Acta Polytechnica **54**(5):367–377, 2014
doi:10.14311/AP.2014.54.0367

available online at http://ojs.cvut.cz/ojs/index.php/ap

TOWARDS EVOLUTIONARY DESIGN OF COMPLEX SYSTEMS INSPIRED BY NATURE

Jaroslav Vítků*, Pavel Nahodil

Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Cybernetics, Technická 2, 16627, Prague 6, Czech Rep.

* corresponding author: vitkujar@fel.cvut.cz

Abstract. This paper presents the first steps towards the evolutionary design of complex autonomous systems. The approach is inspired by modularity of the human brain and the principles of evolution. Rather than evolving neural networks or neural-based systems, the approach focuses on evolving hybrid networks composed of heterogeneous sub-systems implementing various algorithms/behaviors. Currently, evolutionary techniques are used to optimize the weights between predefined blocks (so-called Neural Modules) in order to find an agent architecture appropriate for a given task. The framework, together with the simulator of such systems is presented here. Then, examples of agent architectures represented as hybrid networks are presented. One architecture is hand-designed and one is automatically optimized by means of an Evolutionary Algorithm. Even such a simple experiment shows how evolution is able to pick-up unexpected attributes of the task and exploit them when designing a new architecture.

Keywords: agent, architecture, artificial life, creature, behavior, hybrid, neural network, evolution.

1. Introduction — problem decomposition

Many researchers have had a long-term goal: to build an autonomous robotic system that is able to operate in real-world conditions in a robust way. However, after many years of research and development, there are still no satisfactory results. The uneasy task of building reliable robots is composed of too many smaller sub-problems, such as vision, reasoning in unknown domains, etc. On this long way, there are simply too many problems. Moreover, most of these smaller problems do not yet have a feasible solution.

1.1. Modular design in practice

Nevertheless, many of these small problems have already been solved relatively well, such as for example pattern recognition, planning, reinforcement learning, etc. In order to build a well-performing system it is often possible just to use these known systems. Or even better, it is suitable to *use implementation of known algorithms*, which are tested and have performed on a given task. Building a more complex system then becomes faster and simpler. This approach of re-using modules was started by the Robotic Operating System (ROS). The main goal of ROS is to provide nodes — implementations of algorithms — that can be re-used in robotic applications (such as path-planning, vision etc.) [1]. The user picks selected nodes and composes the resulting system from them. Generally, such an approach is called *Top-Down*.

The opposite direction is called a *Bottom-up* design approach. Connectionist Artificial Neural Networks (ANNs) provide an example. In ANNs, no piece of the system (neuron) implements useful processing alone. However when composed together, complex behavior can be obtained often with the use of emergence.

The most traditional design of ANNs is as follows: (1) predefine some structure of ANN; (2) optimize the weights between neurons, in order to obtain the desired behavior. Optimization can be done by some local algorithm [2], or by global approaches, such as neuro-evolution [3]. More recently, ANNs have often been designed by a principle called "Neural Engineering". This is a Top-down approach, which works with Modular Neural Networks (MNNs) [4]. The purpose of each module (sub-network) is defined. The required behavior of the resulting system is then obtained by composing multiple sub-networks together [5]. In this way, complex systems can be composed [6].

1.2. How can systems be designed in a general way?

Both of the approaches mentioned here have their own benefits and their own weaknesses. The use of already-implemented pieces of robotic software provides high reliability of the system, and the user has big insight into the inner functionality. However, the resulting design can be very constrained, and the design possibilities are limited. ANNs provide unconstrained design options, and are very good at dealing with uncertainty in data, but their structure is often too complex to be understood directly and altered well.

Here, authors focus on designing agent architectures (learning and decision-making systems for (virtual) robots) in a hybrid way. The process of designing agents is not be fixed to any of the design approaches mentioned above, but takes advantages of both. The novel presented framework is called Hybrid Artificial Neural Network Systems (HANNS). It is an attempt to

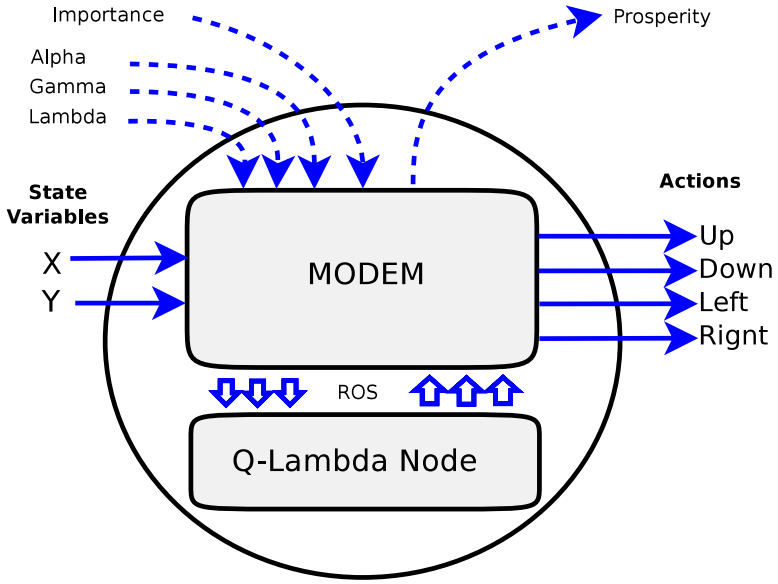


Figure 1. A Neural Module with two data inputs, four data outputs and four configuration inputs. The module also contains "Prosperity" output, which provides subjective heuristics defining how well the module performs in the current architecture during the current simulation. Here, a modem serves as a communication interface between the NengoROS simulator and an external ROS node, which implements a given algorithm/functionality.

unify Modular Neural Networks with purely top-down and practical sub-systems, such as those implemented in ROS. One of main goals of this framework is to be able to automate the design of new architectures for a given task. This approach will be shown on a simple example here.

1.3. Structure of the text

The following chapter will describe the main concepts of our HANNS framework, together with the NengoROS simulator. The third chapter will show these principles applied to a simple agent architecture implementing Motivation-Driven Reinforcement Learning (RL). Then the evolutionary approach is used to design a similar architecture automatically. Finally, the two results are compared and discussed.

2. Hybrid Artificial Neural Network Systems

Hybrid Artificial Neural Networks can be described as Modular ANNs [4] composed of heterogeneous subsystems. Each subsystem can implement various methods of decision-making and employ various representations of information processing, from neural-like to symbolic representations [7]. This means that (for a connecting symbolic-based module to a neural network) the symbolic representation has to be derived from the activity of particular neurons. This has to be done for example by means of a predefined lexicon, rule extraction or similar methods.

This chapter describes the main principles employed by the presented Hybrid Artificial Neural Network Systems (HANNS) framework. The framework uses *seamless communication* between all modules in the network and *encapsulated information transformation* where needed. The particular strategy of information transformation belongs to own module and is hidden from the rest of the hybrid network.

2.1. Communication and sub-system Representation

In order to deal with connecting different modules which use different types of communication, a common type of communication had to be defined. Since there is a possibility that every system could be implemented by neural computation one day, the aim is to *add higher-level subsystems into ANNs*. Therefore the entire network uses neural-like communication in this framework. Each *"Neural Module"* subsystem can implement arbitrary behavior and has a defined number of input and output connections. Each connection can represent a real-valued number, typically from the interval $\langle 0, 1 \rangle$. The scheme of an example of a Neural Module is shown in Fig. 1.

The figure shows that the framework uses simple communication in the network, while a transformation from/to a symbolic (or other) domain is implemented inside a particular Neural Module (depicted as *Modem* in the schematics), if needed. According to [7], this framework uses "hybrid system coupling interleaved by function calls", where all activity on inputs of Neural Module is translated into the inner representation of Modules and is processed accordingly. After processing the data, the Module encodes the result back into the "neural communication" and sets data on its outputs.

2.2. Configuration of a Neural Module

This type of Neural Module can implement various types of information processing. However, even do-

main independent algorithms often need fine-tuning of their parameters in order to work efficiently. Therefore the Neural Module has configuration inputs in addition to data inputs and outputs. These inputs are represented in the same way as data inputs, but their purpose is to define the parameters of the algorithm(s) encapsulated in the Neural Module. These parameters can be used as normal data inputs, which means that their value can be changed during the simulation. The only difference is that these inputs *can be ignored*. If the configuration inputs are left unconnected, these hold a predefined (default) value of the algorithm parameters. This can be used for simplifying the overall complexity of the network topology.

2.3. Prosperity of the Neural Module

One of the main aims of this framework is to study new, alternative use-cases of known algorithms/subsystems. During the automatic design of these Hybrid Artificial Neural Network Systems, the following use cases of the Neural Module can occur:

- The Neural Module is connected in a *completely wrong way*: the corresponding algorithm is used inefficiently or it does not work at all.
- The Neural Module is connected in an *unexpected, new way*: the corresponding algorithm is employed, but in a way not anticipated by the designer, a potentially new use of the algorithm.
- The Neural Module is connected in an *expected way*: the algorithm is employed as expected during the Neural Module design.

The second of these three cases does not necessarily mean that the algorithm is not advantageous in the architecture. However, the behavior of an algorithm (and potentially its purpose in the architecture) can be often hard to analyze by hand.

In order to identify incorrectly used parts of the resulting architecture, it would be convenient to distinguish between these three use-cases automatically. Furthermore it would be useful to be able to evaluate the performance of the algorithms in a given situation. However, this is possible only by means of heuristics. The value of *Prosperity* output defines *subjective heuristics* defining "how well the algorithm performs" in a given architecture during the simulation. This enables the user (and potentially EA) to distinguish between good and bad parts of a particular architecture. It is up to the designer of a particular Neural Module how to define its Prosperity function. The function should produce values in the interval $\langle 0, 1 \rangle$.

2.4. The NengoROS simulator

The next goal of our HANNS framework is to provide a platform for simulating agent architectures. In order to rapid prototype and simulate these architectures, the simulator of Hybrid Artificial Neural Network Systems was created. The main objectives were the following: maximum reuse of current implementations of algorithms, decentralized and event-driven simulation and integration with an advanced simulator of large-scale ANNs.

The resulting open-source system is called NengoROS [8]. It connects a simulator of large-scale ANNs of the 3rd generation called Nengo [9] with the Robotic Operating System (ROS) [1]. The Nengo simulator was created with the goal of implementing the Neural Engineering Framework (NEF) and it therefore supports modular ANNs. The addition of ROS enables the simulator to use any ROS-enabled subsystem in the simulation. The integration of ROS into the simulator is depicted in the Fig. 1, where the component called "Q-Lambda Module" is an external ROS node and the "Modem" serves as the interface between Nengo and ROS. The ROS-based components can be run remotely and can represent a particular piece of SW, a simulated world or even robotic HW.

3. Description of selected Neural Modules

An example of the use of the presented framework will be shown on two selected Neural Modules.The first module implements the Reinforcement Learning algorithm, and the other module presents the physiological state of the agent. Together, these modules implement a principle called *Motivation-driven Reinforcement Learning*. For each of these algorithms the theory will be briefly described. Then, integration into the Neural Module will be introduced.

3.1. Reinforcement Learning Module

Agent architectures from the domain of Artificial Life (ALife) often require online and model-free Reinforcement Learning (RL). An algorithm called Q-Learning meets these requirements. This discrete algorithm learns a desired strategy only by means of interaction with the environment based on actions produced rewards/punishments received. The algorithm learns behavior which leads towards the nearest reward while avoiding punishments. The algorithm is encapsulated as a standalone sub-system — Neural Module here. This particular example (the use of RL in HANNS) can be likened to Ensemble Algorithms in Reinforcement Learning [10], or to the Aggregated Multiple Reinforcement Learning System (AMRLS) [11]. Compared to these, a single ensemble is represented as a Multiple-Input Multiple-Output sub-system, which communicates compatibly with 2^{nd} generation of artificial neurons. The Action Selection Mechanism (ASM) is also currently integrated in the Neural Module currently.

Figure 2 presents a graphical representation of our modification of the Q-Learning algorithm. This algorithm runs inside a standalone ROS node and communicates externally only by means of ROS messages. Figure 1 shows the integration of this ROS node into a Neural Module. The Module is compatible with the

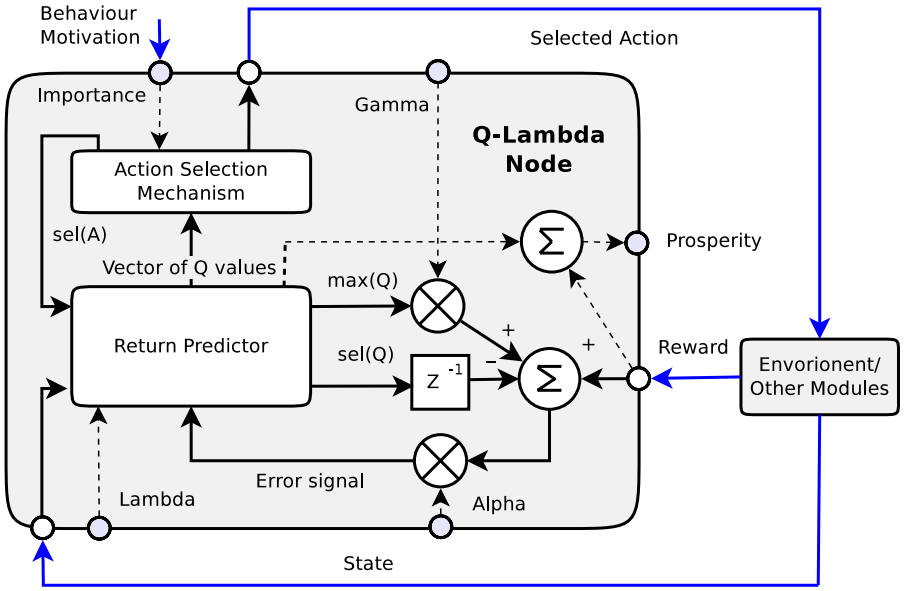


Figure 2. Scheme of the Stochastic Return Predictor (SRP) implementation. It is composed of the Q-Lambda algorithm and ASM. The (sub-)system is implemented as a stand-alone ROS node, which can be used as a Neural Module. Outputs encode a selected action by means of the $1ofN$ code (where N is number of available actions). M state variables are encoded by M data inputs, and each data input is sampled in a predefined number of discrete values. A node selects one action at each time-step and expects information about the new state and the reward. The node has the following configuration inputs: α, γ, λ, *Importance*, which affect learning, and Action Selection Methods (in the case that these inputs are not connected, default values are used). The *Prosperity* heuristics represents the average between *MCR* and overall coverage of the state space (number of visited states).

HANNS framework, and can be seamlessly connected into a network of heterogeneous nodes.

3.1.1. Learning

For learning, the Neural Module uses standard algorithm called Q-Learning (more exactly: Q-Lambda, see below). It is named according to its Q matrix, which maps state-action pairs to utility value. At each environment state, the $Q(s,a)$ matrix stores utility values for all possible actions:

$$Q : A \times S \rightarrow \mathbb{R}, \tag{1}$$

where A is a set of all available actions and S is the set of all possible states of the environment. The utility value represents the discounted future reinforcement that will be received by the agent if it will follow a given action a in a given state s. Online learning is governed by obtaining new $Q(s,a)$ values into the matrix. A change of the value in the matrix is represented by the following equation:

$$\delta = r_{t+1} + \gamma max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t). \tag{2}$$

The algorithm stores the current state and the action that was just executed (s_t, a_t). Action a_t may cause receiving the reward r_{t+1} and a transition[1] into the new state s_{t+1}. Based on this information and the optimal action in the new state: $a^*_{t+1} = max_a Q(s_{t+1}, a_{t+1})$, the value $Q(s_t, a_t)$ is updated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta, \tag{3}$$

the following parameters used: $\gamma \in \langle 0; 1)$ is a forgetting factor and $\alpha \in (0; 1\rangle$ is a learning rate.

According to the equation (3), the Q-Learning algorithm updates only one value at a time. The learning speed can be enhanced by modification called *Eligibility Trace*, which enables the algorithm updates values of multiple past state-action pairs at one step. Such modification of Q-Learning is called Q-Lambda, or $Q(\lambda)$ algorithm. By introducing the error function, which is the fundamental for the eligibility traces-based approaches, the equation can be rewritten as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta e(s, a), \tag{4}$$

where the parameter error is defined for each state-action pair as follows:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) & \text{if } (s,a) \neq (s_t, a_t) \\ \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } (s,a) = (s_t, a_t) \end{cases} \tag{5}$$

The equation states that for all state-action pairs, there is an error function value that decays in time. If the state-action pair is used, the error function is increased by 1. Such a modified equation (4) means that *all state-action pairs are updated* in each step.

The decay parameter $\lambda \in \langle 0, 1\rangle$ defines the magnitude of the update of the previous states. In the

[1] Note that this theoretically requires environment with Markov Property.

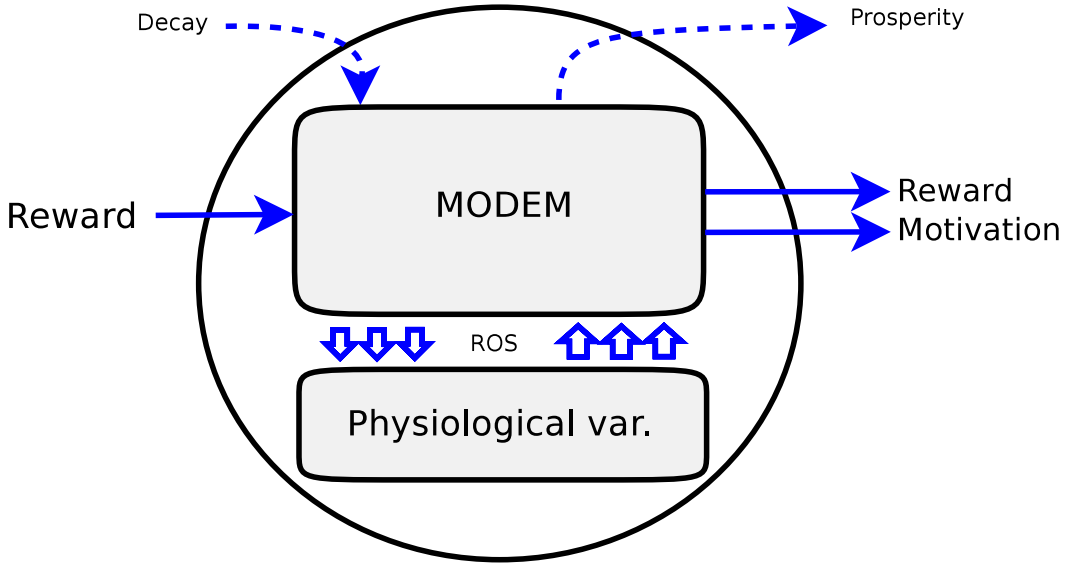


FIGURE 3. The Neural Module implementing the physiological state space. This node produces/represents motivation for the behavior which leads to the correct reinforcement. The value of its physiological variable decreases in each time step with given dynamics, and thus increases the motivation. Once the correct reinforcement is received, the value of the variable goes back towards the limbo area, where no motivation is produced. The value of the *Prosperity* output is defined as $P_t = 1 - SF_t$ (Mean State Distance to the limbo area).

case that $\lambda = 0$, pure one-step Temporal Difference (TD) learning is used. In the case of $\lambda = 1$, Monte-Carlo learning is obtained. Correct estimation of λ can improve the speed of learning, but also can cause oscillations in learning.

In the implementation of this Neural Module, the modification of the $Q(\lambda)$ algorithm is used. Here, the *Eligibility trace* is constrained to *finite length* of N previous steps, which saves computational resources and prevents bigger destabilization of learning convergence by an incorrect value of the λ parameter.

3.1.2. Action Selection Method in non-episodic experiments

A typical use of the $Q(\lambda)$ algorithm is in episodic experiments. In episodic experiments, the initial state of the environment is selected randomly. This helps toward uniform exploration of (and learning in) the entire state-space. In order to add domain-independence, the designed module has *to be able to operate in non-episodic experiments*. In non-episodic experiments (particularly those simpler with one attractor), it is necessary to achieve balance between knowledge exploitation and exploration/learning.

A typical Action Selection Method (ASM) for efficient knowledge exploitation is called a *Greedy strategy*, where the action with the highest utility is selected:

$$a_{t+1} = a^*_{t+1} = max_a Q(s_{t+1}, a_{t+1}). \quad (6)$$

This strategy may stick at a local optimum, so the ϵ-Greedy AMS is often used. In this ASM, parameter ϵ affects the amount of randomization. Random action is selected with probability of ϵ, while the Greedy strategy is followed with probability of:

$$P(a^*_{t+1}) = 1 - \epsilon. \quad (7)$$

Parameter ϵ therefore directly balances between exploitation of knowledge and exploration of the state space around the nearest attractor.

In order to provide efficient learning ability in non-episodic experiments, authors introduce an input to the Neural Module called *"importance"* which generally defines the *current need for "services" provided by the module*. For the $Q(\lambda)$ module, the importance input represents the *motivation for the behavior represented by this node*, see Fig. 2. The amount of randomization in the ASM should be indirectly proportional to the importance input. Here, the ϵ-Greedy ASM is used, but the randomization is defined as:

$$\epsilon = 1 - Importance. \quad (8)$$

By increasing the importance of the $Q(\lambda)$ module (increasing the motivation for executing a behavior represented by this node), the probability of taking the greedy action a^* increases. This means that the importance enables the agent to learn by exploration in free time and to exploit the information if needed.

3.1.3. Prosperity of the Reinforcement Learning Module

This subjective online heuristics estimates how efficiently the module is used in the current topology during the current simulation (task). Since the $Q(\lambda)$ module has to follow two antagonistic objectives (exploitation vs. exploration), it can be difficult to represent the efficiency of its use by a single value.

The heuristics that is currently being used is represented by the following equation:

$$P_t = \frac{Cover_t + MCR_t}{2}, \quad (9)$$

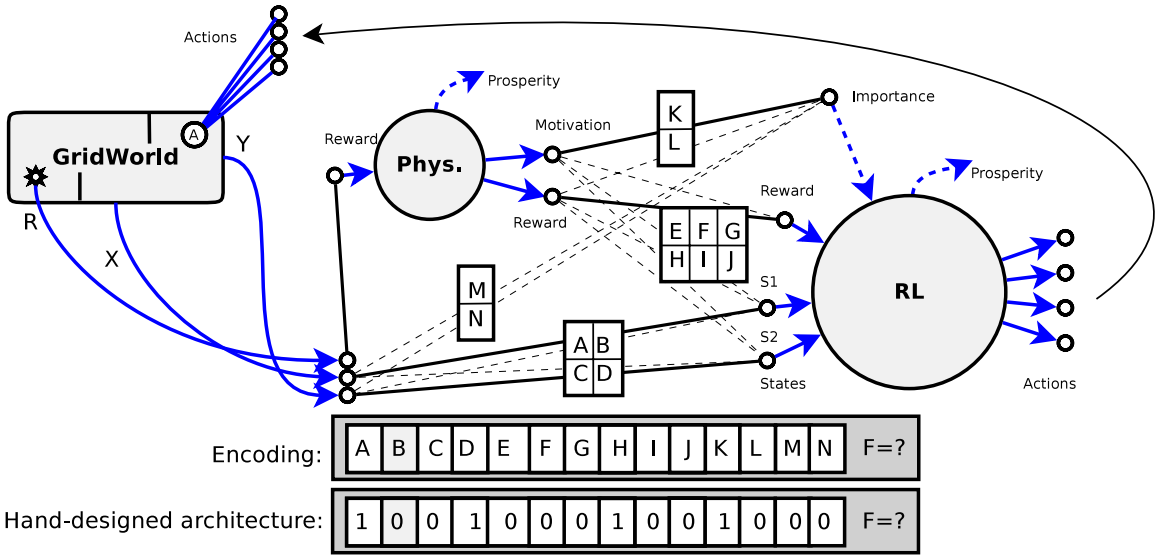


FIGURE 4. Scheme of mapping the genotype (vector of binary/real values) to the phenotype (working agent architecture). The Physiological Module is wired to the reward source in the map, and this determines the main goal of the agent (by the module's Prosperity value). All configuration inputs are unconnected, so the default parameters are used. Outputs of the Q-Lambda Module are directly wired to agent's actuators. The genotype of the *hand-designed architecture* is depicted at the bottom and its connections are highlighted in the scheme. Variables representing the state of the environment (X, Y coordinates) are connected to the data inputs of the Q-Lambda node. The reinforcement is connected to the Physiological Module, which produces motivation for the ASM and a reward for learning in the Q-Lambda Module.

where the Mean Cumulative Reward (MCR) is defined as the mean reward (R) received during the simulation until time step t. This represents the efficiency of knowledge exploitation:

$$MCR_t = \frac{\sum_i r_i}{i} \; \forall i = 0, 1, \ldots, t. \tag{10}$$

The value of $Cover_t$ represents how many states of the entire state-space have been visited so far:

$$Cover_t = \frac{\sum_{i \in Visited} s_i}{\sum_{s \in S}}, \tag{11}$$

This value represents exploration efficiency.

3.2. Motivation Source Module

In order to represent agents' needs, their physiology can be modeled [12]. It has been shown that decomposing the task into subtasks can help the RL to learn more efficiently, which is beneficial especially in more complex tasks or in tasks that are difficult for RL to learn [13]. The agents' needs can represent the motivation to execute/learn these subtasks of such a more complicated policy. The second Neural Module, which serves as a motivation source and holds one *physiological variable* is used here. The value of this variable decays with predefined dynamics in time. The value of 0 represents *purgatory area* and the value of 1 represents the *limbo area*. The amount of motivation is indirectly dependent on the physiological variable. In the limbo area, no motivation is produced, while the purgatory area represents the maximum motivation/need.

The simple dynamics of the physiological variable is defined as follows:

$$V_{t+1} = V_t - decay. \tag{12}$$

The amount of motivation that is produced is determined by applying the sigmoid to the inverse value of physiological variable V. The resulting amount of Motivation M at time t is:

$$M_t = \frac{1}{1 + e^{min + (max - min) \times (1 - V_t)}}, \tag{13}$$

where the min and max parameters are chosen so that the value of the variable $V_t = 0$ roughly corresponds to the motivation of $M_t = 1$. If the reward is received, the value of V_{t+1} is set to one, and therefore the motivation decreases towards 0, which switches the agent back towards exploration.

3.2.1. Prosperity of the Motivation Source Module

If the agent behaves efficiently enough, the mean motivation produced by this module is low. The mean motivation value can be expressed by the Mean State Distance to optimal conditions (SF), which is defined as follows:

$$SF_t = \frac{\sum_i d_i}{i} \; \forall i = 0, 1, \ldots, t, \tag{14}$$

where d_i is the distance of state variable V_i from the optimal conditions of $V = 1$. SF_t is computed

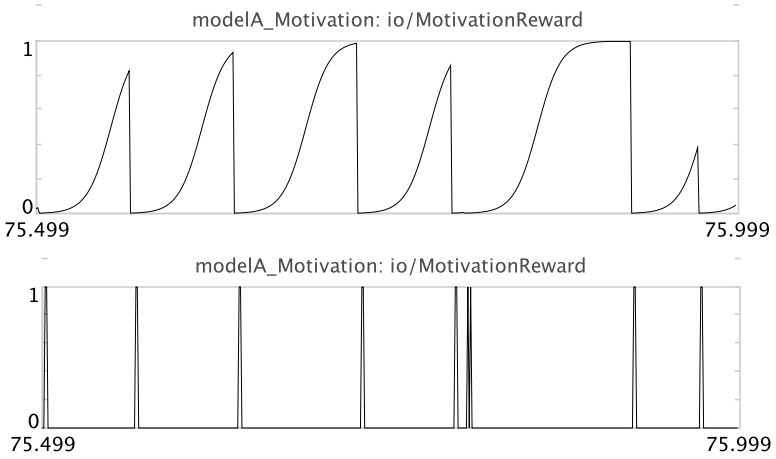


FIGURE 5. Observing the motivation-driven RL behavior of the architecture: the exploration vs. knowledge exploitation is dynamically balanced, according to the agent's current needs. The X-axis represents the simulation steps; the Y axis is value of the motivation/reward. Peaks in the lower graph represent the binary event of receiving the reward. These events correlate with the amount of motivation that is currently produced (upper graph). The speed of receiving the motivation depends on quality of the agent's knowledge (Fig. 6b and Fig. 6c) and the agent's current distance to the reward source.

online for each simulation step. Since the Prosperity is indirectly proportional, its value is computed as:

$$P_t = 1 - SF_t. \tag{15}$$

4. EXPERIMENTS

First, experiments which validate the expected functionality of particular Neural Modules are performed. Then their correct interaction is evaluated on the architecture,which is hand-wired for a given experiment. Two types of Evolutionary Algorithms (EAs) were used. The standard generational model of the Genetic Algorithm (GA) and its modification for vector of real-valued numbers called Real-valued GA (RGA) were used. Both, the GA and RGA were tested and compared on designing new architectures for a given task.

In the experiments, the agents are allowed to move in a 2D discrete world 15×15 positions in size. The map contains two obstacles and one source of motivation (see Fig. 6b). The agent has 4 actions — moving in four directions — and if the agent steps on a tale with the reward, a positive reinforcement is received. Therefore, the Q-Lambda Module is configured to have four outputs (four actions) and two inputs (X, Y coordinates on the map) sampled into 15 expected values (see Fig. 4). The value of physiological variable is configured to decrease with value of $decay = 0.01$ each step.

The simulated environment is also implemented as an ROS node and is also used in the NengoROS simulator as a Neural Module. The simulation is non-episodic; the agent is placed in the environment and is allowed to interact with the world for a predefined number of simulation steps.

4.1. DESCRIPTION OF HARDWIRED AGENT ARCHITECTURE

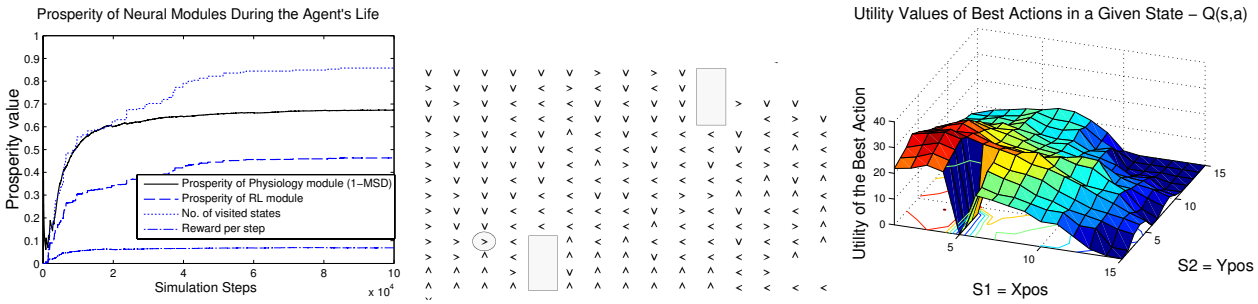
The modules are connected so that the Physiological Module receives a reward from the environment and produces motivation for the Q-Lambda Module. States of the environment are connected to the data inputs of the Q-Lambda Module. Actions produced by the Q-Lambda Module are then directly applied as actions of an agent in the environment.

The scheme of the hand-wired architecture represented as a hybrid artificial neural network is depicted in Fig. 4. Together, this network implements the motivation-driven reinforcement learning used in a non-episodic simulation.

4.1.1. SIMULATION OF HARDWIRED ARCHITECTURE

During the simulation, the Prosperity values of both the Q-Lambda and Physiological Module are observed. These should correlate with the successfulness of the behavior of the agent. Note that Q-Lambda Prosperity is composed of MCR_t and $Cover_t$ values (how successful the learning is and how efficient the exploitation is) and the Prosperity of the Physiological Module is defined as $1 - SF_t$ (defining how satisfied the agent is).

Figure 6a shows the course of these values in time. It can be seen that at about step 20000, the prosperity of the Physiological Module and the agent's average reward/step converge. Since the behavior is learned reliably at about step 20000, the agent is able to exploit the knowledge efficiently (fulfills the requirements for a reward faster) and therefore it has more time to explore. This causes further discovering of new states between steps 20000 and 40000. Figure 5 shows relatively stable behavior of the motivation-driven RL system during this part of the simulation. Here, the agent balances between exploration and exploitation



(a) Convergence of Prosperity values of both Neural Modules in time. Since the behavior is learned reliably around the step 20000, the agent is satisfied and systematically explores further new states.

(b) Visualization of knowledge learned by the agent during the first 10000 simulation steps. Each position in the table is filled with representation of action with the highest utility value.

(c) Utility value of the best action (see b) based on agent's current position in the map. The nearer to the reward source, the higher the expected outcome of the best action is.

FIGURE 6. Simulation of the hand-designed agent architecture. Each position in b) and c) represents one position in the map. There are two obstacles and one source reward in the environment. b) and c) show the agent's learned knowledge. Greedy ASM selects the action with the highest utility in each state. These actions are shown in b) and their utility is shown in c). Note that the utility values on the Z-axis are rescaled for better visibility.

of knowledge based on the current motivation value. The longer delay between satisfying the motivation, the further the agent was from the source of a reward (potentially unexplored area).

The knowledge that has been learned by the agent can be visualized in two ways. Figure 6b shows a graphical representation of the best action (a^*) based on the agent's current position in the map (state s of the environment). Convergence to the optimal strategy can be observed especially near the source of the reward. Figure 6c depicts the actual utility values in the $Q(s,a)$ matrix for the best action a^* in the state. The higher the surface is, the higher is the expected reward. Zero values represent unexplored states, such as obstacles. Note that the behavior of this architecture can be seen in the attached movie.

4.2. Principle of the evolutionary design of new architectures

Two types of simple Genetic Algorithm (GA) were used for the neuro-evolutionary design (optimizing the connection weights between modules) of new architectures here. As was mentioned above, the agent architecture is represented as an oriented graph, where the nodes are Neural Modules and the edges are the connection weights between their inputs/outputs. Similarly to the neuro-evolutionary design of ANNs [14], the topology of agent architecture is optimized by modifying the connection weights between nodes.

Each *individual* consists of a *genome* and *fitness* value. A genome is a vector of binary/real values representing the connection weights between modules. The fitness value represents the quality of a given architecture (its performance on a given task) and is determined by means of Prosperity values (see below). The generational model of the GA/RGA (Real-valued GA) is used here.

In this particular case, the mapping of genotype (genome) to phenotype (architecture) is depicted in the Fig. 4. The representation of the architecture is inspired in feed-forward ANN topologies, where the inputs/outputs between particular layers are fully connected. The weights of these connections are optimized by the GA/RGA.

The previous experiment suggested that during the simulation of 20000 steps the architecture should be able to successfully learn the desired behavior. The evaluation of an individual is therefore determined by means of Prosperity values of the Physiological Neural Module, and is obtained after simulation of the architecture for 20000 steps. The parameters of GAs and RGAs were empirically set to the following values. The population size is $PopSize = 50$, the number of generations $MaxGens = 80$. Each gene is mutated with probability of $pMut = 0.05$. The one-point crossover is applied to two individuals with the probability of $pCross = 0.8$. The GA mutation is defined as flipping the value of a given gene. For the RGA, the mutation was implemented as sampling the Gaussian Function with the standard deviation of $\sigma = 1$ with mean value of $\mu = gene_i$, where $gene_i$ is the value of mutated $gene_i \in \mathbb{R}$ (the constraints of interval are applied after the mutation).

4.3. Evolutionary design of new architectures

Since the prosperity of the Physiological Module is *determined by the agent's ability to learn* new knowledge and use it, the suitability of the wiring of both modules is represented in the Prosperity of the Physiological Module. We therefore define the Simple Fitness (SF) is defined, which is equal to the Prosperity of the Physiological Module:

$$P_{arch} = SF_{arch} = P_{phys} = 1 - SF_t, \qquad (16)$$

FIGURE 7. Comparing GA and RGA with Single Objective Fitness (*SF*). The fitness is defined as a Prosperity of the Physiology Module (but the courses of evolution for the composed-fitness *CF* look similar). The GA finds solution with similar quality faster than the RGA in both cases. Note that each result is averaged over 10 runs of the algorithm and fitted by a polynomial for better readability.

Parameters (see Fig. 4)	A E K	B F L	C G M	D H N	I	J
Hand-designed	1	0	0	1		
$SF = 0.625$	0	0	0	1	0	0
	1	0	0	0		
SOGA – Ind1	1	0	0	1		
$SF = 0.699$	0	0	0	1	0	0
	1	1	1	1		
SOGA – Ind2	1	1	0	1		
$SF = 0.697$	0	0	0	1	0	1
	1	1	1	1		
SORGA – Ind1	0	1	1	0.71		
$SF = 0.745$	0	0	0	1	1	0
	0.89	0	1	1		
SORGA – Ind2	0	0.51	1	0		
$SF = 0.723$	0	0	0	1	0	0
	0.76	0	0.3	0.44		

TABLE 1. Comparison of agent architectures' genomes: hand-designed, two SORGA-designed and two SOGA-designed architectures.

where P_{phys} is the Prosperity of the Physiological Module. This means that Single-Objective GA (SOGA) and Single-Objective RGA (SORGA) will maximize only the Prosperity of the Physiological Module.

Here, GA and RGA were used to design new architectures for the described environment. The graph in Fig. 7 compares the convergence of the two types of evolutionary design of agent architectures. Again, it can be said that SOGA is able to find a similarly fit solution faster than SORGA. The following sections will describe some typical automatically found architectures and their properties.

4.3.1. Analyzing new architectures found by SOGA

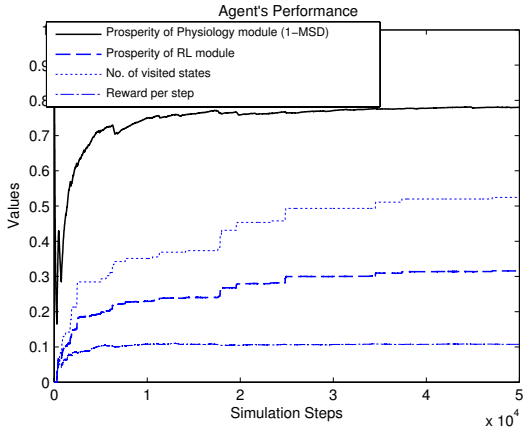
Table 1 shows several automatically-designed architectures by means of SORGA and SOGA, and compares them to the manually-designed architecture. SOGA found functional architectures relatively fast. Both selected typical individuals (marked as Ind1 and Ind2 in the table) have the following properties:

- The reward input of the Q-Lambda Module is wired correctly, so that the RL part works as expected.
- Both environment state variables, and also the motivation output of the physiology are connected to the motivation input of the Q-Lambda Module. This means that the motivation-driven RL is also used. The motivation is directly proportional to the motivation produced by physiology and to the agent's position/distance from the reward source.
- The binary reward output of the physiology is also connected to the motivation source, but this has no significant influence on the agent's behavior.

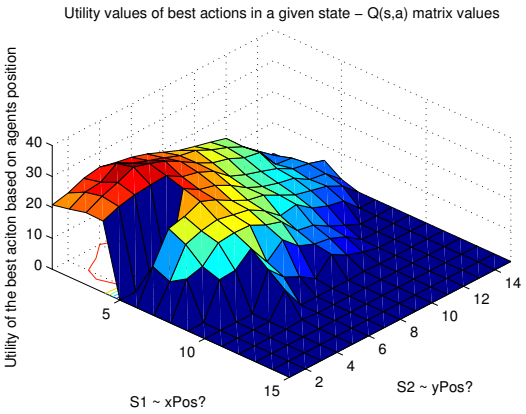
Also note that Ind1 has correctly wired state variables to the agent's position, while the Ind2 has one dimension "diagonalized" (both, the X coordinate and the Y coordinate are connected to the $S1$ variable). This means that only one half of the $Q(s, a)$ matrix was used here, but still the architecture still performed relatively well. Figure 8 shows the behavior of a typical best architecture found by SOGA. The architecture performs similarly to the hand-designed architecture (see Fig. 6). Compared to the hand-designed architecture, this architecture has bigger motivation to stay near the reward source, so that the overall prosperity of the Physiological Module is higher, while the number of explored states is lower.

4.3.2. Analyzing new architectures found by SORGA

Finding new architectures by means of SORGA took more than generations than for SOGA. However, SORGA has wider options for connecting modules. For example, it can be seen from the Table 1 that Ind2 used only half of the $Q(s, a)$ memory. Ind1 uses swapped coordinates with one dimension slightly diagonalized. In both architectures, motivation-driven RL is used. Again, the amount of motivation originates from the physiology and the agent's position in the map (the agent is therefore *afraid of going further away from the food*). Figure 9 shows the course of learning of Ind1 and the content of its memory. The knowledge is diagonalized (see Fig. 9b), but the learned data corresponds to the reward source. The position of the obstacle is also visible. The separated peak is caused by the fact that the reward output of the Physiological Module is connected to the $S1$

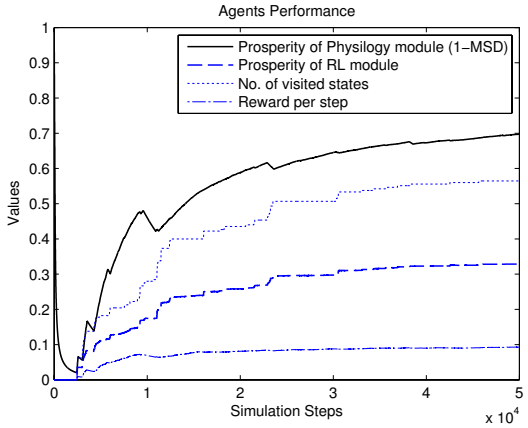


(a) Course of life of typical best agent found by the SOGA – Ind1. The behavior is similar to the hand-designed architecture (see Fig. 6a).

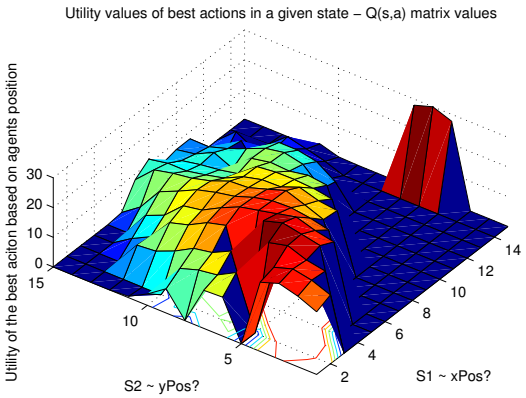


(b) Knowledge of selected agent found by the SOGA – Ind1. Representation is identical to the hand-designed architecture. Not all environment states are explored here.

Figure 8. Analyzing the typical architecture found by the SOGA (marked as Ind1). It can be seen that the higher motivation (combined from two sources) causes the agent to stay nearer the reward source (once found) than in the hand-designed architecture. Compared to this, the CORGA approach is able to weight the amount of importance produced by particular sources.



(a) Course of the life of the typical best agent found by SORGA – Ind1. Compared to the COGA-designed architecture (Fig. 8a), this agent learns more slowly (slower convergence of Prosperity of the Physiological Module). This is caused by sub-optimal representation of knowledge in the Q-Lambda Module.



(b) Knowledge of the selected agent found by SORGA – Ind1. The Q-Lambda Module has swapped axes and the value of the $S2$ variable is computed as follows: $S2 = X + 0.71Y$. This causes slower convergence of learning. The "peak" in the graph is caused by connecting the reward output to the $S1$ input.

Figure 9. Behavior and knowledge learned in the SORGA-designed architecture.Although the convergence of learning is slower, the overall results of the behavior are similar to the SOGA-designed architecture (Fig. 8).

input. This means that while receiving the reward, the perceived X position "jumps" to the maximum value.

4.3.3. Discussion

It has been shown that the agents' ability to weight between exploration and knowledge exploitation can be represented only by means of the Prosperity of the Physiological Module. Table 1 shows that all architectures found by SOGA and SORGA have notably higher fitness values than the hand-designed architecture. Furthermore, the results of SORGA (particularly Ind1) suggest that it is more efficient to use two components as a source of motivation: time (agent's physiology) and space (agent's position). This is a simple example of how evolution is able to correctly identify possibly hidden attributes of the task and employ then while designing architectures suitable for the task. Solutions provided by the evolution typically use less efficient representation of the knowledge in the SRP's memory, but are able to use Neural Modules in an unexpected manner. One of goals of our research is: to test how known algorithms can be employed in new, unexpected ways.

5. Conclusion

A novel approach for designing hybrid agent architectures, which is inspired by neuro-evolution has been presented. This approach uses the newly presented framework of Hybrid Artificial Neural Network Systems (HANNS). It searches for new topologies of these hybrid networks by modifying the connection weights between particular Neural Modules. This method enables semi-automatic design of agent architectures that are specialized for a given task.

It has been shown that this approach is able to do the following:

- Determine *whether and how to the module will be used* by connecting its outputs.
- Define the *form of knowledge representation in the module* by connecting its inputs.

It has been shown that, during the design of new architectures, the approach is able to pick the important aspects of the task and make efficient use of this knowledge for designing the architecture. This method of automatic design is able to find design solutions that are as good as, or even better than, those designed by hand. This can be particularly useful in designing autonomous systems, where the task is too complicated to be fully understood by the human designer.

In this particular case, evolutionary design was able to *choose the inner representation of a problem inside a Neural Module* (represent X, Y coordinates), to *discover inherent properties of the task* (that the reward source is near the coordinates to $0, 0$) and to *define the solution by wiring the connections between Neural Modules* (e.g., that the importance of knowledge exploitation is directly dependent on the agent's position). The RGA discovered that it is a more robust approach to use two independent sources of motivation; one based on the agent's position (environment state) and the other based on the agent's physiology.

Last but not least, particular nodes in the HANNS framework are implemented in a way that can be used in a variety of different (e.g., more complex) architectures and/or in a variety of modular systems in the future. For example, multiple (differently configured) RL modules can either compete (for learning/action selection of antagonistic goals) or cooperate to learn/execute one (hierarchically decomposable) behavior.

Acknowledgements

This research has been funded by the Dept. of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, under SGS Project SGS14/144/OHK3/2T/13.

References

[1] M. Quigley, K. Conley, B. Gerkey, et al. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*. 2009.

[2] F. Jiang, H. Berry, M. Schoenauer. The impact of network topology on self-organizing maps. In *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, GEC '09, pp. 247–254. ACM, New York, NY, USA, 2009. DOI:10.1145/1543834.1543869.

[3] J. Bullinaria. Generational versus steady-state evolution for optimizing neural network learning. In *In: Proceedings of the International Joint Conference on Neural Networks Ijcnn*. 2004.

[4] G. Auda, M. Kamel. Modular neural networks: a survey. *Int J Neural Syst* **9**(2):129–151, 1999.

[5] C. Eliasmith, C. H. Anderson. *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. The MIT press, Cambridge, ISBN: 0-262-05071-4, 2003.

[6] E. Crawford, M. Gingerich, C. Eliasmith. Biologically plausible, human-scale knowledge representation. In *35th Annual Conference of the Cognitive Science Society*, pp. 412–417. 2013.

[7] K. Mcgarry, S. Wermter, J. Macintyre. Hybrid neural systems: from simple coupling to fully integrated neural networks. *Neural Computing Surveys* **2**:62–93, 1999.

[8] Dept. Cybernetics FEE, CTU in Prague. *NengoROS*. `http://nengoros.wordpress.com/` [2014-10-01].

[9] Centre for Theoretical Neuroscience, U Waterloo. *Nengo*. `http://nengo.ca/` [2014-10-01].

[10] M. Wiering, H. van Hasselt. Ensemble algorithms in reinforcement learning. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* **38**(4):930–936, 2008. DOI:10.1109/TSMCB.2008.920231.

[11] J. Jiang. *A Framework for Aggregation of Multiple Reinforcement Learning Algorithms*. Ph.D. thesis, Waterloo, Ont., Canada, Canada, 2007. AAINR34511.

[12] D. Kadlecek, P. Nahodil. Adopting animal concepts in hierarchical reinforcement learning and control of intelligent agents. In *Proc. 2nd IEEE RAS & EMBS Int. Conf. Biomedical Robotics and Biomechatronics BioRob 2008*, pp. 924–929. 2008. DOI:10.1109/BIOROB.2008.4762882.

[13] T. Watanabe, T. Sawa. Instruction for reinforcement learning agent based on sub-rewards and forgetting. In *Fuzzy Systems (FUZZ), 2010 IEEE International Conference on*, pp. 1–7. 2010. DOI:10.1109/FUZZY.2010.5584788.

[14] J. Fekiac, I. Zelinka, J. C. Burguillo. A review of methods for encoding neural network topologies in evolutionary computation. In *Proceedings 25th European Conference on Modelling and Simulation ECMS*, pp. 410–416. 2011. ISBN: 978-0-9564944-2-9.