/1611.04135, 2016.

Jordan Zlatev and Christian Balkenius. Introduction: Why epigenetic robotics ? 2001.

Chapter A

Appendix

A.1 EvoAgents: Defining sensors and actuators

The *<MyAgentName>.botdesc* describes the sensors and actuators available to the MIND hierarchy. It must start with the keyword SENSORS, list all the sensors, one sensor per line with the format:

sensorName:sensorType(:optionalParameters)

Then the keyword ACTUATORS, list all the actuators, one actuator per line with the format *actuatorName:actuatorType*. (note: the types are not used in simulation mode, only for remote bot control, it is still a good thing to write what it's supposed to do ...)

The names of the sensors and actuators are the ones used when defining a skill.

Example:

```
SENSORS
S1:proxSensor
S2:proxSensor
TARG_AZIM:TargetSensor:1
TARG_ELEV:TargetSensor:2
ACTUATORS
M1:propeller
M2:propeller
```

A.2 EvoAgents: Defining variables

The *<MyAgentName>.vardesc* describes the variables available to the MIND hierarchy. List one variable per line with the format:

variableName:variableType:optionalDefaultValue.

So far there are 3 variable types:

- VariableModule: Used as an input it will provide its current value. Used as an output it will store the value outputted (using the MIND principle of influence as if it were an actuator).
- TickWaveSinVariableModule: Used as an input it will provide the value of *sin(t)* normalized between 0 and 1, with t the tick (the number of times the MIND hierarchy has been asked to process inputs). Used as an output it will vary the frequency of the sin wave (1.0 = t*3 , 0.0 = t/1000).
- CounterVariableModule: Used as an input it will provide its current count from 0 to 10 (as 0.0,0.1,0.2,...0.9,1.0). Used as an output on a rising edge above 0.8, it will increment, on a falling edge under 0.2 it will reset to 0. Overflow resets to 0.

The names of the variables are the ones used when defining a skill.

example:

```
VAR_SIN:TickWaveSinVariableModule
VAR_BASE:VariableModule
#testComment
VAR_COUNT:CounterVariableModule
VAR_ROLE:VariableModule:0.5
```

A.3 EvoAgents: Defining skills

A skill is defined by a folder named after it (ex: MySkillName). This folder contains a skill description file using the *.ades* extension, named after the skill (MySkillName.ades).

The structure is as follows:

```
*MySkillName*
input:*inputcount*
*inputType*:*inputName*:*optionalParameter*
*inputType*:*inputName*:*optionalParameter*
...
output:*outputcount*
*outputType*:*outputName*
*outputType*:*outputName*
*outputType*:*outputName*
...
*internalFunctionType*
*internalFunctionParameter*:*internalFunctionParameterValue*
*internalFunctionParameter*:*internalFunctionParameterValue*:...
...
```

The skill folder will also contain any files required by the internal function and the learning process, such as compiled Java classes or neural networks persistence files.

A.3.1 Keywords for the skill description file

Input types Inputs can provide their value, a derivative of their value or an *history* value which is the value of the input a number of steps in the past (given as the optional parameter).

- SE: a sensor's value
- SD: a sensor's derivative
- SH: a sensor's history
- VA: a variable's value
- VD: a variable's derivative
- VH: a variable's history

Output types

- MO: transmit a command to an actuator
- SK: transmit influence to a subskill
- VA: transmit a value to a variable

Internal function types

- neural: A neural network. Its only parameter is *layers* which defines the number of hidden layers to use (note: the layer parameter is not used by the NEAT algorithm which defines the network topology on its own).
- hardCoded: A Java procedure. Its only parameter is *class* which defines which Java class to use.

Neural Network Parameters

TYPE: the type of network used

- BASIC: Multi-Layered Perceptron.
- CNN: Convolutional Neural Network.
- ELMAN: Elman pattern of reccurent network.
- RNN: Reccurent Neural Network.

Other parameters (if applicable)

- LAYERS: the number of hidden layers.
- HIDDEN_TRANSFER: the transfer function of the hidden layers, ReLu by default, can be set to SIGMOID.
- HIDDEN_NEURON_ADDED: the number of neurons added to the hidden layers after the MAX(inputs, outputs) count.

Parameters for the CNN The Convolutional Neural Network can define as many convolution layers as needed, connected in any fashion. The last convolution layer is connected as input to a basic Multi-Layered Perceptron, along with inputs selected to bypass the convolution layers.

- CONVO_LAYER:<INT>:<INT> defines a new convolution layer and its kernel. The first number is the number of inputs of the kernel, the second number is the number of neurons in the layer.
- CONVO_LINK:<INT>:<INT>:N*<INT> links the inputs to the convolution layer. The first number is the kernel to use (in the order declared, starts at 0). The second number is the neuron which is defined (starts at 0). The remaining numbers are the input (in the order declared, starts at 0), as many as the kernel requires. The first layer uses the global input, the following layers use the neurons of the previous layer.
- CONVO_BYPASS:N*<INT> which inputs are sent to the fully connected network without going through the convolution layers.

A.3.2 examples

A.4 EvoAgents: Defining tasks

Task files use the extension *.simbatch*. Parameters can be listed in any order and superfluous parameters will be ignored (such as defining which learning method to use in a demo task). Here is an example task file to learn an *Avoid* behaviour in a 2d simulator.

```
TYPE:LEARNSIM
BOTMODEL:BotType6
BOTNAME:T6-NEAT
MASTERSKILL:Avoid
LEARNINGSKILL:Avoid
```

```
Collect
input:2
SE:SENSOBJ
SE:SENSDZ
output:3
SK:GTO+Avoid
SK:GTDZ+Avoid
SK:ClawControl
neural
layers:2
```

```
moveForward
input:0
output:2
MO:MotL
MO:MotR
hardCoded
class:MoveForward
```

```
Avoid
input:15
SE:US_1
SE:US_2
SE:US_3
SE:US_4
SE:US_5
SE:US_6
SE:US_7
SE:US_8
SE:US_9
SE:US_10
SD:US_10
SD:US_1
SD:US_2
VA:VAR_SIN
SE:DOF
output:4
SK:moveForward
SK:turnLeft
SK:turnRight
SK:moveBackwards
neural
layers:2
```

Figure A.1: 3 skill description, from left to right: **Collect** a complex skill. **moveForward** a base skill, hard coded and using no input.**Avoid** a complex skill.

```
Avoid
input:36
SE:S1
SH:S1:5
SH:S1:10
SE:S2
SH:S2:5
SH:S2:10
SE:S3
/ ... /
SE:S8
SH:S8:5
SH:S8:10
SE:S9
SH:S9:5
SH:S9:10
SE:S10
SH:S10:5
SH:S10:10
SD:S1
SD:S2
SD:S10
SE:DOF
SD:DOF
VA:VAR_TIME
output:4
SK:moveForward
SK:turnLeft
SK:turnRight
SK:moveBackwards
neural
TYPE:CNN
LAYERS:2
CONVO_LAYER:3:10
CONVO_LAYER:3:10
CONVO_BYPASS:30:31:32:33:34:35
CONVO_LINK:0:0:0:1:2
CONVO_LINK:0:1:3:4:5
CONVO_LINK:0:2:6:7:8
CONVO_LINK:0:3:9:10:11
CONVO_LINK:0:4:12:13:14
CONVO_LINK:0:5:15:16:17
CONVO_LINK:0:6:18:19:20
CONVO_LINK:0:7:21:22:23
CONVO_LINK:0:8:24:25:26
CONVO_LINK:0:9:27:28:29
CONVO_LINK:1:0:0:1:2
CONVO_LINK:1:1:1:2:3
CONVO_LINK:1:2:2:3:4
CONVO_LINK:1:3:3:4:5
CONVO_LINK:1:4:4:5:6
CONVO_LINK:1:5:5:6:7
CONVO_LINK:1:6:6:7:8
CONVO_LINK:1:7:7:8:9
CONVO_LINK:1:8:8:9:0
CONVO_LINK:1:9:9:0:1
```

Figure A.2: **Avoid** in a Convolutional Neural Network version, abridged (shown in Fig. 5.6).

```
ENV:EXP_Avoid
LEARNINGMETHOD:NEAT
ROOTFOLDER:Minds
NUMGEN:50
EVALREP:3
TICKLIM:100000
```

A.4.1 Keywords for the task description file

- TYPE: the type of task to run, see the task types chapter
- BOTMODEL: which built-in bot model to use for the simulation, correspond exactly the name of the Java class that describes the agent. (see simulated bot body chapter)
- BOTNAME: the name of the agent, corresponds to the mind folder name (MyAgentName, see A Mind Folder first chapters)
- MASTERSKILL: the name of the skill that will run the MIND hierarchy (corresponds to the skill folder/ skill file name, see skills chapter)
- LEARNINGSKILL: the name of the skill that will be affected by the learning algorithm (corresponds to the skill folder/ skill file name, see skills chapter)
- ENV: name of the build-in environment to use correspond exactly the name of the java class that describes the environment. (see simulation environment chapter)
- LEARNINGMETHOD: the learning method to use. Choices are NEAT (the NEAT algorithm, multi-threaded, support resume training), GENETIC (a basic genetic algorithm, multi-threaded, does not support resume training), ANNEALING (simulated annealing, single threaded, does not support resume training)
- ROOTFOLDER: relative path to the .jar where the mind folder is located (see a mind folder chapter)
- NUMGEN: the number of generation/iterations to run.
- EVALREP: how many times the same agent is evaluated, final score is the sum of each evaluation. (to minimize the impact of random elements in the environment)
- TICKLIM: the tick limit for an evaluation.

A.4.2 Task types

The EvoAgentApp can perform a number of task and is left open to add more. It is recommended to stick with DEMOSIM and LEARNSIM, using 2d environments.

2D environments

The 2d environment uses the JBox2d physics library for collision, actuator forces, sensor raytracing...

DEMO2DSIM will run your agent in real time in the environment specified and provide you with a viewer to observe his behaviour.

LEARN2DSIM will run a multi-threaded learning algorithm using the environment and its reward function. Here the simulation is ran as fast as the processor is able and no viewer are provided. Progress of the learning algorithm will be displayed.

LEARN2DSIMCONTINUOUS is meant for retraining in final context. All of the skills of the hierarchy will be successively trained in the specified environment, in an infinite loop. (see the MIND article for retraining in a broader context).

Multi agent environments

The 2d multi agent environment uses the JBox2d physics library for collision, actuator forces, sensor raytracing. It also manages multiple teams of multiple agents.

DEMO2DSIMMULTI will run multiple teams of agents in real time in the environment specified and provide you with a viewer to observe their behavior. Each team can have a different configuration of body and MIND hierarchy.

LEARN2DSIMMULTI will run a multi-threaded learning algorithm using the environment and its reward function. Here the simulation is ran as fast as the processor is able and no viewer are provided. Progress of the learning algorithm will be displayed.

3D environments

The 3d environment uses the JBullet physics library for collision, actuator forces, sensor raytracing... This is a work in progress. The tasks keywords are **DEMO3DSIM** and **LEARN3DSIM**.

Remote environments

It is possible to create your own simulation environment in a separate program, for instance the Unity3D game engine. EvoAgentApp will use a network socket to contact your program (which should be setup as a server). This is also the technique used to control the robot of the MIND project. The tasks keywords are **DEMOREMOTE** for the robot, **DEMOREMOTESIM** for a custom simulation and **LEARNREMOTESIM** (not tested).

Open ended development

OPENENDED the holy grail ... not quite there yet.

A.5 EvoAgents: Defining custom simulation elements for the 2D environment (Java programming)

This section will cover the creation of custom agents, environments and reward functions for the 2D physics environment. The *Simulated bot body* section will explain the creation of an agent using existing sensors and actuators. The *Simulation environment* section will explain the creation of the environment with pre-existing world elements. *Reward functions* will explain the design of reward function used by learning algorithms. *Control functions* will explain the creation of functions used to interrupt or reset the simulation which plays an indirect role in the learning process. *Agent sensors* and *Agent actuators* will explain the creation of custom sensors and actuators *World elements* will explain the creation of static and mobile obstacles, target objects, trigger zones ...

A.5.1 Simulated bot body

Create a new class in the *evoagent2dsimulator.bot* package. The name of the class will be the name of type of the agent (the BOTMODEL parameter of a task). Your agent class must extend the *BotBody* class. You're free to override any function you wish (at your own perils). The description of the agent is done in the class constructor.

First, define the physical form of the agent, using the JBox2d library. Create a new *FixtureDef* in the *sd* field (sd: shape definition). The *sd* variable will define the shape and collision layers (see JBox2d documentation for more information) Create a new *BodyDef* in the *bd* field (bd: body definition). The *bd* variable will define the dynamic properties of the agent (see JBox2d documentation for more information)

Example:

```
sd = new FixtureDef();
sd.shape = new CircleShape();
sd.shape.m_radius = (float)size;
//sd.friction = 1.0f;
sd.density = 2.0f;
sd.filter.categoryBits = CollisionDefines.CDBot;
sd.filter.maskBits = CollisionDefines.CDAllMask;
bd = new BodyDef();
bd.type = BodyType.DYNAMIC;
bd.angularDamping = 20.0f;
bd.linearDamping = 5.0f;
bd.allowSleep = false;
```

Then, add sensors and actuators to the agent in the *sensors* and *actuators* hashmaps. The key for the hashmap must correspond to the elements described in the agent description file (see MyAgentName.botdesc chapter) Each sensor and actuators has specific parameters, but usually start by a Vec2 representing the x,y coordinates relative to the agent body and a double representing the angle relative to the agent front orientation.

Example:

```
sensors.put("US_1",new S_ProximityArcSensor(new Vec2((float)size,0.0f),0, this,
    12.0,0.8));
sensors.put("SENSOBJB",new S_ObjectDetector(new Vec2((float)size,0),0,
    this,null,2.0));
```

```
sensors.put("SENSVZ",new S_ZonePresence(new Vec2(0,0),0, this,null));

actuators.put("MotL",new A_Wheel(new Vec2(0.0f,-(float)size),0, this,80.0f));
actuators.put("MotR",new A_Wheel(new Vec2(0.0f,(float)size),0, this,80.0f));
actuators.put("EMAG",new A_AutoClaw(new Vec2((float)size,0f),0, this,1.5f));
```

A.5.2 Agent sensors

Sensors must extend the *Sensor* class and must define their *getNormalizedValue* and *reset* behaviour. The class name should start with *S_*.

getNormalizedValue must return a real value int the [0.0 , 1.0] range

computeWorldPosAndAngle() must be called before using the sensor's position in calculations (this will update the position with the translations and rotations of the bot's body).

```
//get the distance to a target, normalize bewteen [0,MaxDistance], 1.0 if over
    MaxDistance.
public class S_Distance extends Sensor{
  public VirtualWorldElement target = null;
  private double maxDist;

  public S_Distance(Vec2 lp, float la, BotBody b, VirtualWorldElement targetin,
      double maxD) {
    super(lp, la, b);
    target = targetin;
    maxDist = maxD;
  }

  public double getValue() {
    if(target != null){
      computeWorldPosAndAngle();
      Vec2 vec =
        new Vec2(target.getWorldPosition().x-worldPosition.x,
            target.getWorldPosition().y-worldPosition.y);
      return vec.length();
    }
    else
      return maxDist;
  }

  @Override
  public double getNormalizedValue() {
    normalizedValue = Math.min(1.0, getValue() / maxDist);
    return normalizedValue;
  }

  //no reset operation needed, the default one will be used (do nothing)
}
```

A.5.3 Agent actuators

Actuators must extend the *Actuator* class and must define their *step* and *reset* behaviour. The class name should start with *A_*.

Before the *step* method is called, the actuator will receive its command and store it in the *normalizedValue* member. The *normalizedValue* will always be a real number in the [0.0 , 1.0] range. The *step* method will convert the *normalizedValue* command into a concrete action of the actuator.

computeWorldPosAndAngle() must be called before using the actuator's position in calculations (this will update the position with the translations and rotations of the bot's body).

```
\\converts a [0 , 1] commands into a [-maxForce , +maxForce] impulse.
\\a 0.5 command results in a null force.
public class A_Wheel extends Actuator {
  public float maxForce = 30.0f;
  public float actuatorValue = 0.0f;

  public A_Wheel(Vec2 lp , float la, BotBody b,float mf){
    super(lp,la,b);
    maxForce = mf;
  }

  @Override
  public void step() {
    computeWorldPosAndAngle();
    // conversion from command to force value.
    actuatorValue = (((float)normalizedValue*2.0f)-1.0f)*maxForce;
    Vec2 f =
      new Vec2((float)Math.cos(worldAngle)*actuatorValue,
            (float)Math.sin(worldAngle)*actuatorValue);
    Vec2 p = new Vec2(worldPosition);
    bot.body.applyForce(f, p);
  }
}
```

A.5.4 Drive module

Actuators must extend the *DriveModule* class and must define their *doStep* and *checkInputOutputUse* behaviour. The class name should start with *DM_*.

The *checkInputOutputUse* method is called when loading the MIND hierarchy, and is used to request modules, and activate them if they are not already used by the master skill.

The *doStep* method is called before the skill hierarchy and can be used to set de values of variable, add a concurrent command to an actuator or send influence to skills (including skills not present in the hierarchy).

The following example is the GoToTarget drive module. The GoToTarget skill uses a variable to represent its target, this variable must be set by another skill. When GoToTarget is set as a master skill (when learning the skill for the first time), no other skill can set the target variable, this is done by the drive module instead.

```java
public class DM_GTT extends DriveModule{
  SensorModule targetSensor;
  VariableModule targetVariable;
  SensorModule targetDistSensor;
  VariableModule targetDistVariable;

  public DM_GTT(EvoAgentMind mind)
  {
    super(mind);
  }

  public void doStep() {
    targetVariable.overrideValue(targetSensor.getValue());
    targetDistVariable.overrideValue(targetDistSensor.getValue());
  }

  public void checkInputOutputUse() {
    targetSensor = mind.getSensor("RADOBJA");
    targetSensor.setInUse(true);
    targetVariable = mind.getVariable("VAR_TARGET");
    targetVariable.setInUse(true);
    targetDistSensor = mind.getSensor("DISTOBJA");
    targetDistSensor.setInUse(true);
    targetDistVariable = mind.getVariable("VAR_TARGETDIST");
    targetDistVariable.setInUse(true);
  }
}
```

A.5.5 Simulation environment

Create a new class in the *evoagent2dsimulator.experiments* package. The name of the class will be the name of environment (the ENV parameter of a task). Your environment class must extend the *SimulationEnvironment* class. You're free to override any function you wish (at your own perils).

In the class constructor you set the name and enable the obstacle generation by setting the *hasObstacles* field to true.

```java
  public EXP_GTDZA(String botMod)
  {
    super(botMod);
    this.name = "GTDZ+Avoid";
    hasObstacles = true;
  }
```

The obstacle generation method generates a grid of static obstacle with some degree of randomness. In the case of learning methods, it will generate a set of obstacle grids, different for each repetition (successive evaluation of the same agent). All the agents will be evaluated on the same set of obstacle grids (minimizing the bias of random generation).

The parameters of this grid generation can be tweaked by setting the following fields:

```java
    protected double minObstacleSize = 1.0;
    protected double maxObstacleVariability = 2.0;
    protected double obstacleSpacing = 18.0;
```

or the method generating an obstacle grid can be overridden:

```java
public ArrayList<ObstaclePos> generateObstaclesParameters();

public class ObstaclePos
  public ObstaclePos(Vec2 position,float orientation, float size)
```

The definition of your environment is done by overloading the *init* method. Here follows a commented example that covers most of what you'll have to do.

```java
@Override
public void init()
{
  super.init();
  // set the corrdinates of the starting position of the bot, then run the
      creation method.
  botStartPos = new Vec2(-00.5f,-0.0f);
  makeBot();

  // create wold elements and add them to the worldElements list.
  targetZone = new TriggerZone(new Vec2(-20,-20), (float)(Math.PI/4), 5);
  getWorldElements().add(targetZone);

  // create control function and reward functions and add them to their
      respective lists.
  controlFunctions.add(
    new CF_NextOnCollisionAndTimeout(bot,this, 20000));
  rewardFunctions.add(
    new RW_SensorOverThreshold(
                bot,
                100,
                bot.sensors.get("SENSDZ"),
                0.5));
  rewardFunctions.add(
    new RW_ClosingOnTarget(bot, 0.001, targetZone));

  // some sensors need to be linked to a world element
  ((S_ZonePresence)bot.sensors.get("SENSDZ"))
                .setTarget(targetZone);
  ((S_Radar)bot.sensors.get("RADDZ"))
                .setTarget(targetZone);

  // 2d physics world initialisation
  makeWorld();

  // 2d physics bot initialisation
  bot.registerBotToWorld(getWorld());

  // post initialisation custom functions (here randomly placing a world element)
  randomlyPlaceTargetZone();
}
```

If you have custom world elements that needs operations when the simulation resets, overload the *reset* method

```java
@Override
public void reset()
{
  // this resets the reward and control functions, also place the bot at the
      starting position
  super.reset();
  // when the simulation resets, generate a new random position for the target
  randomlyPlaceTargetZone();
}
```

Finally, you can overload *postStepOps* which is a method that is called after each simulation tick.

```java
@Override
protected void postStepOps() {
```

```
  super.postStepOps();
  // when the bot has reached the target
      if(((S_ZonePresence)bot.sensors.get("SENSDZ"))
          .getNormalizedValue() > 0.5)
  {
    // generate a new random position for the target
    randomlyPlaceTargetZone();
    // and reset the reward functions (since this is called after the simulation
        tick, the bot was already rewarded for reaching the target)
    for(RewardFunction r: rewardFunctions)
      r.reset();
  }
}
```

A.5.6 World elements

At this point you must know the JBox2d library. Defining shapes, dynamic properties, and the collision mask system.

VirtualWorldElement World elements that does not have physical presence should extend the *VirtualWorldElement* class. (ex: waypoints, markers, trigger zones...)

```
public class TriggerZone extends VirtualWorldElement {
  public String name = "dropZone";

  public TriggerZone(Vec2 worldPos, float worldAng,float s, String Label) {
    super(worldPos, worldAng,s);
    name = Label;
  }

  public boolean isPointInDZ(Vec2 point) {
    //find the world coordinates of the zone
    Vec2 points[] = new Vec2[4];
    points[0]= new Vec2(-size,size);
    points[1]= new Vec2(size,size);
    points[2]= new Vec2(size,-size);
    points[3]= new Vec2(-size,-size);
    for(int i = 0 ; i < 4 ; i++)
      points[i].set(getWorldPoint(points[i]));
    //clockwise check of all edges of a convex polygon.
    //if the point is on the left of any edge,
    //it is outside of the polygon
    for(int i = 0 ; i < 4 ;i++)
      if(!isVectorRight(
        new Vec2(points[(i+1)%4].x-points[i].x,
              points[(i+1)%4].y-points[i].y),
        new Vec2(point.x-points[i].x,
              point.y-points[i].y)))
        return false;
    return true;
  }

  private boolean isVectorRight(Vec2 v1, Vec2 v2) {
    if(((v1.x) * (v2.y)) - ((v2.x) * (v1.y))>0.0)
      return false;
    else
      return true;
  }
}
```

StaticWorldElement World elements that have physical presence but will NEVER move of be affected by forces should extend the *StaticWorldElement* class. (ex: static obstacles of any kind)

```java
public class ObstacleStaticBox extends StaticWorldElement {

  public ObstacleStaticBox(Vec2 worldPos, float worldAng,float size, World w){
    super(worldPos, worldAng,size);
    sd.shape = new PolygonShape();
    ((PolygonShape)sd.shape).setAsBox(this.size, this.size);
    sd.friction = 0.0f;
    sd.density = 2.0f;
    registerToWorld(w);
  }
}
```

DynamicWorldElement World elements that have physical presence and can be subjected to forces should extend the *DynamicWorldElement* class. (ex: projectiles, a rolling ball, boxes to stack...)

```java
// the target object is a ball to catch
public class TargetObject extends DynamicWorldElement {
  public TargetObject(Vec2 worldPos, float worldAng, float s) {
    super(worldPos, worldAng, s);
    sd.shape = new CircleShape();
    sd.friction = 0.0f;
    sd.restitution = 1.8f;
    sd.density = 2.0f;
    sd.filter.categoryBits = CollisionDefines.CDTargetObj;
    sd.shape.m_radius = size;
    bd.angularDamping = 1.5f;
    bd.linearDamping = 0.15f;
  }
}
```

A.5.7 Reward functions

Reward function must extend the *RewardFunction* class and must define their *computeRewardValue* and *reset* behaviour. The class name should start with *RW_* .

```java
//reward the bot for getting closer to the target, punish him for getting away.

public class RW_ClosingOnTarget extends RewardFunction{
  public VirtualWorldElement target =null;
  double dist = -1;

  public RW_ClosingOnTarget(BotBody b, double rewardSt, VirtualWorldElement
     targetin) {
    super(b, rewardSt);
    target = targetin;
  }

  @Override
  public double computeRewardValue() {
    double ret = 0.0;
    double curDist =  MathUtils.distance(
             bot.body.getPosition(),
             target.getWorldPosition());
    //only reward if there is a previous distance to compare
    if(target != null && dist != -1)
      ret = rewardStep*(dist-curDist);
    dist = curDist;
    return ret;
  }

  @Override
  public void reset()
  { //comparing with the state of a previous simulation wouldn't make sense
    dist = -1;
```

```
  }
}
```

A.5.8 Control functions

Control function must extend the *RewardFunction* class and must define their *performCheck* and *reset* behavior. The class name should start with *CF_*

```
//stop the simulation after a set number of ticks
public class CF_NextOnTimeout extends ControlFunction {
  int tickLimit;
  int tickCounter = 0;

  public CF_NextOnTimeout(BotBody b,SimulationEnvironment2DSingleBot w, int
      ticklim){
    super(b,w);
    tickLimit = ticklim;
  }

  @Override
  public boolean performCheck(){
    tickCounter++;
    if(tickCounter > tickLimit)
      return true;
    return false;
  }

  public void reset(){
    super.reset();
    tickCounter = 0;
  }
}
```

A.6 CogLogo manual

CogLogo Suro (2017) is a NetLogo Wilensky (1999) extension written in Java which provides an implementation of the cogniton architecture that integrates smoothly with NetLogo ability to describe an environment, physical agents, objects and interactions.

CogLogo integrates with NetLogo by providing procedures to modify the cognitons weights, organize groups of agents and modify culturons weights. The procedure *cogLogo:choose-next-plan* is called at the modeller's discretion, and returns a string which can be used to call any corresponding NetLogo or user defined procedure.

CogLogo has a graphical editor to design the internal cognitive architecture of the agents: the *Cognitive Scheme*. The *Cognitive Scheme* describes the individual thought process with cognitons and collective elements with culturons.

Multiple Cognitive Schemes can be defined and used in the same model, and can be assigned to different agent kinds (NetLogo's *breeds*). For each Cognitive Scheme we can choose among several decision makers.

CogLogo also provides tools to observe its elements during the simulation. The agent watcher window will show the weight of each cogniton and the calculated weights of the plans for a selected agent in real time.

A.6.1 Execution cycle

- 1 - The agent calls *cogLogo:choose-next-plan*.
- 2 - Active plans are evaluated: plans that do not have at least one conditional link to an active cogniton are deactivated.
- 3 - Plan weight is calculated: the value of each cogniton is multiplied by the influence link value and summed in the plan.
- 4 - The decision maker is called and the plan is selected according to its mechanism.
- 5 - *cogLogo:choose-next-plan* returns the name of the plan, which can be executed with the *run* command.
- 6 - The plan acts on the environment and on the agent's mind: cogniton values are altered, cognitons can be activated or deactivated, the agent can join or leave groups and change their degree of participation, feedback can be sent through reinforcement links.
- 7 - next simulation step repeats the process.

A.6.2 Links

The Cognitive Scheme editor is used to create the cognitons and culturons, and the influence links to the plans. The value of each influence link can be defined as a positive or negative real number.

CogLogo also offers two kinds of links not discussed before, the conditional links and the reinforcement links, which we will briefly explain.

The *conditional links* play a role in the simulation by telling the decision maker which plans are available. Cognitons can be activated and deactivated, and a plan can take part in the selection process only if it has at least one conditional link to an active cogniton (even if the cogniton weight is 0). This means that even if, through other cognitons, a plan obtains the highest calculated weight, it will never be selected without a conditional link to an active cogniton.

The *reinforcement links* provide a simple and more readable way to implement the reinforcement mechanism. While the evolution of certain cognitons are more a matter of perception and regulation (such as hunger, climate influence or age), others are involved in the long term behaviour of the agent and reflect a learning process (such as a social specialization or an attitude towards external factors). When an agent runs a plan, a feedback operation (*CogLogo:feed-back-from-plan*) can be called with a value parameter, this value is multiplied by the weight of the reinforcement link defined in the Cognitive Scheme and added to the corresponding cogniton.

The use of reinforcement links is entirely optional and the same effect can be accomplished by using the regular procedures to set the values of the cognitons. They are simply meant to simplify reinforcement mechanisms and make them visible in the CogLogo interface.

A.6.3 Decision Makers

MaximumWeight : The plan with the maximum weight is selected.

WeightedStochastic : The weight of each plan represents the probability for the plan to be selected. If PlanA = 5 and PlanB = 3, the probability of being selected is: PlanA = 5/8 = 0.625 and PlanB = 3/8 = 0.375. A random function (0,1) is then called to select the plan.

BiasedWeightedStochastic : Works the same way as the WeightedStochastic, but increase the probability of selecting the better plans accordingly to the bias factor. The plans are sorted from higher probability to lower, covering the range from 0 to 1 (the highest probability covers the range from 0 to p, then the next plan from p to p+1 ...). The random function result is then elevated to the degree of the bias specified, which will bias the random function towards giving values closer to 0 (a bias of 1 will give the same results as the regular WeightedStochastic, but with a slightly higher computing cost).

A.6.4 Interface

General

Save model: saves all cognitive schemes

View

Cognitive Scheme editor: shows the Cognitive scheme editor view of your current cognitive scheme. You can edit cognitons in this view.

Groups and roles editor: shows the Groups and roles editor view of your current cognitive scheme. You can edit groups, roles and culturons in this view.

Select

Create new cognitive scheme: Create a new cognitive scheme.

Cognitive scheme list: lets you select the current cognitive scheme.

Options

Setting of the current cognitive scheme: the setting window for the current cognitive scheme. You can set the decision maker in this window.

Save current cognitive scheme: Save changes to the current cognitive scheme only.

Delete current cognitive scheme: Deletes the current cognitive scheme.

Observe

Observe a single agent: Opens the agent watcher window.

Help

Help: shows the readme.txt.

Console: shows the CogLogo debugging console. May help you understand what went wrong with your model.

About: Licence and such.

A.6.5 NetLogo Commands

General

OpenEditor: Opens/close the editor panel.

choose-next-plan: Returns a string, the name of the next plan to run (ex: run cognitonsfornetlogo:choose-next-plan).

feed-back-from-plan <string planName> <double val>: Propagates val to all the cognitons (or culturons) linked to the plan by a reinforcement link. The value parameter (val) is multiplied by the value of the reinforcement link, the result is added to the value of the cogniton (or culturon).

report-agent-data: reports the value of all cognitons and the resulting weights of the plans to our observation panel.

reset-simulation: to be called in the setup, resets group count and agent data tracking.

Cognitons

init-cognitons: Must be called for each agent using a cognitive scheme, only during the setup.

add-to-cogniton-value <string cognitonName> <double val>: Adds val to the corresponding cogniton.

set-cogniton-value <string cognitonName> <double val>: sets the value to val of the corresponding cogniton.

get-cogniton-value <string cognitonName>: Returns the value of the corresponding cogniton. If the cogniton is not active the function returns 0.

activate-cogniton <string cognitonName>: Activates a cogniton and sets its value to 0. All plans connected to this cogniton via dependency links will be available for the next call of choose-next-plan (if they were not already).

deactivate-cogniton <string cognitonName>: Deactivates a cogniton. All plans connected to this cogniton via dependency links will be unavailable for the next call of choose-next-plan if they are not connected to another active cogniton via dependency links.

Culturons

create-and-join-group <string groupName> <string roleName>: Create an instance of <groupName> type and join it in the role <roleName>. When joining, the participation (involvement) value is set to 1.0.

join-group <string groupName> <string roleName> <double groupId>: Join the instance <groupId> of type <groupName> in the role <roleName>. When joining, the participation (involvement) value is set to 1.0.

add-to-participation <string groupName> <double val>: Adds val to the corresponding group involvement.

set-participation <string groupName> <double val>: Sets the value to val of the corresponding group involvement.

get-group-id <string groupName>: Returns the group identifier of groupName (a number >= 0) of the agent. If the agent is not in any role of this group type the function returns -1.

get-group-role-id <string groupName> <string roleName>: Returns the group identifier of groupName (a number >= 0) of the agent if the agent has the role of roleName. If the agent is not in this role of this group type the function returns -1.

leave-group <string groupName>: Leave the group of groupName (if the agent is not in this type of group, does nothing).

leave-all-groups: The agent leaves all his groups.

add-to-culturon-value <String groupName> <String culturonName> <Double value>: Add <value> to the culturon <culturonName> of the group type <groupName>.

set-culturon-value <String groupName> <String culturonName> <Double value>: Set <value> as the value of the culturon <culturonName> of the group type <groupName>.

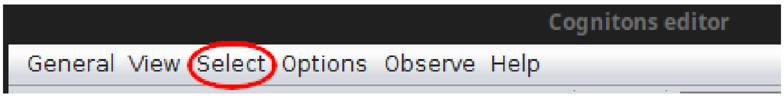
get-culturon-value <String groupName> <String culturonName>: Return the value of the culturon <culturonName> of the group type <groupName>. If the agent is not in any role of this group type the function returns 0.

A.7 CogLogo: A short tutorial

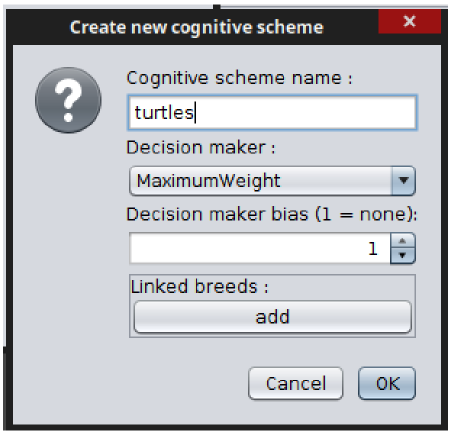
A.7.1 Creating the cognitive scheme

Before you can open the editor, import the coglogo extension (*extensions [CogLogo]* in the code window) and save your model. Open the editor (by command or button).

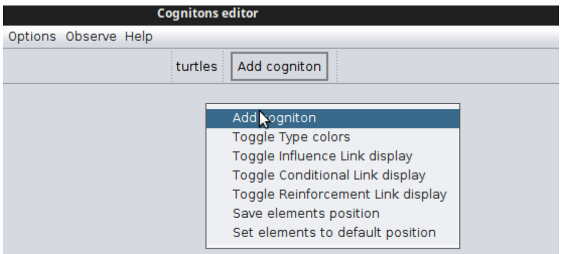
In the menu bar: Select>create new cognitive scheme.



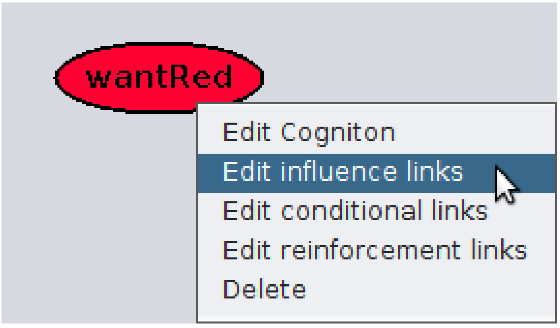
Name the cognitive scheme “turtles” (the breed we use in this model), leave default for other settings.

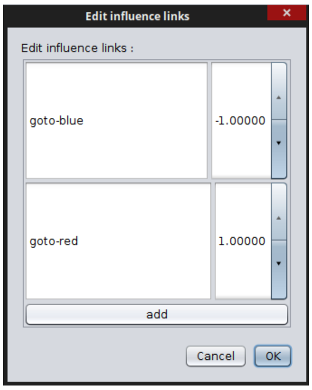


Right click on the background >add cogniton (or use the button). Name it “wantRed” and tick the “is added at birth?” box.

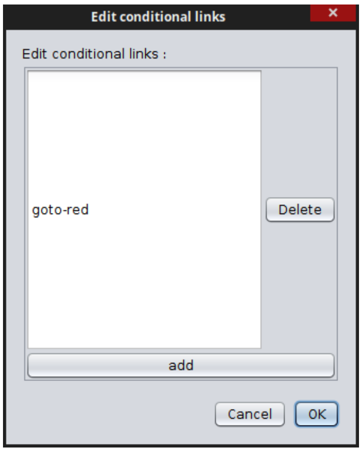


Right click on the “wantRed” cogniton >edit influence link. Add a link to the plan "goto-red" (type in the text box) and set its value to 1. Add a link to the plan "goto-blue" and set its value to -1 and close the window (ok).





Right click on the "wantRed" cogniton >edit conditional link. Add a link to the plan "goto-red" and close the window (ok).

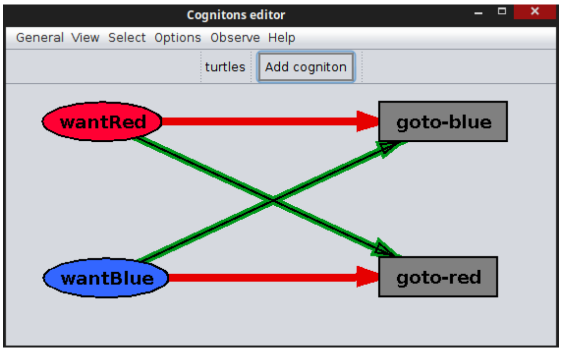


Now we repeat the process for the "wantBlue" cogniton: Right click on the background >add cogniton (or use the button). Name it "wantBlue" and tick the "is added at birth?" box set the default value to 1. Right click on the "wantBlue" cogniton >edit influence link. Add a link to the plan "goto-blue" (type in the text box) and set its value to 1. Add a link to the plan "goto-red" and set its value to -1 and close the window (ok). Right click on the "wantBlue" cogniton >edit conditional link. Add a link to the plan "goto-blue" and close the window (ok). The *turtles* cognitive scheme should look like this:

A.7.2 Using the cognitive scheme in NetLogo

The following shows the complete **setup** procedure. Each agent using a cognitive scheme must call the *coglogo:init-cognitons* procedure (line 8).

```
1 to setup
```




```
  clear-all
  reset-ticks
  set redpatch patch 9 9
  set bluepatch patch -9 9
  set greenpatch patch 0 -9
  create-turtles 1 [
    coglogo:init-cognitons
    set color white
    set size 3 ; easier to see
    setxy random-xcor random-ycor
  ]
  ask redpatch [
    set pcolor red
    ask neighbors [set pcolor red]
  ]
  ask bluepatch [
    set pcolor blue
    ask neighbors [set pcolor blue]
  ]
  ask greenpatch [
    set pcolor green
    ask neighbors [set pcolor green]
  ]
end
```

In the **go** procedure, each agent calls *coglogo:choose-next-plan* which returns the name of the selected plan as a character string. The run primitive is able to run a NetLogo procedure from a character string corresponding to its name.

coglogo:report-agent-data makes the data available to the CogLogo Agent Watcher. Here reporting is done each tick, you can choose to report every 100 ticks or only report for a specific agent.

```
to go
  ask turtles [
    act-on-cognitons                ; cognitons evolve
    run coglogo:choose-next-plan    ; plan chosen by cognitons
```

```
    coglogo:report-agent-data      ; send data to agent watcher interface
  ]
  tick
end
```

In **act-on-cognitons** we set the values of the cognitons.

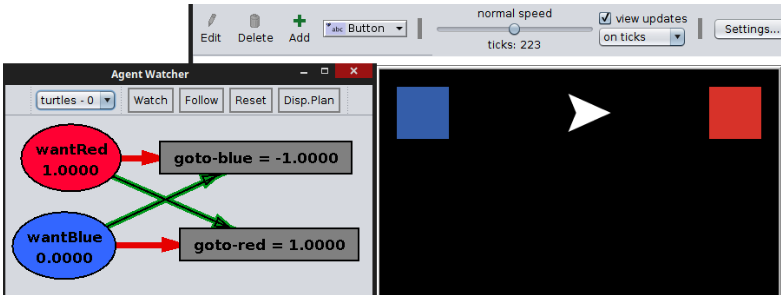
```
to act-on-cognitons
  if pcolor = red[
    coglogo:set-cogniton-value "wantRed" 0
    coglogo:set-cogniton-value "wantBlue" 1
  ]
  if pcolor = blue[
    coglogo:set-cogniton-value "wantRed" 1
    coglogo:set-cogniton-value "wantBlue" 0
  ]
end
```

Each plan in the cognitive scheme MUST have a corresponding NetLogo procedure. The name of the procedure must correspond exactly.

```
to goto-blue
  face bluepatch
  fd 1
end
```

Don't forget to define goto-red

Try your model, the turtle should go back and forth between the blue and red patch.



A.7.3 Adding reinforcement links to the cognitive scheme

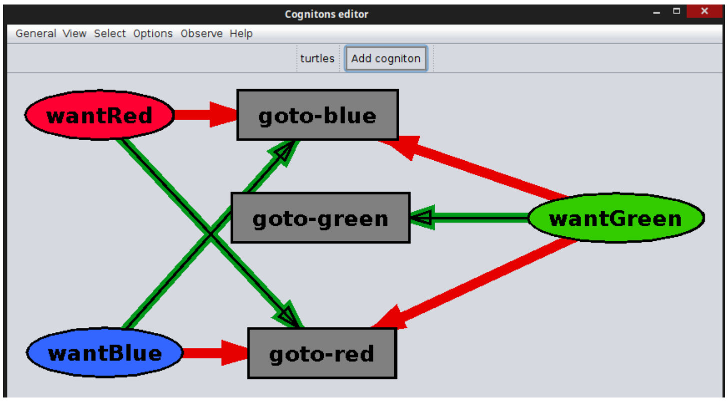
We will repeat the previous process for a "wantGreen" cogniton.

Right click on the background >add cogniton (or use the button). Name it "wantGreen" and tick the "is added at birth?" box.

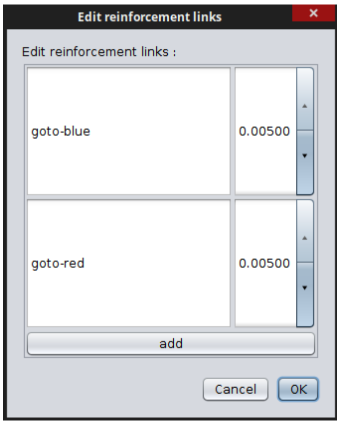
Right click on the "wantGreen" cogniton >edit influence link. Add a link to the plan "goto-green" (type in the text box) and set its value to 1. Add a link to the plan "goto-red" and set its value to -1. Add a link to the plan "goto-blue" and set its value to -1 and close the window (ok).

Right click on the “wantGreen” cogniton >edit conditional link. Add a link to the plan “goto-green” and close the window (ok).

At this point your model should look like this:



Right click on the “wantGreen” cogniton >edit reinforcement link. Add a link to the plan “goto-red” and set its value to 0.005. Add a link to the plan “goto-blue” and set its value to 0.005 and close the window (ok).

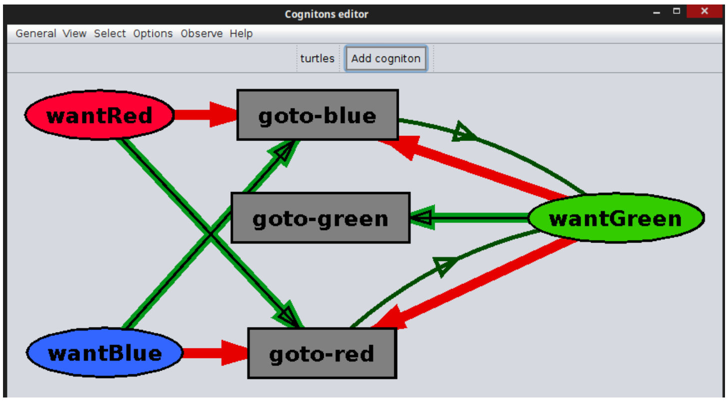


The complete model should look like this:

A.7.4 Using reinforcement links in NetLogo

Every use of the goto-blue and goto-red plans will reinforce the “wantGreen” cogniton. Simply add the *coglogo:feed-back-from-plan “goto-*” *value** to the corresponding plan.

```
to goto-blue
  face bluepatch
  fd 1
  coglogo:feed-back-from-plan "goto-blue" 1.1
end
```



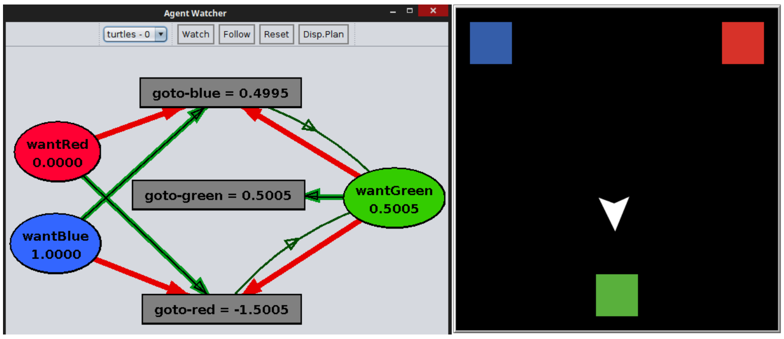

```
to goto-red
  face redpatch
  fd 1
  coglogo:feed-back-from-plan "goto-red" 1.1
end

to goto-green
  face greenpatch
  fd 1
end
```

To complete the model, we will reset the "wantGreen" cogniton to 0 when the agent reaches the green patch.

```
to act-on-cognitons
  if pcolor = red[
    coglogo:set-cogniton-value "wantRed" 0
    coglogo:set-cogniton-value "wantBlue" 1
  ]
  if pcolor = blue[
    coglogo:set-cogniton-value "wantRed" 1
    coglogo:set-cogniton-value "wantBlue" 0
  ]
  if pcolor = green[
    coglogo:set-cogniton-value "wantGreen" 0
  ]
end
```

Try you model, the turtle should go back and forth between the blue and red patch. After enough reinforcement is accumulated, the turtle should go to the green patch. Reaching the green patch will reset the value of wantGreen cogniton, the turtle will resume the red-blue cycle.



Agent Watcher
turles - 0
Watch
Follow
Reset
Disp.Plan
goto-blue = 0.4995
wantRed
0.0000
goto-green = 0.5005
wantGreen
0.5005
wantBlue
1.0000
goto-red = -1.5005

List of Figures

List of Tables