Coping with network dynamics using reinforcement learning based network optimization in wireless networks

Milos Rovcanin, Eli De Poorter, Ingrid Moerman, Piet Demeester

Ghent University - iMinds, Department of Information Technology (INTEC), Gaston Crommenlaan 8, Bus 201, 9050 Ghent, Belgium e-mail: milos.rovcanin@intec.ugent.be.

Abstract

Due to the increasing number of co-located networks, inter-network cooperation is gaining importance to obtain optimal network performance. To increase the network performance, increasingly advanced algorithms for optimizing co-located wireless networks are being proposed, including cognitive techniques. Networks are thus becoming capable of solving complex optimization problems on their own, without manual intervention. This paper investigates the inherent trade-offs that occur when using reinforcement learning techniques in dynamic networks: the need to keep the network running optimally at the same time whilst at the same time different (suboptimal) network settings need to be continuously investigated to cope with changing network conditions. To this end, two algorithms are investigated. The first one is a simplistic and extensively used "epsilon greedy" algorithm. A novel alternative is proposed, based on a logarithmic probability distribution function, well fitting to this specific use case and easily re-adjustable.

Keywords:
Network cooperation, self-awareness, reinforcement learning, linear approximation, network service negotiation, ϵ greedy, logarithmic state access distribution;

1. Introduction

Increasingly complex, heterogeneous and dynamic networks require intelligent resource usage, high efficiency, low levels of interference with co-located networks, while maximizing the life-time. Manual configuration and

tweaking of different (interfering) networks, along with a static configuration of their parameters, is becoming highly impractical. It can often cause an under-performance in regards to a given network requirements. Therefore, network solutions that dynamically support at run-time cooperation between devices from different types of networks are becoming a necessity.

Our research is motivated by the fact that a network's performance can be improved through a usage of certain combinations of services, provided by co-located networks (packet sharing, data aggregation, interference avoidance, MAC and routing protocols etc). To optimize co-located wireless networks, a wide range of such optimization techniques and services can be found in literature, ranging from interference avoidance, to dynamic power adaptations and shared routing and MAC solutions. Such a form of cooperation could improve individual network performances, using different combinations of configuration options than the one being pointed out as the optimal ones in a stand-alone case. Obviously, the major issue is to efficiently determine the optimal configuration parameters for all the participating networks, ensuring that the cooperation is mutually beneficial.

In [1], a communication framework was described that is capable of dynamically activating or deactivating a number of these optimization techniques (referred to as network services). This work did not yet include any directions about when to activate the different network services. As such, a reinforcement learning paradigm was utilized in [14] to develop an engine capable of determining the best performing set of the utility services for each cooperating network, in regards to their specific performance requirements. In other words, the engine is able to determine the optimal, joint set of services for the entire network.

The engine operates in two distinctive phases.

- An exploration phase considers an exhaustive search throughout the problem space to obtain all the relevant data.
- An exploitation phase utilizes a form of reinforcement learning called the Least Squares Temporal Difference (LSTDQ), in order to efficiently exploit the gathered data and discover the optimal set of configuration parameters.

The exploitation phase determines the performance of the different combinations of network services. The best combination of combination results

in the optimal network performance. In a dynamic environment, with changing network conditions or changing application requirements, parameter reconfiguration is commonplace. As pointed out before, the main trade-off of using a self-learning engine is its ability to keep the network in the optimal state, while being able to notice possible environmental changes that would demand reconfiguration of the network. The solution is a continuous probing of the sub-optimal service combinations.

In existing approaches, probing often relies on the well known and widely used "epsilon greedy" algorithm. This paper presents an alternative, simplistic and easily re-adjustable method, based on the logarithmic probing probability distribution. In order to select the optimal solution for different network environments, this paper analyzes for both approaches in detail (i) the response time to detect the new optimal state when network changes occur and (ii) the time the system spends in the (sub)optimal state.

The remaining of the paper is structured as follows. The following section 2 brings a brief overview of the related work in this area of research. Next, mathematical fundamentals of reinforcement learning and LSTDQ algorithm are given in Section 3. Section 4 focuses on explaining the suitability of the LSTD based algorithm for solving the inter-network optimization problem, described in [14]. Section 5 describes the experimental set-up and implementation details. A thorough analysis of the two probing algorithms is presented in Section 6. Finally, Section 7 concludes the paper.

2. Related work

Reinforcement learning typically transforms a given problem into a dynamic, discrete and stochastic Markov Decision Process. In each step, the process resides in a certain state. A large state space is one of the major problems that stands out in the learning systems, since it directly influences a learning time. For example, Q-learning [4] with an inadequately chosen input data and an inexperienced choice of function approximations often leads to a poor scaling of the size of a state space. Further more, Q-learning, along with some other fundamental RL algorithms, based on temporal difference [5], shows linear dependence on size of the state space, which is too slow for the real life problems.

LSPI represents an approximate reinforcement learning method, used in generalized situations. Approximated learning tends to generalize a given problem to a new situation where the learning time and space are far more

reasonable. This way, the algorithm's complexity becomes independent of the size of the problem space. Paper [6] surveys a number of approximate policy iteration issues, related to our research: convergence and rate of convergence of approximate policy evaluation methods, exploration issues, constrained and enhanced policy iteration etc. The main focus is on the above mentioned LSTDQ and its scaled variant algorithm.

Research published in [7] tests and proves the convergence of a model free, approximate policy iteration method that uses linear approximation of the action-value function, using on-line SARSA updating rules. The update rule is how the algorithm uses experience to change its estimate of the optimal value function. SARSA updating is exclusively used in an on-policy algorithms, where the successor's Q value, used to update the current one, is chosen based on the current policy and not in a greedy fashion, as with Q-learning.

There is also a significant number of optimization techniques, used for solving multi-objective optimization problems in heterogeneous WSNs. For example, in [8] authors propose evolutionary algorithms (EAs): NSGA-II (Non-dominated Sorting Genetic Algorithm II) [9] and SPEA-II (Strength Pareto Evolutionary Algorithm II) [10] as tools for solving reliability and packet delay issues in a heterogeneous WSN.

Simultaneous optimization of the high network lifetime and coverage objectives, tackled in [11]. Authors state that the previous methods either tried combining the two objectives into a single one or constraining one while optimizing the other. The novel approach employs the multi-objective evolutionary algorithm based on decomposition - MOEA/D [12] as a feasible solution.

It is important to notice that a policy iteration RL methods so far have been used on single objective problems, while the aforementioned multi-objective optimization techniques are computationally demanding. Especially when used on a problem where objectives are frequently changing, thus demanding often re-runs of the optimization algorithm.

To our knowledge, neither of the mentioned methodologies have been used in the process of service wise multi-objective heterogeneous network optimization - service negotiation. Our previous research [13] provides the conceptual framework that can be used to tackle this problem.

3. Reinforcement learning

3.1. Fundamentals

Reinforcement learning approach [17] [18], models a problem as a Markov Decision Process (MDP). Decision maker passes through a continuous or discrete space of states $S = s_1, s_2, s_3, ..., s_n$. In the continuous case, $n \to \inf$. Taking actions causes a switch between states. Set of available actions can also be continuous or discrete, $A = a_1, a_2, a_3, ..., a_n$. A reward is given upon taking each action. Learning, in this case, considers using an actions that maximizes a given reward at each state:

$$Q(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) maxQ(s',a') \tag{1}$$

Above given is the well known Bellman equation, where $Q(s,a)$ represents the state-action function, also known as the Q function. It assigns a numeric value to every state-action pair from the problem's state-action-space, based on the immediate reward $r(s,a)$, given for taking an action a at the state s and the future expected reward $\sum_{s'} P(s'|s,a) maxQ(s',a')$. The last argument includes the state-action transition probability, which is generally not known a priory.

3.2. LSTDQ fundamentals

LSTDQ was introduced by M.G.Lagoudakis and R.Parr as a part of the well known Least Squares Policy Iteration algorithm (LSPI) [19] [16]. Regardless the fact that the modifications we made to our previous work, published in [14], are not related to the fundamentals of the LSTDQ algorithm, in order to clearly justify the usage of the LSTDQ algorithm and to better understand the changes that have been made to the original approach, the section explaining the mathematical fundamentals of the LSTDQ algorithm is taken from [14] and presented in this subsection.

The basic idea of the algorithm is to represent Q function as a linear combination of *basis functions* (problem features) and their respective weights:

$$Q(s,a;w) = \sum_{k} \phi_j(s,a)\omega_j \tag{2}$$

The value function can also be presented in a matrix form as:

$$Q^{\pi} = \Phi\omega \tag{3}$$

where the $|S||A|$ dimensional Φ matrix contains values of basis functions, defined for each state/action (s, a) pair and π designates the decision making policy that is being used. Generally, the number of basis functions is much smaller than the number of state/action values, $k << |S||A|$. A proper example of a basis function would be: network's duty cycle, average time spent in a radio receiving/sending mode, residual energy of nodes etc.

Combined, equations (1),(2) and (3), produce the matrix outlook of the Bellman equation, modified in accordance with the approximations introduced by the LSTDQ algorithm:

$$\Phi\omega = R + \gamma P^{\pi}\Phi\omega \tag{4}$$

The reward R is given upon transferring from one state to another, with γ being a discount factor. Assuming that all the basis functions are independent (columns of a Φ matrix):

$$\Phi^T(\Phi - \gamma P^{\pi}\Phi)\omega^{\pi} = \Phi^T R \tag{5}$$

The weight factors are calculated by solving the following system:

$$\omega = A^{-1}b \tag{6}$$

$$A = \Phi^T(\Phi - \gamma P^{\pi}\Phi) \tag{7}$$

$$b = \Phi^T R \tag{8}$$

Here, P^{π} represents a transition probability matrix, describing the probability of a certain state/action pair sequence ((s, a), $(s', \pi(s'))$.

In general, P^{π} and R are not known a priori. The values they contain must be learned from sampled data in order to determine matrices A and b. A set of samples from the environment can be represented as $D = (s_{d_i}, a_{d_i}, s'_{d_i}, r_{d_i}|i = 1, 2, ..., L)$, where (s'_{d_i}) is sampled using $P(s'_{d_i}|s_{d_i}, a_{d_i})$, which corresponds to a decision making policy that is used at the moment. The approximate versions of $\Phi, P^{\pi}\Phi$ and R are generated in the following way:

$$\hat{\Phi} = \begin{pmatrix} \phi(s_1, a_1)^T \\ ... \\ \phi(s_n, a_n)^T \end{pmatrix} \widehat{P^{\pi}\Phi} = \begin{pmatrix} \phi(s'_1, \pi(s'))^T \\ ... \\ \phi(s'_n, \pi(s'))^T \end{pmatrix} \hat{R} = \begin{pmatrix} r_1 \\ ... \\ r_2 \end{pmatrix} \tag{9}$$

Matrices A and b can now be approximated in the following way:

$$\hat{A} = \hat{\Phi^T}(\hat{\Phi} - \gamma \hat{P^{\pi}} \Phi) \tag{10}$$

$$\hat{b} = \hat{\Phi^T} \hat{R} \tag{11}$$

As the number of samples grows, the consistency between the approximated and true values of matrices A and b ($\hat{A}$ and $\hat{b}$) also grows. This can be mathematically described in the following manner:

$$E(\hat{\mathbf{A}}) = \frac{L}{|S||A|}\mathbf{A} \tag{12}$$

$$E(\hat{b}) = \frac{L}{|S||A|}b \tag{13}$$

where L represents the cardinal number of the sample set. Notice that a new sample from the same state/action space, $(\hat{A_1}, \hat{b_1}, \hat{A_2}, \hat{b_2})$ can be added to an existing set to yield an even better approximation:

$$\hat{\mathbf{A}} = \hat{\mathbf{A}_1} + \hat{\mathbf{A}_2} \tag{14}$$

$$\hat{b} = \hat{b_1} + \hat{b_2} \tag{15}$$

4. Applying LSTDQ to an inter-network cooperation process

LSTDQ was primarily used in an off-line type of approach that demands a training set of samples to be formed before the initiation of a learning process. Each policy iteration selects samples from the training set and feeds them inside the LSTDQ. Run-time and learning processes, in this case, are separated.

Within an on-line approach, samples are gathered during the coarse of a learning process. In the extreme case, weights are recalculated after each new sample is collected. A decision making policy is therefore being improved continuously. A cross-comparison between on-line and off-line LSPI algorithms is elaborated in [20].

When it comes to modeling, there is no fundamental difference between the two approaches. There is a set of functional prerequisites that must be met in order to initiate and conduct the learning process.

- Define system states
- Define actions
- Define the basis functions
- Collect measurements
- Calculate Q-values
- Calculate rewards

The following explanation regarding each item is taken from [14].

4.1. Defining system states

Every state represents a distinct combination of the currently active network services. Reducing the initial number of system states speeds up the exploration phase of the algorithm, since the exhaustive search is performed over the entire state space. Some states can be discarded a priori, by the system architect. This relates to a situation where a combination of network services is not allowed.

The number of states can further be reduced during the exploitation phase. A state can be discarded if the recorded performance was bellow acceptable. This will strongly depend on the network requirements and metrics that are used for performance evaluation.

4.2. Defining actions

Activation or deactivation of a certain network service is considered as action. Consequently, the number of available actions at each state depends on the number of available services that can be switched on or off. In terms of changing states upon taking a certain action, our our algorithm is non-ambiguous, which means that $P(s'|s,a) = 1$ for every (s,a,s') tuple. In other words, an action taken at each state can lead to one and only one state.

4.3. Defining basis functions

Basis functions are used to describe the system performance during a single learning episode. They are crucial in the process of calculating Q values for each state/action combination. When comparing two action sets, the one that produces stronger response to the same change in network parameters

is considered to be the more effective one. It produces a larger difference between Q values, thus making it easier to enforce certain decision making paths. LSPI requires the basis functions to be linearly independent to prevent overlap when calculating the rewards.

Increasing the number of basis functions improves the accuracy with which the network performance is modeled. Acquiring information regarding basis functions typically introduces an additional communication overhead. As such, as the number of monitored properties of the network increases, a cost-effectiveness analysis in regards to the additional collection overhead is advised.

Basis functions are mainly needed for calculating the optimal actions for each state. However, it is worth noting that sudden changes in basis functions can also give an indication of a sudden change of network conditions. These can trigger alerts to the network administrator and, can require the re-initiation of a learning process.

4.4. Collecting samples

During the exploration phase, samples are collected according to an exhaustive search methodology. In our case, it is a random walk through a problem space. The searching algorithm keeps track of the visited states, so none of the states is visited twice before the process is over.

During the exploitation phase, samples are collected in accordance to the adopted decision making policy. This continuous sample collection is performed for two reasons:

- Fine tuning of the weight parameters $W = w_1, w_2, w_3, ..., w_k$
- Detection of a network condition change

Performance of the two different policies will be evaluated in details in the following sections.

4.5. Calculating rewards

It is useful to enforce an upper limit to the contributions of each network metric. Otherwise, a service combination that significantly 'overshoots' only one requirement can receive a higher reward than the one performing somewhat worse, but equally accomplishing all the given requirements. The following rewarding function is designed to prevent such a behavior:

$$R_i = 1 - e^{-3\frac{\phi_i}{\phi_{goal}}} \tag{16}$$

Rewards, in this case, increase very slowly once the requirements are met (see Figure 1). If the requirements do not describe an upper performance limit, rewards can be unlimited.

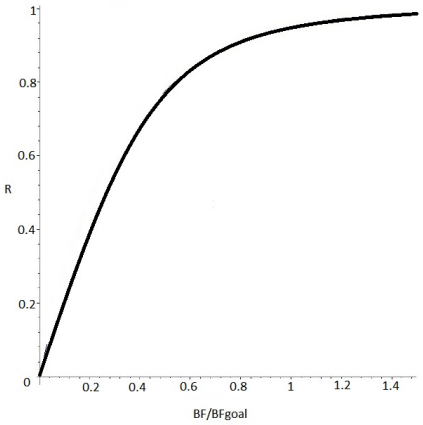


Figure 1: Rewards are calculated using the relative difference between the basis function (BF) values collected at the end of an episode and the desired values. The function also sets up a horizontal asymptote to an associated reward, thus making sure the reward increases slowly once the requirements are met.

5. Experimental setup and implementation details

5.1. Use case and the experimental set up

A fairly simple case of only 4 system states is examined in [14]. For the purpose of testing the modified approach, a more complex, this publication utilizes a more complex fictive use case. It will serve as a proof of concept and simultaneously provide an insight into the scalability of the original idea. The use case is designed to evaluate the reasoning engine's capability of perceiving differences between the system states (different combination of network services), its capability to distinguish between "good" and "bad" states and its versatility to adopt to dynamic network changes.

Similar to the set up described in [14], the concepts are evaluated for a use case in which two co-located, interleaved wireless sensor networks (see

Figure 2) are present. In the remaining sections of the paper, they will be referred to as networks A and B. Communication between the two networks is possible using a similar set-up as described in [13]: both networks collect network statistics in the sink and sinks can communicate with each other using wired technologies.

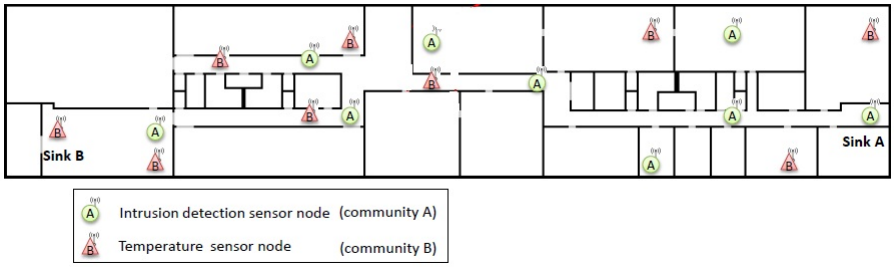


Figure 2: Interleaved networks A and B - as a part of the WiLab.t testbed [22], located at the iMinds research facilities, University of Ghent, Belgium

A set of (fictive) network services is available to both networks. Network A provides fictive services $N_{S1} = [ServiceA, ServiceB]$. Similarly, network B provides services $N_{S2} = [ServiceC, ServiceD]$. Whereas [14] measured the performance of these services in real-time and concluded that the optimal services could indeed be identified, this paper focuses on the adaptivity (in terms of coping with network dynamics) and convergence speed (in terms of exploration episodes). As such, the performance of the network under any possible combination of these services is described using fabricated measurements. The measurements are defined according to a simple set of rules, explained in the following section 5.2.

5.2. Implementation details

The system state is determined by the joint combination of active/non-active services in both networks. Given the set of four services N_{S1} and N_{S2}, the total number of states is $N_{states} = 16$. Service combinations are given on Figure 3.

State 0 (no services activated) is taken as the reference, state CB represents the worst performing service combination, while DCA represents the optimal service set. To calculate Q values for each state-action pair, two basis

State number	Service combination				State
	service **D**	service **C**	service **B**	service **A**	
0	0	0	0	0	**0**
1	0	0	0	1	**A**
2	0	0	1	0	**B**
3	0	0	1	1	**BA**
4	0	1	0	0	**C**
5	0	1	0	1	**CA**
6	0	1	1	0	**CB**
7	0	1	1	1	**CBA**
8	1	0	0	0	**D**
9	1	0	0	1	**DA**
10	1	0	1	0	**DB**
11	1	0	1	1	**DBA**
12	1	1	0	0	**DC**
13	1	1	0	1	**DCA**
14	1	1	1	0	**DCB**
15	1	1	1	1	**DCBA**

Figure 3: System states as combinations of active network services from networks A and B

functions are used: ϕ_1 and ϕ_2. We emphasize that the measurements regarding basis functions, as stated before, are fabricated. Q values are calculated in the following manner:

$$Q(s, a) = \phi_1\omega_1 + \phi_2\omega_2 \tag{17}$$

The cognitive engine is allowed to switch between any of the two states, which means that the set of 16 actions ($A_c = a_1, a_2, a_3, ..., a_{16}$) is available at each state. After a number of state changes, the engine is expected to determine which state is the optimal one and force that respective service combination. Forcing a certain service combination in this context means forcing transitions from any given state to the optimal one. If the system is already in the optimal state, the optimal decision would be to remain in it.

5.3. Exhaustive exploration phase

Before the engine starts, an exploration phase is used to measure the performance of a number of states. Thanks to the unique property of the MDP we created, the N_{states} episodes are enough to perform an exhaustive exploration over the entire state/action space. This can be illustrated with the following example. Let's say that the state the MDP is transferring into into is the state S_1. It makes no difference what state preceded

the current one, the performance of the current state does not depend on the states that were selected before. This implies that all the transitions $S_{1,2}, S_{2,1}, S_{3,1}, ..., S_{Nstates}, 1$ can be updated with the same values.

The end result is the initial set of weight factors $W = [\omega_1, \omega_2]$. In addition, a corresponding set of Q values is calculated for every state-action pair. Figure 4 depicts the initial Q values, calculated for each state upon completion of the algorithm's *exploration phase*.

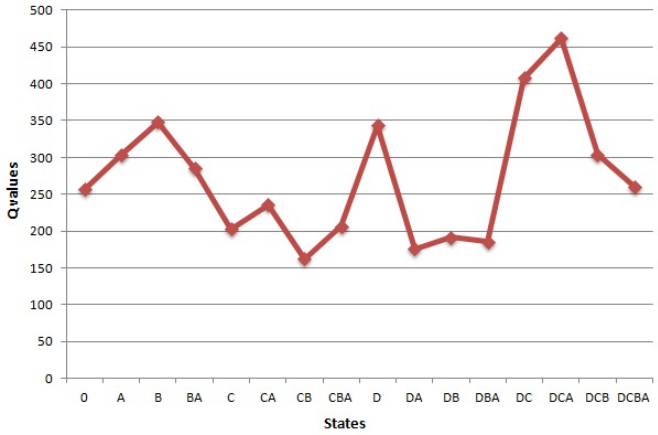


Figure 4: Q values describing network's performance when utilizing a given service combination. Values are calculated upon completion of the *exploration phase*

The exhaustive exploration results in an initial selection of "good" states (high Q-value) and "bad" states (low Q-value). However, in dynamic networks, relying solely on these initial results might lead to a sub-optimal performance, once the conditions in the network change. As such, the *exploitation phase*, initiated at the end of the *exploration phase*, utilizes these Q-values to cope with changing network conditions.

5.4. Exploitation phase

The exploitation phase is initiated once the exploration phase is terminated. The main challenge for the reasoning engine during this phase is to balance between keeping the optimal configuration setup active and checking if the performance of the sub-optimal states has changed as the result network disturbances that might happen over time. Therefore, the two main objectives of the reasoning engine are the following.

- Update the sample set with the new samples after each learning episode and re-calculate the ω factors and the respective Q values.
- Detect the occurrence of a network condition disturbance and reshape the decision making policy accordingly.

The trade-of between selecting the known optimal state and exploring other states depends on the methodology applied for the state inspection. This paper investigates two different methods.

- An "ϵ greedy" based approach.
- A new approach based on a logarithmic distribution of the state access probability.

The "ϵ greedy" exploration technique [21], is a widely adopted method. It is based on simple rules: with the ϵ probability, the reasoning engine picks action with the highest Q value. With $1-\epsilon$ probability, the action is picked in a random fashion. In the second approach, proposed in this paper, the states are investigated according to a probability distribution based on a states' Q values and the "disturbance probability factor" - α.

Both approaches will be explained in details in the following sections.

6. Results and discussions

To evaluate the advantages of both approaches, the following situations are investigated in detail.

- Performance of the reasoning engine in a steady network environment
- Performance of the reasoning engine after an unpredicted network disturbance occurs

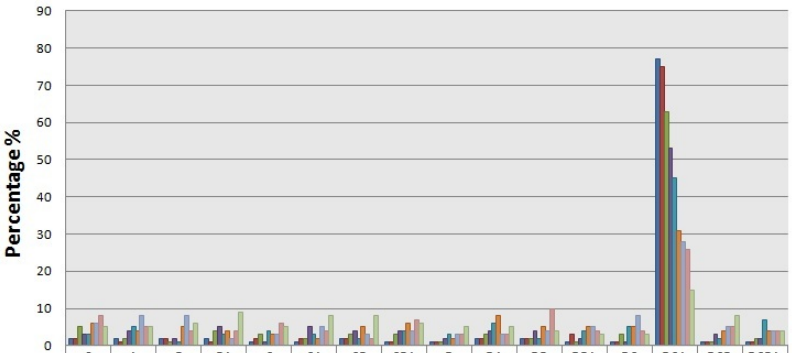
6.1. Exploitation based on the "ϵ greedy" algorithm

Using this approach, with $1 - \epsilon$ probability, the action is picked in a random fashion. As such, the ϵ factor dictates the frequency with which sub-optimal states are investigated. Investigating sub-optimal states is necessary to detect any possible changes of the network's behavior. However, at the same time, the network should be kept in the optimal state to result in optimal performance.

6.1.1. Steady network conditions

Figure 5 depicts for different epsilon values the percentage time that was spent in each state during 100 learning episodes of the *exploitation phase*. The best performing state is DCA, in wich service A is activated in the Network A and services C and D activated in the Network B. Service B should be kept inactive.

After the exploration phase is over and the initial Q values are calculated, this state is clearly the favored one in the figure. Instances of the algorithm with the lower ϵ values tend to keep the system in the optimal state for as much as possible, whereas using higher ϵ values promotes exploring other (non-optimal) states. The percentages vary from 77 down to only 15 percent, in cases when ϵ was set to 0.1 and 0.9, respectively. It is worth noticing that, for all ϵ values of 0.4 and lower, our cognitive engine keeps the system in the optimal state for more than 50 percent of the time.



	0	A	B	BA	C	CA	CB	CBA	D	DA	DB	DBA	DC	DCA	DCB	DCBA
ε=0.1	2	2	2	2	1	1	2	1	1	2	2	1	1	77	1	1
ε=0.2	2	1	2	1	2	2	2	1	1	2	2	3	1	75	1	1
ε=0.3	5	2	1	4	3	2	3	3	1	3	2	1	3	63	1	2
ε=0.4	3	4	2	5	1	5	4	4	2	4	4	2	1	53	3	2
ε=0.5	3	5	1	3	4	3	2	4	3	6	2	4	5	45	2	7
ε=0.6	6	4	5	4	3	2	5	6	2	8	5	5	5	31	4	4
ε=0.7	6	8	8	2	3	5	3	4	3	3	4	5	8	28	5	4
ε=0.8	8	5	4	4	6	4	2	7	3	3	10	4	4	26	5	4
ε=0.9	5	5	6	9	5	8	8	6	5	5	4	3	3	15	8	4

Figure 5: Percentage of time (Y axis) the system spent in the corresponding states (X axis), during a 100 episodes long exploitation phase

During the exploitation phase, ω factors are being constantly recalculated and Q values reshaped. Figure 6 illustrate this process in three distinct cases, with the ϵ values set to 0.1, 0.5 and 0.9, respectively:

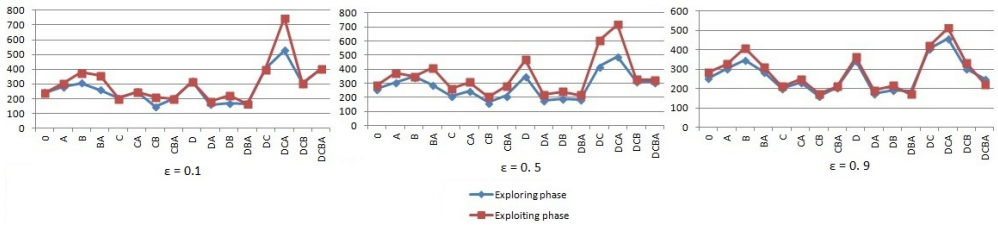


Figure 6: Behavior of the Q values during the exploitation phase, in regards to different values of the ϵ factor. ϵ is set to 0.1, 0,4 and 0.9, respectively

Exploration policies with lower ϵ values shape the Q values so that the optimal one is being increased at the highest rate. It appears as if the the sub-optimal ones are being repressed. This is an expected behavior since every visit to a certain state shapes the ω factors in its favor, thus constantly increasing the corresponding Q values. In contrast, in the case with an almost completely random policy ($\epsilon = 0.9$), the states are picked almost uniformly, without favoring any in particular. Therefore, Q values remain shaped do not change significantly after the exploration phase.

6.1.2. Reaction to a network condition disturbance

During the network operation, the performance of different states will vary depending on outside conditions. To analyze the resilience of the reasoning engine to these changes, Figure 7 depicts the number of episodes needed to detect an (extreme) condition change scenario in which the best and worst performing states switch places (DCA - CB).

Results are averaged over 10 trials of 250 learning episodes. The quickest response is achieved with $\epsilon = 0.9$ (13.2 episodes), since this mechanism checks system states in a near-uniform manner. With $\epsilon = 0.1$, it takes almost 120 episodes, in average, to react on disturbances and re-shape Q values accordingly. An exponential function can be fitted to above mentioned results to approximate the dependency between the ϵ factor and number of episodes needed to detect condition disturbances:

$$N_{episodes} = 130e^{-2.75\epsilon} \tag{18}$$

Results presented in sections 6.1.2 and 6.1.1 reveal one major issue of the ϵ-greedy approach - how to find a compromising solution for the ϵ value, so that the system is kept in the optimal (or the nearest-to-optimal states)

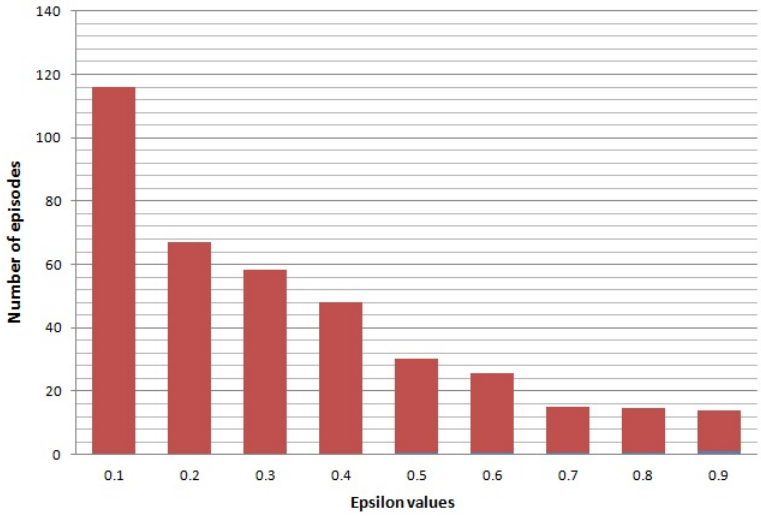


Figure 7: Number of episodes the reasoning engine required to detect a network condition disturbance. The results are given in respect to different values of the ϵ exploration factor

for as long as possible, while being able to "quickly" react to environmental changes? To illustrate this, with $\epsilon = 0.5$, the system is kept in the optimal state most of 45 percent of time, while it took the engines on average 30 episodes to detect condition changes. Fixing the ϵ to a fixed value throughout the entire exploitation phase obviously does not provide acceptable results neither from the condition change versatility nor from the optimality point of view. One simple improvement of algorithm's efficiency during the exploitation phase is described and evaluated in the following sub-section.

6.1.3. A simple method for increasing the algorithm's convergence speed

In setups where the worst case scenario is improbable (moderate network dynamics), an intuitive and fairly simple method can be used to improve the performance of the algorithm during the exploitation phase: service combinations that are below 50 percent of the optimal performance are discarded prior to initiation of the exploitation phase (Figure 8):

States C, CB, CBA, DA, DB, DBA are ruled out. The reduced set of 10 remaining states is taken into account during the exploitation phase. Figure 9 describes the percentage of the number of episodes that are spent in each state during a 100 episodes long exploitation phase:

As expected, there is no significant difference in the number of episodes

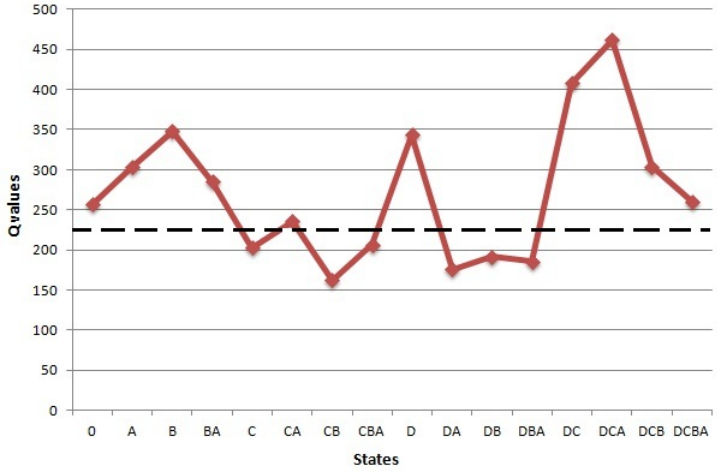


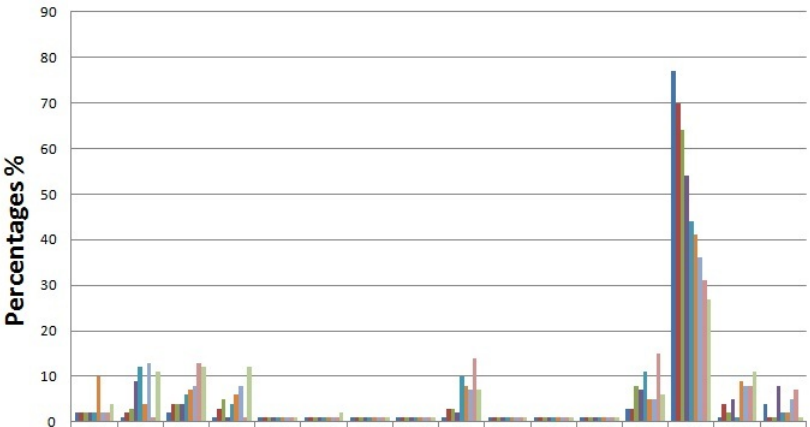
Figure 8: Applying a threshold on Q values, prior to initiation of the *exploitation phase*. Threshold is assigned with the value that matches 50 percent of the highest Q value

spent in the optimal state for almost all the values of ϵ. However, after applying a threshold, the number of episodes spent in the worst states is significantly reduced. Furthermore, the number of episodes needed to detect network disturbances has decreased significantly, as depicted on Figure 10.

As such, using this simple speed up adaptation results in improved network disturbance capabilities except for the case when the ϵ factor is set to 0.1. This is an expected behavior, since the sub-optimal states are almost never checked for such low ϵ values. The relation between the ϵ factor and the number of episodes needed to notice network disturbances can be approximated using the following formula:

$$N_{episodes} = 100.5e^{-3.28\epsilon} \tag{19}$$

When considering the fixed value $\epsilon = 0.5$, the system is kept in the optimal state 44 percent of time, with an average response time of around 15 episodes. Moreover, an additional 33 percent of time is spent in the nearest-to-optimal states (DC, D, B), as opposed to only 12 percent when no threshold is applied. As such, even a simple speed up procedure can still yield a considerable performance improvement.



	0	A	B	BA	C	CA	CB	CBA	D	DA	DB	DBA	DC	DCA	DCB	DCBA
ε=0.1	2	1	2	1	1	1	1	1	1	1	1	1	3	77	1	4
ε=0.2	2	2	4	3	1	1	1	1	3	1	1	1	3	70	4	1
ε=0.3	2	3	4	5	1	1	1	1	3	1	1	1	8	64	2	1
ε=0.4	2	9	4	1	1	1	1	1	2	1	1	1	7	54	5	8
ε=0.5	2	12	6	4	1	1	1	1	10	1	1	1	11	44	1	2
ε=0.6	10	4	7	6	1	1	1	1	8	1	1	1	5	41	9	2
ε=0.7	2	13	8	8	1	1	1	1	7	1	1	1	5	36	8	5
ε=0.8	2	1	13	1	1	1	1	1	14	1	1	1	15	31	8	7
ε=0.9	4	11	12	12	1	2	1	1	7	1	1	1	6	27	11	1

Figure 9: Percentages (Y - axis) are given in respect to the corresponding values of the ϵ factor, for every defined system state (X - axis), upon applying a simple efficiency-improving procedure

6.2. Exploitation based on a non-uniform state/action probability distribution

The "ϵ greedy" algorithm is not designed to discriminate between the "good" and the "bad" states. In other words, near-optimal states will be investigated as frequently as the states with a significantly lower performance. In many use cases, this might lead to an unnecessary waste of time and resources. Setting up a performance threshold helps, but there is no general rule to determine the optimal value of this threshold.

Therefore, we propose an alternative approach: during the exploitation phase, the reasoning engine will switch between the states with probabilities that are calculated using their Q values and a factor that describes the probability of a major network disturbance: the σ factor.

We use a logarithmic function to distribute state switch probabilities:

$$P = Q^{\alpha} log Q \tag{20}$$

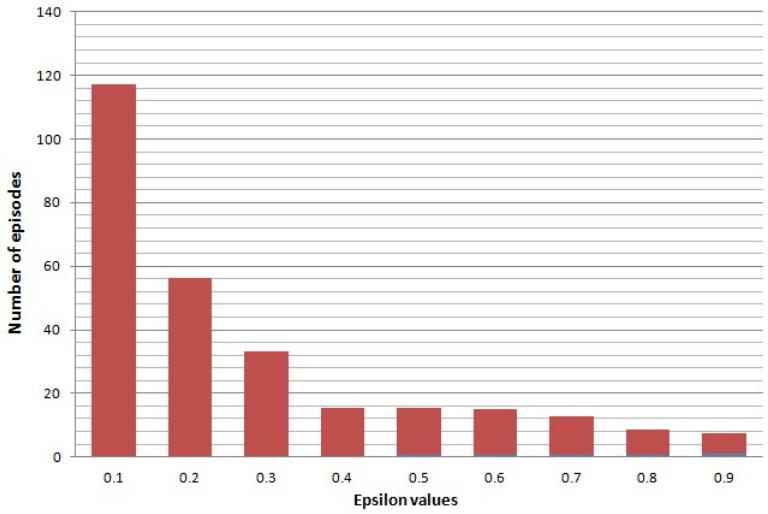


Figure 10: The numbers of episodes the reasoning engine needed to detect network condition disturbances, after applying a simple speed up rule. The results are given in respect to the different values of the ϵ exploration factor

This particular function, for $\alpha \geq 1$, suits our need to more precisely differentiate between the better and the worse performing system states. The function curve (Figure 11) describes the probability of selecting a specific state. Due to the logarithmic nature of the formula, states with poor performance are investigated very infrequently, as opposed to states that are (near) optimal.

Higher values of the α factor mean a steeper functional curve, which implies that only the highest performing states will be investigated during the exploitation phase, which is ideal for networks with a relatively steady behavior. On the other hand, lower α values give the less optimal states more chance to get investigated and are thus better suited for strongly dynamic environments. By manipulating the α factor, this function gives us the ability to selectively improve the probability of better rated state/action pairs, while suppressing the others. The reinforcement learning paradigm is satisfied by assigning the highest probability to those pairs with the highest Q values. To illustrate the logarithmic nature of probability distribution, the results are sorted in an ascending order (see Figure 12).

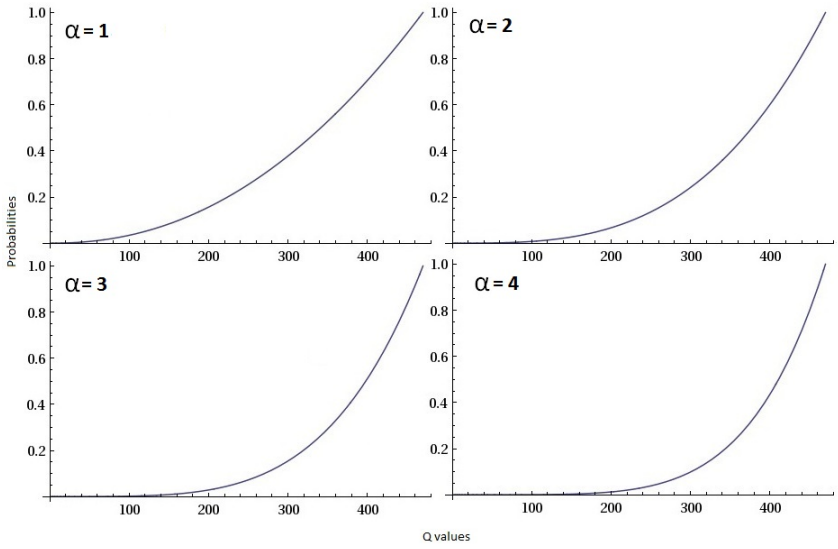


Figure 11: Probability (Y - axis) of investigating a state depends on the Q-value (X -axis) of this state. Different incline can be obtained by choosing different values of the α factor.

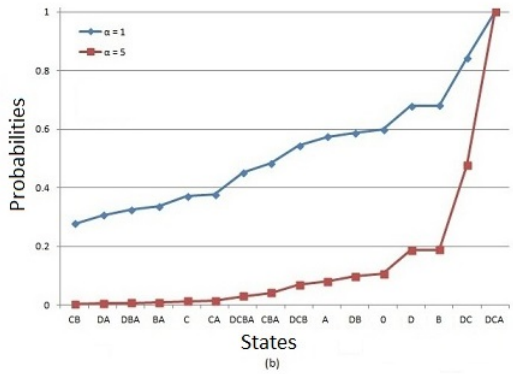


Figure 12: Each state (X - axis) is given a probability (Y - axis), depending on its Q value. Individual probabilities are summed in order to clearly illustrate the logarithmic nature of distribution

6.2.1. Steady state conditions

Depending on the α factor, the focus of the exploitation phase can be arbitrarily narrowed around the optimal system state. Figure 13 shows a

percentage of the number of episodes that our system spent in the three highest regarded states, over a course of 200 episodes long exploitation phase. As anticipated, the optimal and the nearest sub-optimal states are stronger enforced for the higher values of the α factor. For example, for $\alpha = 5$ the joint percentage rises up to 97 percent. The system resides in the optimal state 66 percent of time. For a comparison, with the "ϵ greedy" approach, we get the similar performance for $\epsilon = 0.3$ (see Figure 5). However, the joint percentage of time, spent in the three best rated states is only 67 percent. This percentage increases to 76 percent, after a threshold is used (see Figure 9), which is still significantly lower than 97 percent, achieved using the logarithmic based exploitation.

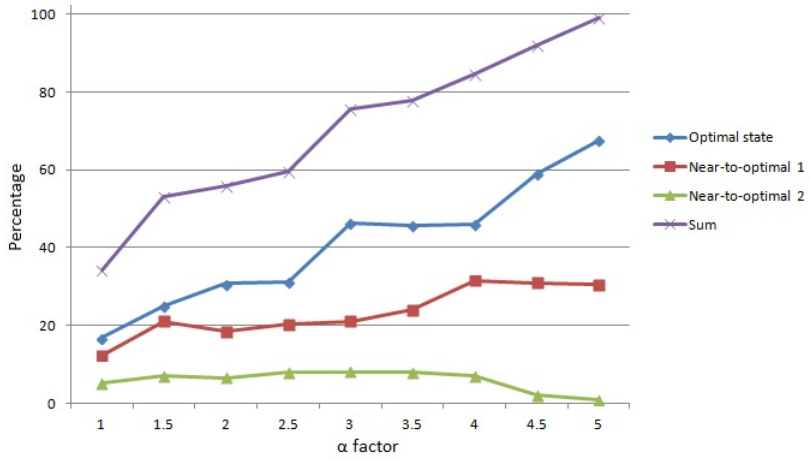


Figure 13: Percentages (Y - axis) of a number of episodes the system spent in three the highest regarded states, calculated over a course of 200 learning episodes. Results are gathered using different several α factor values (X - axis)

Regarding this aspect of performance, the newly proposed method produces significantly better results than the widely adopted "ϵ greedy" approach.

6.2.2. Reaction to a network condition disturbance

The reasoning engine's ability to detect possible state performance disturbances is tested in the following use cases:

- Case 1: Performances of the optimal state and a state performing around 80 percent of the optimal are switched

α factor	Case 1	Case 2	Case 3
1	11.75	33	100+
2	4.625	31	100+
3	8.72	48.5	100+
4	11.5	50+	100+
5	27.1	50+	100+

Table 1: The number of episodes needed for the reasoning engine to detect a network condition change, given different α factor values

- Case 2: Performances of the optimal state and a state performing around 50 percent of the optimal are switched
- Case 3: Performances of the optimal state and a state performing below 50 percent of the optimal are switched

The results are given in Table 6.2.2. In cases 1 and 2, we observe that the number of episodes starts to decrease as the α factor increases. Such a behavior is expected, since the algorithm starts to more frequently exploit states with higher Q values (especially noticeable in case 1). However, with the further increase of the α factor, the number of episodes starts to increase as well. The explanation lies in the gap between the optimal and nearest-to-optimal states, which increases as well (see Figure 12).

The most significant drawback of the algorithm is that it gives poor results in the case when a drastic disturbance occurs (Case 3) for any of the given α factor values. This use case can be compared to the one examined when using the "ϵ greedy" algorithm and it gives significantly worse results (see Figure 7 and Figure 10). Such a behavior limits the use of the algorithm to those cases where a relatively steady network performance is expected.

7. Conclusion

As the number of co-located networks increases, using a manual network configuration will easily lead to a sub-optimal network performance. To cope with this complexity, especially in dynamic environments, solutions that autonomously optimize the network settings and configurations are becoming increasingly important. To this end, this paper used a self-learning LSTDQ

based algorithm to optimize multiple co-located networks, each with a variable number of network functionalities influencing each other. The used algorithm intelligently learns the optimal network settings and detects changes in network conditions. It operates in two phases.

- Exploration phase - During which the algorithm collects information regarding all the existing state/action pairs.
- Exploitation phase - keeping the entire system in the optimal state for most of the time, but remain capable of detecting and adopting to sudden network changes.

This paper focused on improving the exploitation phase. We evaluated two possible approaches:

- ϵ greedy - exploits the information regarding each state/action pair according to the well known ϵ greedy methodology
- Logarithmic state/action probability distribution - we use a logarithmic function $P = Q^{\alpha} log Q$ to define dynamics with which state/action pairs are exploited

For both solutions, the paper analyzed in detail (i) the response time to network condition changes and (ii) the time the system spends in the (sub)optimal state. Based on these results, it is concluded that the logarithmic distribution is much better suited for most networks since it spends much more time exploring states which are likely to perform well. In addition, the distribution can be tweaked to investigate suboptimal states with different probabilities depending on the expected network dynamics. Only in networks with extreme dynamics is the ϵ greedy approach recommended.

8. Acknowledgment

This research is funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWTVlaanderen)through the IWT SymbioNets project, by iMinds through the QoCON project and by the FWO-Flanders through a FWO post-doctoral research grant for Eli De Poorter

9. References

[1] Eli De Poorter, Evy Troubleyn, Ingrid Moerman, Piet Demeester, "IDRA: A flexible system architecture for next generation wireless sensor networks", Wireless Networks, 2011, Vol. 17 (6), pp. 1423-1440

[2] Wakamiya, N.; Arakawa, S.; Murata, M.,"Self-Organization Based Network Architecture for New Generation Networks", 2009 First International Conference on Emerging Network Intelligence, pp.61-68, 11-16 Oct. 2009

[3] R. W. Thomas, L. A. DaSilva and A. B. MacKenzie, "Cognitive networks", Proc. IEEE DySPAN 2005, pp.352-60

[4] Christopher J.C.H. Watkins, Peter Dayan, Technical Note Q-Learning", Machine Learning, 8, 279-292 (1992)

[5] Richard S. Sutton, "Learning to predict by the methods of temporal differences", Machine Learning, Volume 3 Issue 1, August 1988

[6] Dimitri P. Bertsekas, " Approximate Policy Iteration: A Survey and Some New Methods", Journal of Control Theory and Applications, MIT April 2010, Vol.9, pp.310-335, Report LIDS - 2833

[7] Theodore J. Perkins, Doina Precup, " A Convergent Form of Approximate Policy Iteration", Advance in neural Information Processing Systems 15, NIPS 2002, Decembre 9-14, Vancouver, British Columbio, Canada

[8] J.M. Lanza-Gutierrez, J.A. Gomez-Pulido, M.A. Vega-Rodriguez, J.M. Sanchez-Perez, "Multi-objective evolutionary algorithms for energy-efficiency in heterogeneous wireless sensor networks", SAS 2012 : IEEE Sensors Applications Symposium, Feb 7, 2012 - Feb 9, 2012, Brescia, Italy

[9] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap and T Meyarivan, A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II, Parallel Problem Solving from Nature PPSN VI, 2000.

[10] E. Zitzler, M. Laumanns and L. Thiele, SPEA2: Improving the strength Pareto evolutionary algorithm, EUROGEN 2001.

[11] Suat zdemir Baraa A. Attea nder A. Khalil, "Multi-Objective Evolutionary Algorithm Based on Decomposition for Energy Efcient Coverage in Wireless Sensor Networks",

[12] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen, "Evolutionary algorithms for solving multi-objective problems", (2nd ed.). Berlin: Springer, (2007).

[13] E. De Poorter, B. Latre, I. Moerman and P. Demeester, "Symbiotic networks: Towards a new level of cooperation between wireless networks", Published in Special Issue of the Wireless Personal Communications Journal, Springer Netherlands, 45(4):479-495, June 2008

[14] Milos Rovcanin, Eli De Poorter, Ingrin Moerman, Piet Demeester,"A reinforcement learning based solution for cognitive network cooperation between co-located, heterogeneous wireless sensor networks", ADHoc journal

[15] M. Rovcanin, E. De Poorter, I. Moerman, P. Demeester, "An LSPI based reinforcement learning approach to enable network cooperation in cognitive wireless sensor networks", 8th Workshop on Performance Analysis and Enhancement of Wireless Networks (PAEWN-2013),4th to 7th June 2013, Barcelona, Spain

[16] M. Lagoudakis and R. Parr. "Model-free least-squares policy iteration". In Proc. of NIPS, 2001.

[17] T. G. Dietterich, and O. Langley, (2007) "Machine Learning for Cognitive Networks:Technology Assessment and Research Challenges in Cognitive Networks: Towards Self Aware Networks", John Wiley and Sons, Ltd, Chichester,UK. doi: 10.1002/9780470515143.ch5

[18] L. P. Kaelblign, M. L. Littman, A. W.Moore,"Reinforcement learning: A Survey", Journal of Artificial Intelligence Research 4 (1996) 237-285

[19] Michail G. Lagoudakis and Ronald Parr, "Least-Squares Policy Iteration, Journal of Machine Learning Research, 4, 2003, pp. 1107-1149.

[20] Lucian Busoniu, Robert Babuska, Bart De Schutter, Damien Ernst, "Reinforcement Learning and Dynamic Programming Using Function Approximators", 2010 by Taylor and Francis Group, LLC, ISBN 978-1-4398-2108-4 (Hardback)

[21] John Myles White, "Bandit Algorithms for Website Optimization", Published by O'Rilley Media, December 2012, Inc.

[22] Lieven Tytgat, Bart Jooris, Pieter De Mil, Benot Latr, Ingrid Moerman and Piet Demeester, "UGentWiLab, a real-life wireless sensor testbed with environment emulation", 6th European conference on Wireless Sensor Networks (EWSN 2009), url: https://biblio.ugent.be/publication/676545