# Evaluation of a family of reinforcement learning cross-domain optimization heuristics

Luca Di Gaspero and Tommaso Urli

DIEGM, Universit‡ degli Studi di Udine
via delle Scienze 208 – I-33100, Udine, Italy
luca.digaspero@uniud.it, tommaso.urli@uniud.it

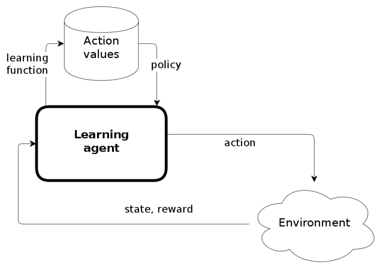
**Abstract.** In our participation to the *Cross-Domain Heuristic Search Challenge* (*CHeSC 2011*) [1] we developed an approach based on Reinforcement Learning for the automatic, on-line selection of low-level heuristics across different problem domains. We tested different memory models and learning techniques to improve the results of the algorithm. In this paper we report our design choices and a comparison of the different algorithms we developed.

## 1 Introduction

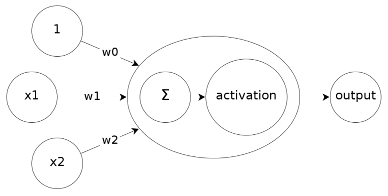
CHeSC 2011 aims at fostering the development of methods for the automatic design of heuristic search methodologies, which are applicable to multiple problem domains (see [1] for more details). The competition sits on the underlying framework of hyper-heuristics, i.e., automatic methods to generate or select heuristics for tackling hard combinatorial problems. In order to ease the implementation of hyper-heuristics and to let the participant compete on a common ground, the competition organizers released a software framework, called HyFlex [3], to be used as a basis for their algorithms. HyFlex is a Java API that provides the basic functionalities for loading problem instances, generating solutions and applying low-level heuristics. Low-level heuristics are black-boxes, and only information about their *family* is known (i.e., *ruin-recreate*, *mutation*, *cross-over* and *local search* steps). The six problem domains considered in the competition are: *MAX-SAT*, *(1D) Bin Packing*, *Permutation Flow Shop Scheduling*, *Personnel Scheduling*, *TSP* and *VRP*. We refer the reader to the CHeSC 2011 website [1] for further details.

## 2 Reinforcement Learning basics

The algorithmic alternatives that we have considered for CHeSC 2011 are all based on Reinforcement Learning (RL) [6]. In order to use RL, one needs to specify at least three components: an *environment* (whose observable features are encoded in *states*), a set of *actions* which can be pursued by the learning agent and a *reward* function which is a numeric feedback about the agent's actions.



(a) Agent-environment architecture



(b) Perceptron

The various choices for these elements, together with the action selection *policy* and the *learning function*, determine a range of different behaviors.

At each iteration, the agent selects which action to take in the current state according to its policy. The policy is a function for selecting an action based on its *value*. Together with the set of learned action values, a policy determines completely (barring stochastic effects) the agent's behavior at each decision step. The execution of an action updates the state of the agent and yields a *reward* value. The *learning function* uses this reward to update one (or mode) action values. Since the policy is usually fixed, this concretely changes the behavior of the agent for the future iterations. The state represents the agent's beliefs about the environment at a specific time; representing the state in a rich, yet complete, manner is key to the success of RL.

### 2.1 Function approximation with MLPs

When the states and actions are discrete and finite, a simple way to store action values is to keep them in a table (*tabular RL*). However, when the states or the actions are continuous or the number of states is just too large, this solution is no more viable. In these cases, the only option is to consider action values as a continuous function and to use function approximation techniques to model it. Multi-Layer Perceptrons (MLP) are a function approximation mechanism which belongs to the class of *supervised* Machine Learning algorithms. We are going to briefly revise MLPs in this section, by starting from the simpler concept of Perceptron. A (Single-Layer) Perceptron is a processing unit with a number of weighted inputs (one of which has always a value of 1) and an output. Upon activation, the Perceptron computes the weighted sum of its inputs and outputs a function of this sum. The algorithm implemented by the perceptron in Figure 1b can be summarized with the following formula:

$$h(\boldsymbol{x}) = activation(\boldsymbol{w}^T\boldsymbol{x}) \tag{1}$$

By varying *activation* one can use perceptrons to approximate different functions, however the complexity of these is very limited. MLPs are layered networks of Perceptrons in which outputs of nodes in a layer are connected to inputs of nodes in the following. Since Perceptrons are actually inspired to neurons, these networks are commonly known as Artificial Neural Networks (ANN). Layers other than input or output are called *hidden*. There are no constraints on the number of hidden layers or nodes, however it has been demonstrated [4] that a MLP with a single, large-enough, hidden layer can approximate any nonlinear function of the input. Unfortunately, there is no rule of thumb on the right number of hidden neurons, which must be worked out with parameter tuning.

### 2.2 Eligibility Traces

Eligibility Traces (ET) are a RL mechanism for *temporal credit assignment*. The idea in temporal credit assignment is that each action on the trajectory to a reward, and not just the last one, must take some credit for that reward. To accomplish this, one a chance is to keep an $e_{s,a}$ value for each visited $(s, a)$ pair. This value tells how long before the pair was visited and is updated as follows

$$e_{s,a} = \begin{cases} 1 & \text{if state is } s \text{ and action is } a \\ \lambda e_{s,a} & \text{otherwise} \end{cases} \tag{2}$$

where $\lambda$ is a decay factor in $[0, 1)$. In RL, $e_{s,a}$ is called *eligibility trace* of $(s, a)$ and is used by the learning function to weight the update to $(s, a)$. Intuitively, recently visited $(s, a)$ pairs are more likely to be responsible for a reward with respect to older ones, and should benefit (or suffer more) from the last obtained reward. In practice, one can implement ETs efficiently by keeping a queue of the last $\lceil \log(\mathit{threshold}) / \log(\lambda) \rceil$ visited pairs where *threshold* is the value of $e$ under which an update is considered neglectable. Then $e_{s,a}$ is computed as $\lambda^{\mathit{position}}$ where *position* is the pair's position in the queue.

## 3 Reinforcement Learning for heuristic selection

In order to describe our RL hyper-heuristics we must identify the following elements: (i) the *environment states*, (ii) the set of *actions*, (iii) the *reward*, (iv) the *policy* and (v) the *learning function*.

**Environment:** HyFlex is designed to support the construction of cross-domain hyper-heuristics. For this reason, all the information about a solution (except its cost) domain is hidden to the user. This makes things difficult because, in principle, RL states must be Markov (i.e. enough informative to allow choosing the right action). After attempting some variants, we resorted to a state representation which tries to capture the concept of reward trend. In particular, when a reward is obtained, the new state is computed as $s_{i+1} = \lfloor (s_i + \mathit{reactivity} * (r_i - s_i)) \rfloor$, where $r_i$ is a normalized cross-domain reward measure and *reactivity* defines the attitude of the agent to switch state.

**Actions:** We defined a possible *action a* in a given state as the choice of the heuristic family to be used, plus an intensity (or depth of search) value in

Table 1: Common hyper-heuristic parameters

| *parameter name* | *values* | *explanation* |
|---|---|---|
| agents | 3, 4, 5, 6, 8, 10 | number of concurrent agents |
| crossoverWith | bestAgent, incumbentOptimum | secondary solution for cross-over |
| reactivity | 0.05, 0.1, 0.25, 0.5, 0.9 | readiness to change state |
| learningRate | 0.1, 0.2, 0.3 | readiness to update action values |
| epsilon | 0.01, 0.05, 0.1 | probability to pick random actions |

the quantized set of values $0.2, 0.4, 0.6, 0.8, 1.0$. Once the family has been determined, a random heuristic belonging to that family is chosen and applied to the current solution with the specified intensity (or depth of search).

**Reward:** The reward $r$ is computed as the solution's $\Delta_{cost}$ before and after action application. Variants which also take into account the time spent applying an action have been tried, but with poor results.

**Policy:** The $\epsilon$-greedy policy (which chooses $\arg\max_a \pi_{s,a}$ with probability $1-\epsilon$ and a random action otherwise) proved to perform better than a number of alternatives and is currently our policy of choice. A note on the *move acceptance criterion*: in our approach we decided to always trust the policy hence we always apply the action chosen even if it deteriorates the solution.

**Learning function** The learning function is based on a very simple update $\pi(s,a)_{k+1} = \pi(s,a)_k + learningRate * (r_k - \pi(s,a)_k)$ which always moves the estimated reward for a $(s,a)$ pair towards the last reward. The discount factor *learningRate* is needed to tackle non-stationarity (i.e. updates are constant, the policy never converges).

## 4 Experimental analysis

In order to tune the variants' parameters and to understand their relationships we carried out an experimental analysis based on the tools commonly employed in the statistical analysis of algorithms. To collect the required data we ran all configurations on three different Intel machines equipped with Quad Core processors (resp. at 2.40, 2.40 and 3.00 GHz) and running Ubuntu 11.04. Differences were leveled through a benchmarking tool provided by CHeSC organizers.

We compare three hyper-heuristics inspired to the RL variants in section 2: tabular reinforcement learning (RLHyperHeuristic), tabular reinforcement learning with ETs (RLHyperHeuristic-ET) and reinforcement learning with MLP function approximation (RLHyperHeuristic-MLP). Although each approach has its own parameters, some of them (Table 1) are common to all hyper-heuristics.

RLHyperHeuristic-MLP requires a number of extra parameters (see Table 2a) related to MLPs. The parameters *hiddenLayers* and *hiddenNeurons* determine the complexity of the function that the MLP is able to approximate. *inputScale* is a parameter to control input normalization. Similarly RLHyperHeuristic-ET introduces two parameters (see Table 2b): *threshold* and *traceDecay* ($\lambda$), which are used to compute the length of the eligibility queue.

Table 2: Specific parameters for RLHyperHeuristic variants

(a) RLHyperHeuristic-MLP

| *parameter name* | *values* |
|---|---|
| hiddenLayers | 1, 2 |
| hiddenNeurons | 20, 30, 40 |
| inputScale | 1, 3 |

(b) RLHyperHeuristic-ET

| *parameter name* | *values* |
|---|---|
| threshold | 0.01 |
| traceDecay | 0.5, 0.9 |

Since the evaluation has to be performed across different domains and on instances with different scales of cost functions we decided to consider as the response variable of our statistical tests the normalized cost function value. That is, the cost value $y$ is transformed by means of the following transformation, which aggregates the results on the same problem instance $\pi$:

$$e(y, \pi) = \frac{y(\pi) - y^*(\pi)}{y_*(\pi) - y^*(\pi)} \tag{3}$$

where $y^*(\pi)$ and $y_*(\pi)$ denote the best known value and the worst known value of cost on instance $\pi$. This information has been computed by integrating the data gathered by our experiments with the information made public by CHeSC organizers. In all the following analyses we employ the R system [5].

*Parameter influence.* The first analysis aims at clarifying the influence of parameters on the outcome of the algorithms, in order to fix some of the parameters to *reasonable* values and to perform further tuning of the relevant ones. For this purpose we perform an *analysis of variance* on a comprehensive dataset including all configurations run throughout all the problem domains. Each variant has been run on each instance for 5 repetitions. We perform separate analysis for each variant and we set the significance level of the tests to 0.95.

**RLHyperHeuristic:** The most relevant parameters are the selection of the cross-over solution and the number of agents, but there seems to be no detectable interaction among them. As for this variant, $\epsilon$ is also significant.

**RLHyperHeuristic-ET:** The relevant parameter is the *traceDecay*, apart of the selection of the agent for the crossover that was relevant also in the previous variant. The number of agents doesn't seem to be relevant.

**RLHyperHeuristic-MLP:** Apart *crossOver*, the *inputScale* and *hiddenNeurons* are relevant in explaining the different performances of the algorithm. We do not found any significant second-order interaction among parameters.

*Parameter tuning.* For tuning the parameters identified in the previous analysis we employed the $F$-Race technique [2]. As for the selection of the best candidates, we took the ones that had the lowest median value of the normalized cost function across all instances and all domains. The setting of the parameters for the three different variants of the algorithm are reported in Table 3.

Table 3: **CW** = crossoverWith, **LR** = learningRate, **R** = reactivity, **TH** = treshold, **TD** = traceDecay, **IS** = inputScale, **HL** = hiddenLayers and **HN** = hiddenNeurons

| *variant* | *agents* | *CW* | $\epsilon$ | *LR* | *R* | *TH* | *TD* | *IS* | *HL* | *HN* |
|---|---|---|---|---|---|---|---|---|---|---|
| RL | 5 | incumbentOptimum | 0.05 | 0.2 | 0.5 | | | | | |
| RL-ET | 4 | incumbentOptimum | 0.1 | 0.1 | 0.1 | 0.01 | 0.5 | | | |
| RL-MLP | 4 | incumbentOptimum | 0.05 | 0.1 | 0.5 | | | 1 | 1 | 20 |

*Comparison with the other participants.* We have compared our variants with the other participants in the CHeSC competition by using the median value of the normalized cost function for ranking. Overall the variant using function approximation improves over our original algorithm (13th place, against 16th), while the one which uses eligibility traces doesn't. A proper tuning of the algorithm we sent to CHeSC has determined a relevant improvement but overall we are still far from the first positions. This is likely to be caused by the state representation, which seems to be insufficiently informative.

## 5 Conclusions and Future Work

This work is part of our investigation about the use of Machine Learning techniques for driving combinatorial optimization algorithms. In our opinion the results are interesting given the fact that there is no move acceptance criteria and the whole control is in the hands of a learning algorithm. However the intrinsic limitations imposed by the competition are too tight to allow a proper RL integration. For this reason, we are currently investigating these approaches outside the HyFlex framework.

## References

[1] ASAP Research Group, Nottingham: CHeSC: the cross-domain heuristic search challenge (2011)

[2] Birattari, M., Stutzle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. vol. 2, pp. 11–18 (2002)

[3] Burke, E., Curtois, T., Hyde, M., Ochoa, G., Vazquez-Rodriguez, J.: HyFlex: A Benchmark Framework for Cross-domain Heuristic Search. Arxiv preprint arXiv:1107.5462 pp. 1–27 (2011), `http://arxiv.org/abs/1107.5462`

[4] Hornik Maxwell, K., White, H.: Multilayer feedforward networks are universal approximators. Neural networks 2(5), 359–366 (1989)

[5] R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2011), `http://www.R-project.org`, ISBN 3-900051-07-0

[6] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning). The MIT Press (1998)