Solving Optimization Problems with Reinforcement Learning
-
Report

Timothé Boulet, Pierre Prévot-Helloco, Alexandre Selvestrel

Centrale Supélec

Abstract. Combinatorial optimization problems are widely encountered in various domains, ranging from logistics [14] and resource allocation [2] to finance and engineering. Traditional methods such as linear programming [11] or metaheuristic [12] provide efficient solutions for these problems. However, they face limitations for non-linear constrained problems and cannot tackle problems where the objective function is nonlinear and highly complex and stochastic. [6]
In this project, guided by the intuition that Reinforcement Learning (RL) is particularly adapted to treat complex sequential decision problems with combinatorial objective functions, we considered using RL as a metaheuristic to solve such optimization problems. We focus on three classical problems in combinatorial optimization: the Knapsack Problem, the Facility Location Problem (FLP) and the Bin Packing Problem. To realize a proof of concepts of RL algorithms on optimization problems, we chose to model those problems as tabular environments, where one environment corresponds to one particular instance of the problem. We implemented many RL algorithms in their tabular versions and compared their performances on the three problems. Furthermore, we show the importance of the densification of the reward allowed by the MDP formalism. Finally, we extend the approach to non-constant-instance setting with Deep Reinforcement Learning, where the agent would learn a policy on a problem and transfer it to new instances of the same problem without retraining. [5 pt]

1 Introduction

Today, concrete optimization problems grouped under the name Operation Research (OR) have become of utmost importance in many domains, such as logistics [14], finance [4], or engineering. Those problems are usually solved using traditional OR methods such as linear programming [11] [12], which provide efficient solutions for these problems. However, those methods are useless when the objective function or the constraint becomes non-linear, which can happen in many real-world problems, or when the dimension of the linear formulation would be too high, or worse, when the objective function becomes stochastic and highly complex [5]. This gives incentives to tackle those problems from a partially black-box perspective, where the objective function f is unknown and can only be evaluated at some points, and where metaheuristics are used to find the optimal solution among the feasible ones, i.e. those that respect a certain number of constraints $C(x)$.

$$\max_x f(x) \quad \text{s.t.} \quad C(x) \tag{1}$$

We consider in particular the case of combinatorial optimization problems, where the search space is discrete. In this context, we are faced with several problems in choosing our metaheuristic. First, the initial (unconstrained) search space is often very large, altough the feasible space is much smaller. This makes any naive search algorithm non-scalable and force the use of complicated problem-aware neighborhood functions or heuristics. Second, the objective function is only black-box and nondifferentiable, which prevents the use of gradient-based methods. Third, classical metaheuristics such as Genetic Algorithms [8] or Simulated Annealing [3] are often not adapted to the sequential nature of the problem, where the solution is built step by step, and possibly of a non-fixed size. Finally, the time-intensive nature of the optimization process on a new problem's instance often prevent the use of those methods

online, where the objective function is evaluated in real-time : the optimization process performed on one instance of a problem cannot be even partially transferred to another instance of the same problem.

2 Reinforcement Learning for Optimization Problems

In order to answer these limitations mentioned above, we propose to modelize the optimization problems as Markov Decision Processes (MDPs), where the agent sequentially constructs the solution x by selecting actions a in a discrete action space $A(s)$. In this framework, the state s of the MDP is the current partial (in construction) solution, the action a is the choice of the next element to add to the solution, and the reward r is the evaluation of the objective function $f(x)$ at the end of the episode. For some optimization problem, it is possible to not give the reward at the end of the episode, but to give it to the agent in a more dense manner, for example, if the objective function involves a sum of term that can be defined at some intermediate steps of the solution construction. The episode ends when the solution is complete. Each episode is an instance of a problem, and the agent can learn a policy on instances of the same problem and transfer it to new instances of the same problem without retraining.

This framework presents the following advantages in comparison to other metaheuristics :

- By building the solution sequentially, RL scales extremely well with the size of the problem
- It can tackle non-fixed size solutions, and stochastic objective functions
- Constraints are easy to implement using Action Masking, which prevents the agent from selecting actions that would lead to infeasible solutions. This is done either by removing the action from $A(s)$ in the tabular case, or by adding $-\infty$ to the Q-value or the log-probability of the action in the Deep RL case.
- The agent can learn a policy on a problem in order to solve new instances of the same problem later without retraining

The last point is the most important one. To our knowledge, RL is the only metaheuristic that is able to learn reusable policies on non-differentiable, black-box, possibly stochastic, and complex objective functions.

We first wanted to realize a proof of concepts of RL algorithms on optimization problems, so we chose to model those problems as tabular environments, where one environment corresponds to one particular instance of the problem. This is a constant-instance setting, which prevents any transfer of knowledge between instances of the same problem. Consequently, each environment is a particular instance of the problem, and the agent has to learn a policy. We assume this could be extended to the case of nonconstant-instance episodes, with deep reinforcement learning, where the agent would learn a policy on a problem and transfer it to new instances of the same problem, but this is out of the scope of this project. In Section 7, we extend the approach to non-constant-instance setting with deep-reinforcement learning.

3 Optimization Problems as MDPs

We focus on three OR problems, which are classical in the literature of combinatorial optimization: the knapsack problem [13], the Facility Location Problem (FLP) [10] and the Bin Packing problem [1]. Here, we present how we modelized those problems as MDPs. Because we are in the tabular setting, the environment in itself already contains the information of the instance, so the only information that we need to give the agent is the current state of the solution. In a non-constant-instance setting with Deep RL however, the state should also contain the information of the instance.

3.1 Knapsack Problem

The Knapsack Problem is a classic optimization problem [9] [7] where the agent has to select a subset of items to put in a knapsack, in order to maximize the total value of the items in the knapsack, while respecting the capacity

constraint of the knapsack. Formally, the problem can be defined as follows : you have a set of n items, each item i has a value v_i and a weight w_i, and the knapsack has a capacity W. The objective is :

$$\max_x \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

We formalized the problem as a MDP as follows :

- The state s_t at timestep t is the information of the items that are already in the knapsack : $(1_{i \in K_t})_{i=1}^{n}$ where K_t is the set of items in the knapsack
- The action space $A(s_t)$ is the set of non-added items that can be added to the knapsack without exceeding the capacity
- The reward r_t is the value of the item added at timestep t
- The episode ends when their is no more item that can be added to the knapsack.

This gives an objective function G_0 that is the sum of the values of the items in the knapsack (if $\gamma = 1$).

$$G_0 = \sum_{t=0}^{T-1} r_t = \sum_{i \in K_T} v_i \tag{3}$$

We also treated this problem with deep RL for non-constant-instance setting. In that case, we gave the algorithm the knowledge of the weights (ω_i) and of the values (v_i) (because they change at each reset of the environment). However, during all the runs, we kept constant the number of objects and the capacity W. We also used the same parameters for the random sampling to create the values and the weights for each episode.

3.2 Facility Location Problem

The Facility Location Problem is a classic optimization problem [10] in which the agent has to select a subset of facilities to open in a set of potential facility sites, to minimize the total cost of transportation between the facilities and the customers. Formally, the problem can be defined as follows : you have a set of $n_{\text{facilities}}$ facilities, a set of $n_{\text{facility sites}}$ potential facility sites and a set of $n_{\text{customer sites}}$ customer sites. In our variant, there is no cost in opening a facility, and the cost to minimize is the sum of L_2 distances between the customer sites and their nearest facility. You have to select a subset of facility sites to open, and to assign each customer site to its nearest facility site.

We formalized the problem as a MDP as follows :

- The state s_t at timestep t is the information of the facility sites that are already opened : $s_t = (1_{i \in F_t})_{i=1}^{n_{\text{facility sites}}}$ where F_t is the set of opened facility sites at timestep t
- The action space $A(s_t)$ is the set of non-opened facility sites that can be opened : $A(s_t) = \{i \in \{1, \ldots, n_{\text{facility sites}}\} \setminus F_t\}$
- The reward is the loss of cost between the solution at timestep $t-1$ and the solution at timestep t : $r_t = \text{cost}(s_{t-1}) - \text{cost}(s_t)$. The cost is the sum of the L_2 distances between the customer sites and their nearest facility site. For convenience, $\text{cost}(s_{-1})$ was defined as the cost of a very bad solution with only one random (or the worst) facility opened. The reward represents therefore "by how much I have improved the solution by placing a facility at this site".
- The episode ends at timestep $T = n_{\text{facility sites}}$ when all facilities have been placed.

This gives an objective function that represents the improvement in cost between the initial (dummy) solution and the final solution :

$$G_0 = \sum_{t=0}^{T-1} r_t = \text{cost}(s_{-1}) - \text{cost}(s_T) \tag{4}$$

We added a method `render` for the environment that plots the current solution build by the RL agent (if requested). This visualization is represented on figure 1 for FLP Medium.

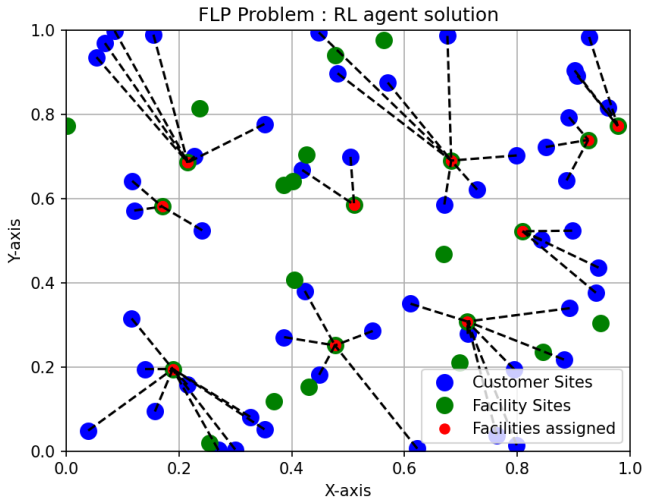

Fig. 1: Facility Location Problem

3.3 Bin Packing Problem

The bin packing problem is a optimization problem [1] where a set of items with varying dimensions must be packed into a minimum number of bins, each with a fixed capacity, and the agent aims to minimize the number of bins used. Formally, the problem can be described as follows: given a set of items with sizes $s_1, s_2, ..., s_{n_objects}$ and a fixed bin capacity C, the objective is to find the minimum number of bins necessary to pack all items.

In our framework, we represent the problem as follows:

- The state s_t at timestep t is the list of bins with remaining size. It contains information on the items that are already in the bins: $(C - c_i^t)_{i=1}^{n_t}$, where n_t is the number of bins at time t. Initially, the state is empty.
- The action space $A(s_t)$ depends on t: At step t, we always choose to place the t^{th}-object. This is based on the trick given in the subsequent paragraph. Thus, at time t, all possible actions are to place the object in all bins with remaining capacity greater than the object's size: $(1_{s_t \leq C - c_i^t})_{i=1}^{n_t}$. We also add the action $(n_t + 1)$ whose output is to create a new bin.
- The reward r_t is null unless the action of creating a new bin is taken, and then the reward is -1.
- The episode ends at time $T = N_{objects}$.

Our environment relies on some computational tricks:

- The order of objects placed at the end is not crucial for a particular outcome of an episode if we don't assign numbers to the bins. Therefore, we can establish the order of objects to be placed at each step during the

initialization of the environment, and thus at every step t, we will place the same t^{th}object. In this way, the agent can inherently determine the size of the object at each step, which reduces the number of combinations.
- When considering the bins, it is preferable to arrange the remaining capacities of each bin in decreasing order after every action is taken. This practice reduces the number of potential states.
- We must handle digit representation carefully, as approximation errors (e.g., 1.9999 instead of 2) can result in two distinct states.

Furthermore, we initialize the environment by creating bins with full capacity filled with random-size objects. Thus, we can have the optimal reward by construction, equal to $n_{\text{optimal bins}}$. For the worst reward, the theoretical worst reward is equal to the number of generated objects. In our case, because we have a better normalized curve, we chose $-2n_{\text{optimal bins}}$

3.4 Environments parameters and instance size

We performed our benchmark on different instance sizes for each problem, controlled by their respective parameters. Knapsack and FLP have 3 different instance sizes, while Bin Packing has 2 different instance sizes. The parameters of the instances are detailed in table 1.

Problem	Small	Medium	Big
Knapsack	10 items, 10 capacity, 2 average weight	30 items, 30 capacity, 2 average weight	100 items, 100 capacity, 2 average weight
FLP	4 facilities, 10 facility sites, 20 customer sites	10 facilities, 25 facility sites, 50 customer sites	40 facilities, 100 facility sites, 200 customer sites
Bin Packing	4 optimal bins of 5 objects each, capacities of 10		10 optimal bins of 10 objects each, capacities of 20

Table 1: Instance sizes and parameters

4 Implemented RL algorithms

We implemented the following RL algorithms in their tabular versions :

- Random
- Q-Learning
- Double Q-Learning
- SARSA
- n-step SARSA; for which we chose $n = 4$
- SARSA(λ); with replacing traces (traces are reset to 0 at each episode)
- Monte Carlo
- REINFORCE

These algorithms are all model-free environment-agnostic algorithms, which means that they can be applied to any MDP without any modification. They were implemented in a modular way by subclassing a generic agent class with the methods `act` and `update`.

4.1 Exploration Policies

We tried different exploration policies :

ϵ-greedy The ϵ-greedy policy is the most common exploration policy in RL. The agent selects the action that maximizes the Q-value with probability $1 - \epsilon$ and a random action with probability ϵ.

$$\pi_{\epsilon\text{-greedy}}(a|s) = \begin{cases} 1 - \epsilon & \text{if } a = \arg\max_{a'} Q(s, a') \\ \epsilon/|A(s)| & \text{otherwise} \end{cases} \tag{5}$$

We chose to exponentially decay the parameter ϵ from 1 to 0.05.

Boltzmann The Boltzmann policy is a stochastic policy that selects the action according to a Boltzmann distribution of the Q values from a softmax function. The softmax is controlled by a temperature parameter τ that decays during training. The infinite temperature gives a uniform distribution over the actions and the zero temperature gives a greedy policy. We chose to decay the temperature parameter exponentially from 10 to 0.001.

$$\pi_{\text{Boltzmann}}(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}} \tag{6}$$

UCB The UCB policy is a deterministic policy that selects the action that maximizes the UCB value, which is a trade-off between an exploitation term (the Q value) and an exploration term.

$$\pi_{\text{UCB}}(a|s) = \arg\max_a Q(s, a) + c * \sqrt{\frac{\log(t)}{N(s, a)}} \tag{7}$$

The exploration term is $c * \sqrt{\frac{\log(t)}{N(s,a)}}$ where $N(s, a)$ is the number of times the action a has been selected in the state s, t is the total number of steps since the beginning of the training, and c is an exploration parameter that controls the exploration-exploitation trade-off. In our case, we chose $c = 2.0$.

4.2 Hyperparameters

We used the same hyperparameters for all the algorithms, which are the following :

Hyperparameter	Value
Learning rate	0.1
Discount factor γ	0.99
Initial exploration rate ϵ	1
Final decay rate ϵ	0.05
Initial Boltzmann temperature	10
Final Boltzmann temperature	0.001
UCB exploration parameter	2.0

Table 2: Hyperparameters

We used a hash table to store the Q-values for value-based algorithms and the policy for policy-based algorithms (REINFORCE). The Q values were initialized randomly following a normal distribution $\mathcal{N}(0, 1)$.

4.3 Scheduling

The exploratory parameters (ϵ and the Boltzmann temperature) were decayed exponentially at each episode between their initial and final values. Their final values were reached at approximately 50% of the total number of episodes. The decay of the ϵ parameter is shown in figure 2.

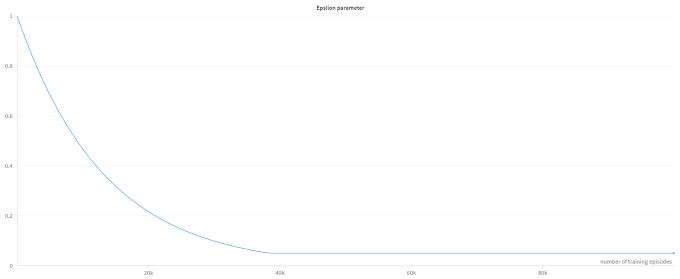

Fig. 2: Decay of the ϵ parameter

This divided the training in an exploration phase and an exploitation phase, which allows the agent to explore the search space at the beginning of the training, then to exploit the knowledge that it has learned in the exploration phase for converging towards a strong local optimum.

5 Experimental Setup

We described here the experimental setup that we used to compare the performances of the different algorithms on the three problems.

5.1 Experimental Protocol

For each environment, we trained our algorithm on three instances of different sizes (small, medium, and big), and we performed 10 runs for each algorithm and each instance size. The seeds used for the benchmark were different between the runs, but the same between the algorithms and the instances, so that the results are comparable.

We trained our algorithms for 100k episodes, evaluating them every 1000 episodes for a duration of only 1 evaluation episode, which made evaluation curves of 100 points. In evaluation mode, the agent does not explore anymore, so the different exploration policies are replaced by a greedy policy, except in the case of REINFORCE which is not a value-based algorithm, so the policy is always stochastic.

The results were averaged over the 10 runs and we calculated the mean and standard error at every evaluation and training step. The results were then plotted to compare the performance of the different algorithms using the WandB platform.

5.2 Normalized Performance Metric

Because the optimal reward is dependent on the instance's parameter, which is different at each run, the reward is not directly comparable between the runs. We chose rather to use the Normalized Performance Metric (NPM) :

$$NPM = \frac{G_0 - G_{\text{bad return}}}{G_{\text{optimal return}} - G_{\text{bad return}}} \tag{8}$$

G_0 is the undiscounted return obtained by the agent during the episode.

$G_{\text{bad return}}$ is a return that was considered as bad depending on the problem. It can be the return of a bad (e.g. random) policy, the worst possible return that can be obtained on the problem, or simply a lower bound on the optimal solution (although in this case the NPM can be negative if the agent performs worse than the lower bound). The NPM is 0 when the agent performs as bad as the bad policy.

$G_{\text{optimal return}}$ is the return of the optimal solution on the problem. This was computed using a linear solver when the problem could be formalized as a linear problem (Knapsack, FLP). When this was impossible (bin-packing), we defined this optimal return by construction when initializing the environment (see Section 3, Bin-Packing Problem). The NPM is 1 when the agent performs as good as the optimal solution.

This give a normalized metric between 0 and 1 that is comparable between the runs, and gives a criteria for whether an algorithm can solve an optimization problem or not.

The way we chose $G_{\text{bad return}}$ is detailed in table 3 and the way we computed $G_{\text{optimal return}}$ is detailed in table 4.

Problem	$G_{\text{bad return}}$	Meaning	Type of "bad return"
Knapsack	0	No item in the knapsack	Lower bound on any solution
FLP	0	No improvement compared to choosing one facility	Lower bound on any solution
Packing	$-2n_{\text{optimal bins}}$	Twice the number of bins of the optimal solution	Very bad feasible solution

Table 3: Bad returns definitions

Problem	$G_{\text{optimal return}}$
Knapsack	LP optimal solution
FLP	LP optimal solution
Packing	$-n_{\text{optimal bins}}$

Table 4: Optimal returns definitions

6 Results

We present here the results of the experiments that we conducted on the three problems. We first show the benchmark of the different algorithms on the three problems and discuss the results. We then present the influence of other factors on the performances of the algorithms, such as the size of the instance, the reward sparsity, or the choice of the explorative policy,

6.1 Benchmark

Here we present the benchmark of the different algorithms for the three problems. We plotted the median and standard error of the NPM through the training at each evaluation step for each algorithm and each problem's instance size. We also summarize the final NPM obtained by each algorithm on each problem's instance size.

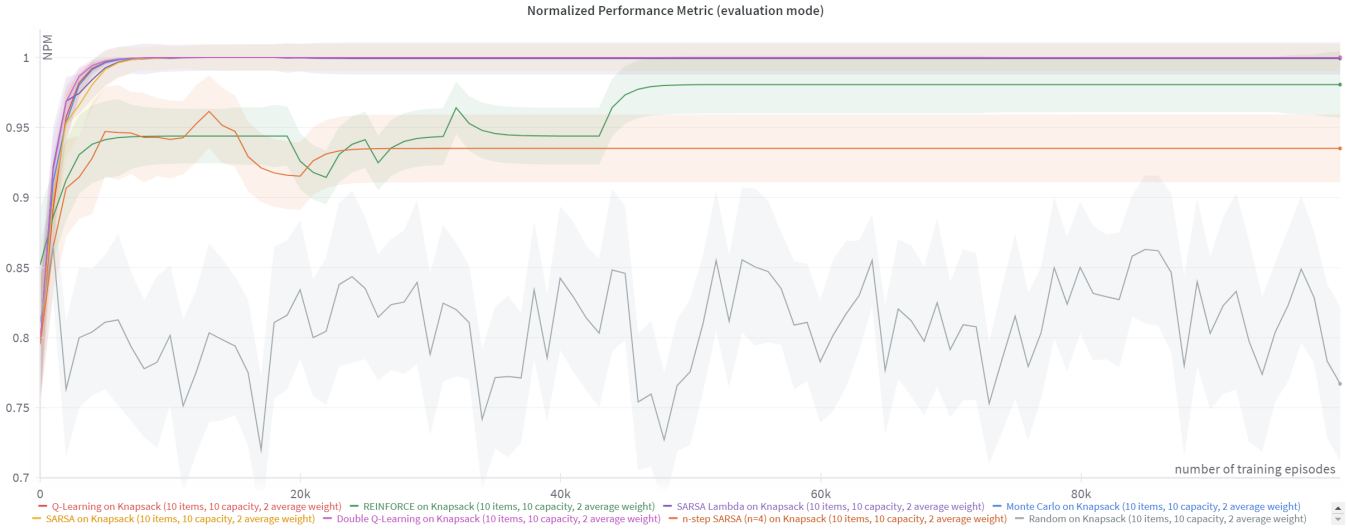

Fig. 3: Evaluation performance on Knapsack Small (10 items, 10 capacity, 2 average weight)

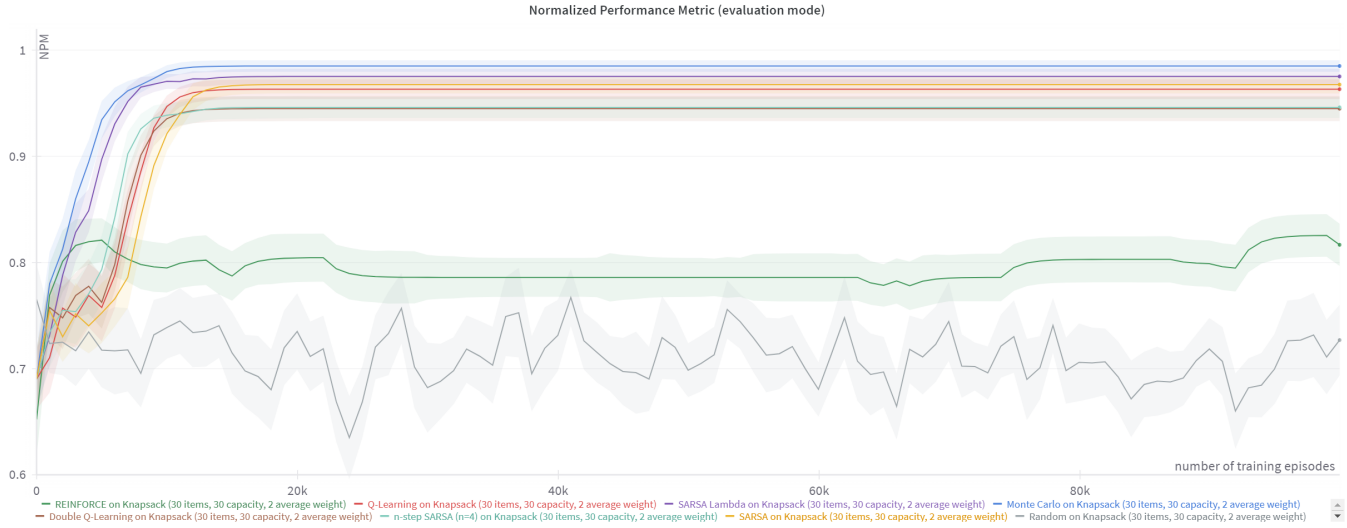

Fig. 4: Evaluation performance on Knapsack Medium (30 items, 30 capacity, 2 average weight)

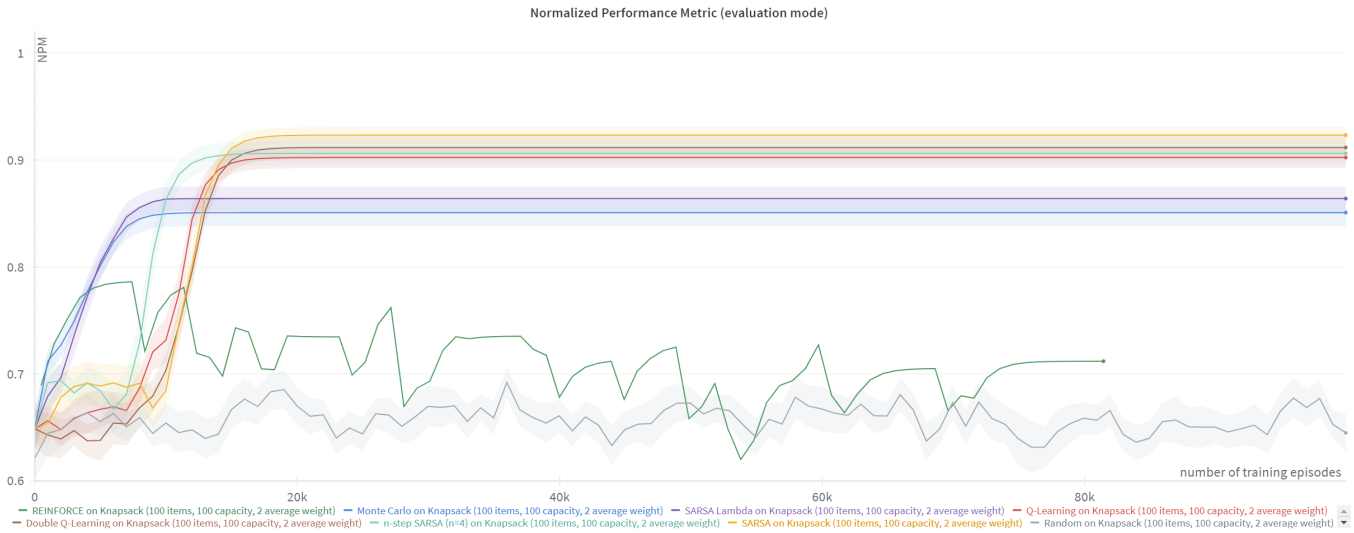

Fig. 5: Evaluation performance on Knapsack Big (100 items, 100 capacity, 2 average weight)

Algorithm	Small	Medium	Big
Random	80.00	72.69	64.50
Q-Learning	**100.0**	96.32	90.24
Double Q-Learning	**100.0**	94.49	91.18
SARSA	**100.0**	96.76	**92.33**
n-step SARSA	93.51	94.61	90.63
SARSA(λ)	99.93	97.52	86.4
Monte Carlo	**100.0**	**98.51**	85.09
REINFORCE	98.07	81.66	-

Table 5: Final NPM (in %) on Knapsack

Knapsack Problem On the three instances, all the algorithms managed to learn a better solution than random (grey). REINFORCE (green) systematically showed inferior performance, finding an only slightly better solution than random on the medium and big instance. TODO : interpret The value-based algorithms (Q-Learning, Double Q-Learning, SARSA, n-step SARSA, SARSA(λ), Monte Carlo) all found significantly better solutions than random. All of them (except surprisingly n-step SARSA (orange)) found the optimal solution on the small instance. In the medium instance, Monte Carlo (blue) found the best solution. In the big instance, SARSA (yellow) found the best solution.

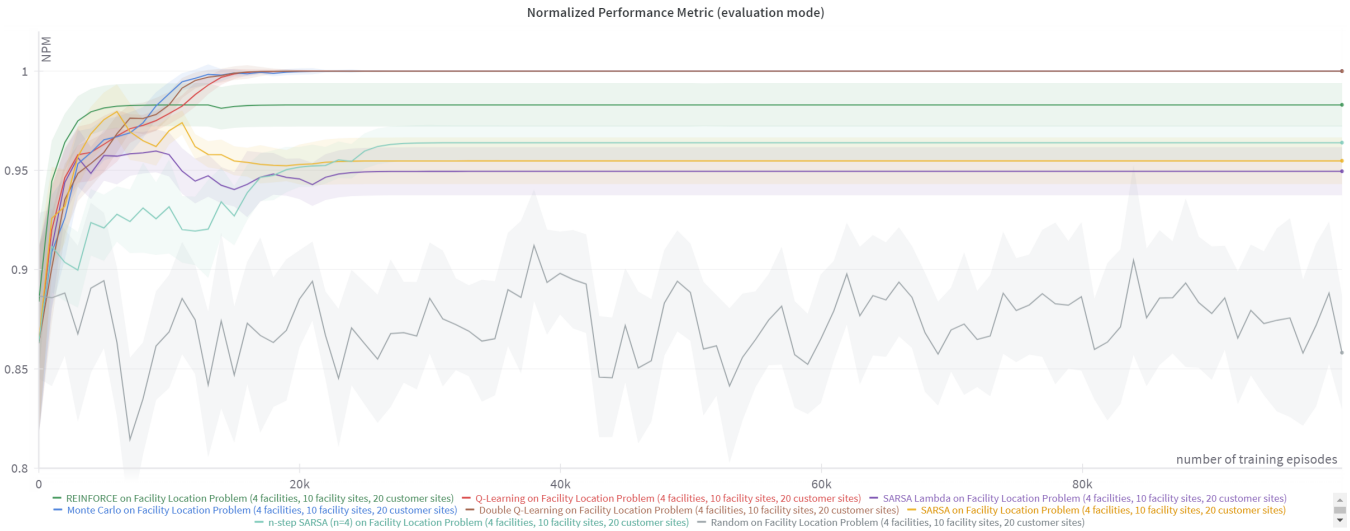

Fig. 6: Evaluation performance on FLP Small (4 facilities, 10 facility sites, 20 customer sites)

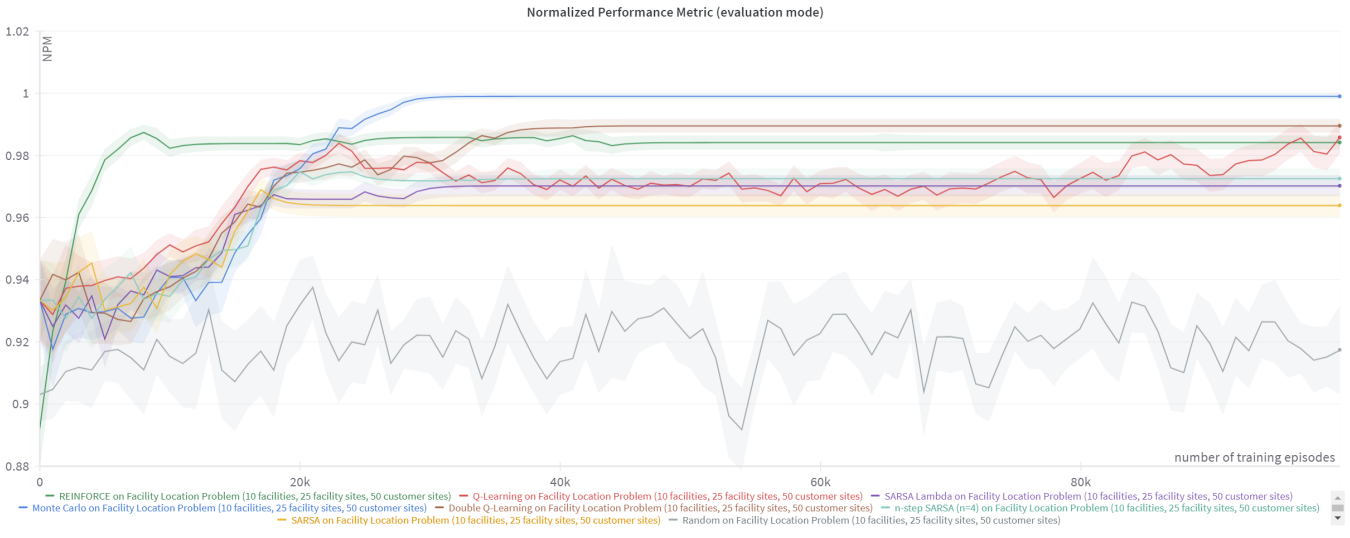

Fig. 7: Evaluation performance on FLP Medium (10 facilities, 25 facility sites, 50 customer sites)

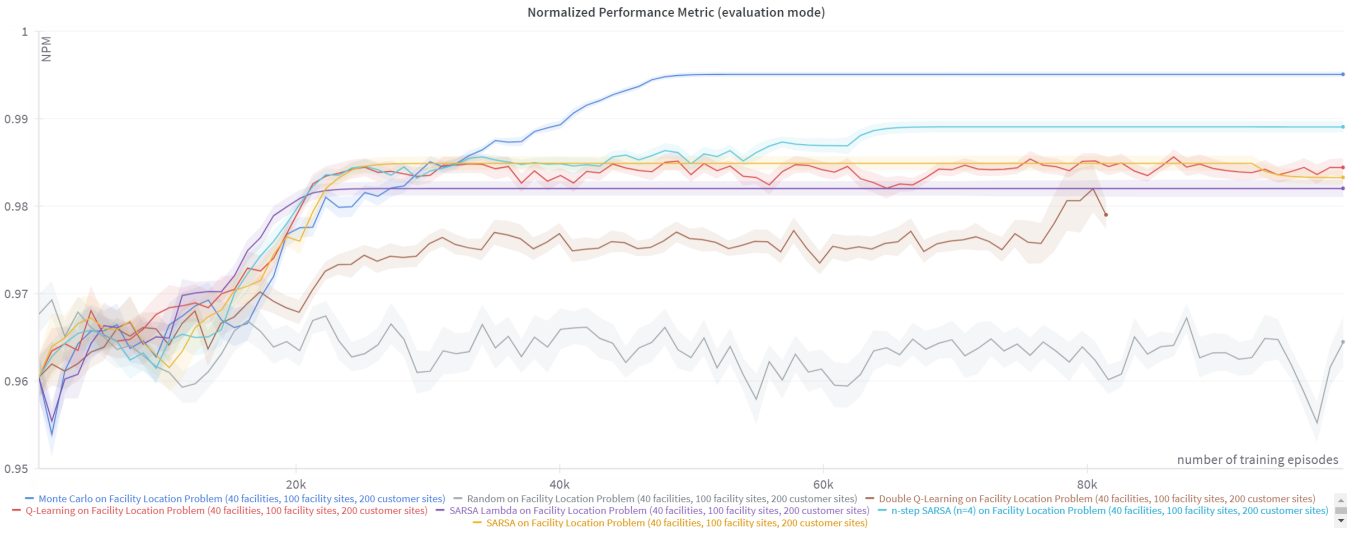

Fig. 8: Evaluation performance on FLP Big (40 facilities, 100 facility sites, 200 customer sites)

Algorithm	Small	Medium	Big
Random	85.82	91.75	96.45
Q-Learning	**100.0**	98.58	98.44
Double Q-Learning	**100.0**	98.95	98.20
SARSA	95.48	96.39	98.33
n-step SARSA	96.40	97.25	98.91
SARSA(λ)	94.96	97.02	98.20
Monte Carlo	**100.0**	**99.90**	**99.50**
REINFORCE	98.30	98.42	-

Table 6: Final NPM (in %) on FLP

Facility Location Problem On the three instances, all the algorithms managed to learn a better solution than random (grey), including REINFORCE this time. In the small instance, Q-Learning (red), Double Q-Learning (brown) and Monte Carlo (blue) found the best solution. In the medium instance, Monte Carlo performed the best, being the only algorithm to find the optimal solution, followed by Double Q-Learning. In the big instance, no algorithms found the optimal solution. Monte Carlo performed the best, and Double Q-Learning the worst this time.

Generally speaking, Monte Carlo seemed to work particularly well on the FLP problem. Double Q-Learning also performed well, but less consistently than Monte Carlo.

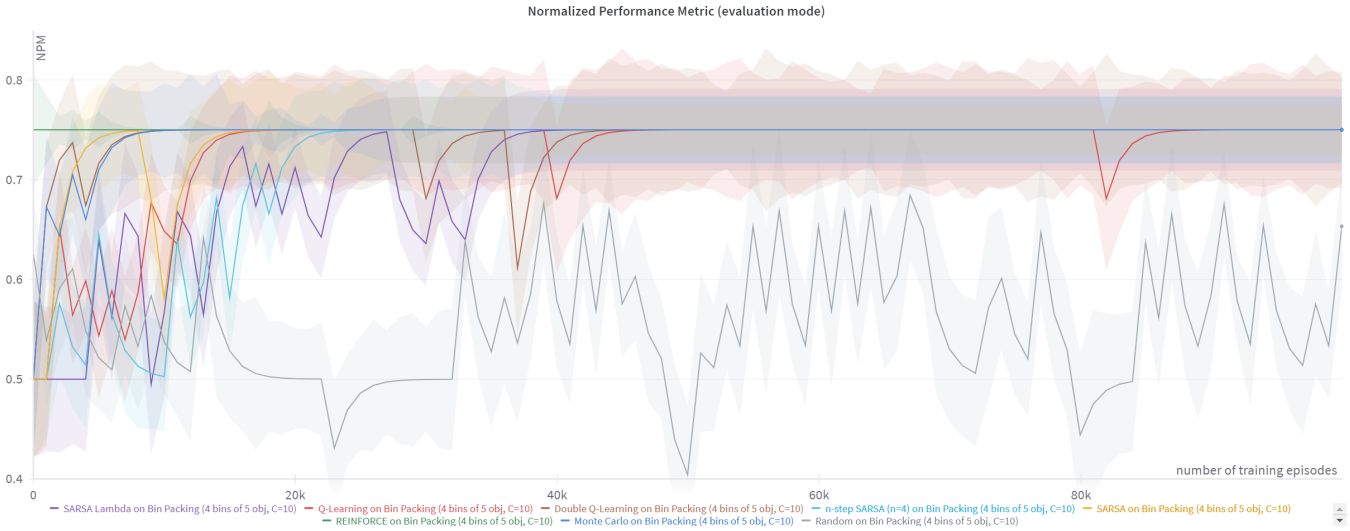

Fig. 9: Evaluation performance on Bin Packing Small (4 bins of 5 obj, C=10)

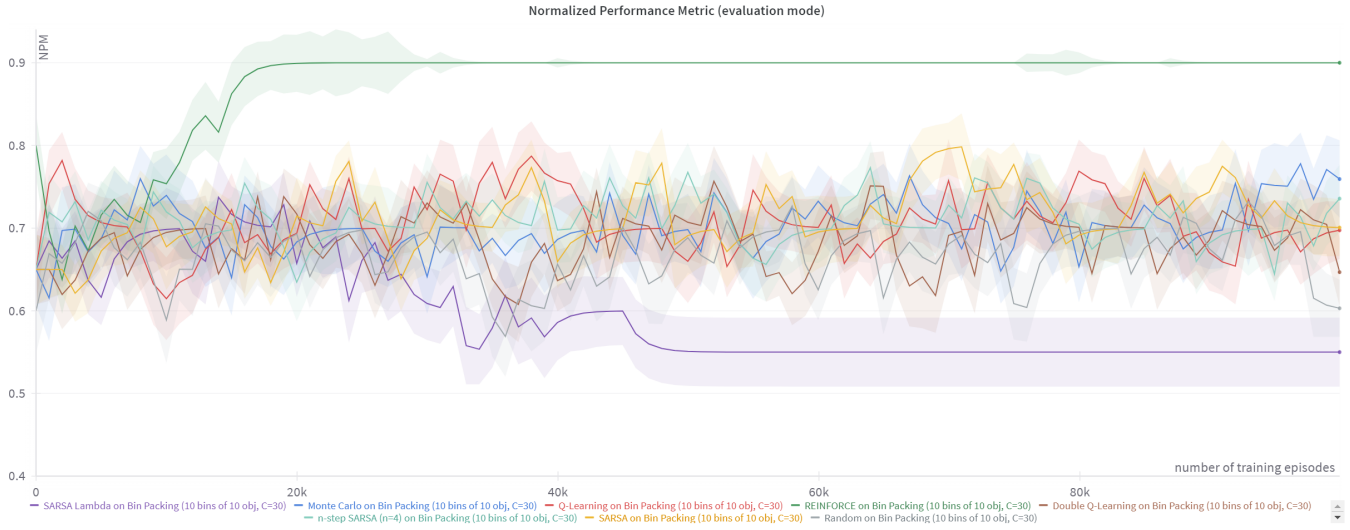

Fig. 10: Evaluation performance on Bin Packing Medium (10 bins of 10 obj, C=30)

Algorithm	Small	Big
Random	65.32	60.3
Q-Learning	**75.00**	69.74
Double Q-Learning	**75.00**	64.67
SARSA	**75.00**	70.06
n-step SARSA	**75.00**	73.57
SARSA(λ)	**75.00**	55.00
Monte Carlo	**75.00**	75.93
REINFORCE	**75.00**	**90.00**

Table 7: Final NPM (in %) on Bin Packing

Bin Packing Problem On the two instances, all the algorithms managed to learn a better solution than random, including REINFORCE. However, detailed results depend on the size of the instance. For the small case, none of the algorithms found the optimal solution, but achieved only 75% of the normalized reward. More precisely, the optimal solution to find was 4 bins, and all find a solution of 5 bins. Since $n_{\text{opt}} = 4$ and $G_{\text{bad return}} = -2n_{\text{optimal bins}}$ we have $\frac{-5-(-8)}{-4-(-8)} = 0.75\%$.

Various factors can explain this problem. Firstly, the complexity of bin packing poses a significant hurdle, as the state space is very sparse. While only the 1 solution exists with precisely four bins by the design of the environment (for small BP), the potential solutions expand substantially with the 5 bins. For example, we generated 10 environment with random sized of objects and computed all possible solutions with 5 bins. We observed an average of 748 possible combinations with 5 bins, contrasting with just one solution with 4 bins. Thus, this environment needs a lot of exploration to find the optimal solution among the huge quantity of almost-optimal ones. Second, the reward, though dense, fails to provide significant information to the agent due to its discretization based on the number of bins. Indeed, many combinations lead to the same reward, 5 for the example of small BP. An alternative approach could involve considering the aggregate sum of spaces available in all bins. However, if implemented densely, this approach would yield poor returns for the agent. On the contrary, if implemented at the end of an episode, it would not provide sufficient information to the agent, given the large number of possible combinations.

We note that REINFORCE reaches the maximum reward from the beginning contrary to the other algorithms. We interpret this as the information of our environment is imperfect since environment with imperfect information can only be optimally learned by policy-based algorithms. Thus, for bin packing, we need a policy-based algorithm. This is also shown in the bigger instance, where REINFORCE get the best results, but not optimal, while the other algorithms even fail to reach the almost-optimal solution, and SARSA performs worse than random.

6.2 Discussion

We see in this benchmark that it was always possible to obtain a better than random solution on the three problems, which is a good sign that RL algorithms can be applied to optimization problems.

However, which RL algorithms to choose depends on the problem and its size. On Knapsack, all the value-based algorithms performed well except n-step SARSA, but REINFORCE was not learning much in comparison to value-based algorithms. On FLP, all algorithms performed well, with Monte Carlo in particular performing very well. On Bin Packing, the results were more mixed, with REINFORCE performing the best, and the value-based algorithms not being able to learn a good policy in the big instance.

Generally speaking, we advise using value-based algorithms for problems with a small state space, in particular Monte Carlo and Q-Learning which performed really well, and policy-based algorithms for problems with a large state space.

Limitations due to the high sparsity of the state space and curse of dimensionality when the instance size becomes larger could be overcome using Deep Reinforcement Learning applied on environments where the instance change at every episode. This will be quickly studied in section 7.

6.3 Infuence of the reward densification

In traditional metaheuristics, the objective function is usually given at the end of the episode, when the solution is complete. With RL, it is possible, instead of giving the objective function as a sparse reward at the end of the episode, to decompose it in a more dense manner, and to give a dense reward to the agent, i.e. a non null reward at each timestep that sums up to the objective function at the end of the episode. This can be done when the objective function can be decomposed in a sum of terms that can be defined at each step of the solution construction. For example, for the Knapsack problem, the objective function is the sum of the values of the items in the knapsack, so it is possible to give the value of the item added at each step as a reward, and the total episodic reward sums up well to the objective function at the end of the episode.

We detail here how we densified the reward for each problem.

Problem	Objective function $J = \sum_{t=0}^{T} r_t$	Dense reward r_t
Knapsack	Sum of the values of the items in the knapsack	Value of the item added at step t
FLP	Final negative cost, with cost being the sum of distances between customer and nearest facility	Loss of cost between solution at $t-1$ and t
Bin Packing	Negative number of bins used	-1 if the item is put in a new bin, 0 otherwise

Table 8: Reward densification

This reward densification allows the agent to learn a policy more efficiently, because it has more feedback at each step of the solution construction. We present here the results of the influence of the reward densification on the performances of the algorithms.

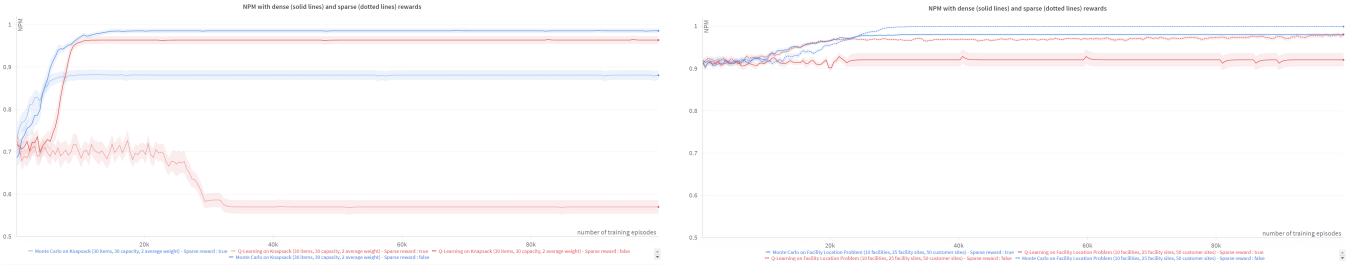

Fig. 11: Performance with dense reward (solid lines) and sparse reward (dashed lines) on Knapsack (left) and FLP (right)

We observe in table 11 a significant improvement of the performance of the algorithms when the reward is densified. On Knapsack Medium, sparsifying the reward decreases the performance of Monte Carlo by 0.1 NPM, and even makes Q-Learning collapse to a very bad solution. On FLP Medium, sparsification also decreases the performance of Monte Carlo and Q-Learning, with Q-Learning in particular not being able to learn anything.

We observe that the loss in performance due to sparsity of the reward in Monte Carlo is lower than in Q-Learning. We conjecture that this is due to the fact that Monte Carlo is already using the full return of the episode to update the Q-values, so this does not change much the learning process. On the other hand, Q-Learning and its bootstrapping mechanism is more sensitive to the sparsity of the reward, because it will have to backpropagate the reward iteratively through the Q-values, which can be difficult if the reward is sparse. In Monte Carlo, there is still a gain in performance with densification because it improves credit assignment, and also reduces the variance since the target is discounted.

We conclude that reward densification is a key factor in the performances of RL algorithms on optimization problems, and that it allows the agent to learn a policy more efficiently.

6.4 Influence of the instance's size

We expect the performance of our RL algorithms to decreases when the size of the instance increases. Due to the curse of dimensionality, the search space increases exponentially with the size of the instance, which makes the problem harder to solve. In tabular RL, this is mainly an exploration problem. The agent has to explore a larger search space, which slows the convergence.

We present here the influence of the instance's size on the performances of the algorithms. We show the NPM curves on Knapsack medium and FLP medium for 4 algorithms : Q-Learning, SARSA(λ), Monte Carlo and REINFORCE. We chose those algorithms because they are representative of the performances of the different types of algorithms that we implemented.

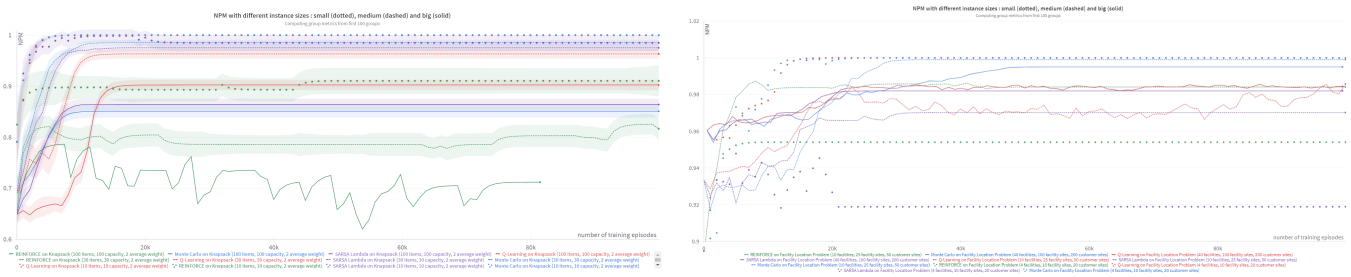

Fig. 12: Performance for different instance sizes on Knapsack (left) and FLP (right) : small (dotted lines), medium (dashed) and big (solid)

We observe in figure 12 left, that the performances of the algorithms decrease with the size of the instance on Knapsack. This is the case for all algorithms. On the small instance, Monte Carlo and Q-Learning manage to find the optimal solution, but on larger instances, the NPM decreases.

For FLP, we observe in figure 12 right, that effect is less pronounced. Performance still decreases with the size of the instance for Monte Carlo and almost for Q-Learning, but the effect is the opposite for SARSA(λ) and REINFORCE, which perform better on the big instance than on the medium instance. We conjecture that this may be due to the fact that those algorithms fell in local minima in the small instances, but that there is less local minima in the big instance, so they can find a better solution. The algorithms whose performance decreases with instance size are also those that perform better globally.

We conclude that the size of the instance has a significant influence on the performances of the algorithms and that the performances usually decrease with the size of the instance. It can happen however that the performance increases with the size of the instance, at least locally, because of some specificities of the problem or the algorithm.

6.5 Choice of the explorative policy

The choice of the explorative policy is crucial in value-based RL, because it determines how the agent explores the search space. We tried three different explorative policies : ϵ-greedy, Boltzmann, and UCB. We present here the influence of the choice of the exploratory policy on the performance of the algorithms. We show the NPM curves on the Knapsack medium and the FLP medium for 3 algorithms : Q-Learning, SARSA(λ), and Monte Carlo.

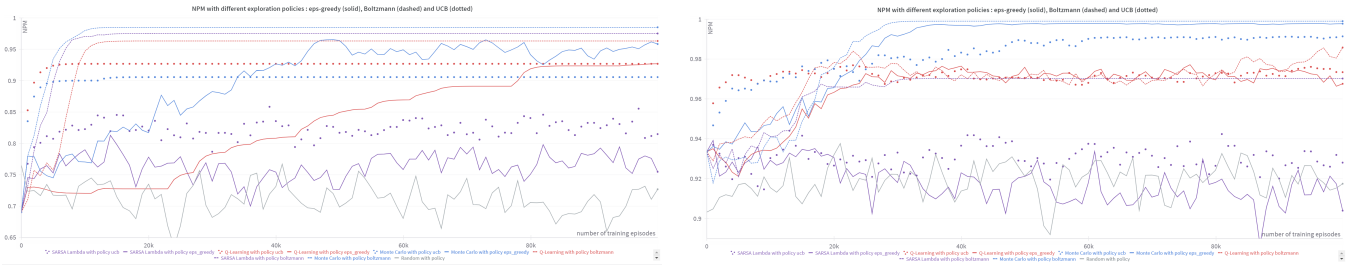

Fig. 13: Performance for different explorative policies on Knapsack (left) and FLP (right) : eps-greedy (solid lines), Boltzmann (dashed) and UCB (dotted)

We observe in Figure 13 that the Boltzmann policy is systematically significantly better than the ϵ-greedy policy and the UCB policy, with the exception of Q-Learning in FLP, where all the policies perform similarly. In SARSA(λ) the Boltzmann policy is even the only policy that allows the algorithm to find a solution significantly better than random in both environments.

The difference between the ϵ-greedy policy and the UCB policy is less pronounced and depends on the algorithm and the environment. This suggests that the performance of a policy in a given environment is often highly dependent on the algorithm used and on hyperparameters.

We conclude that the choice of the exploratory policy is crucial in the performances of the algorithms and that the Boltzmann policy is often the best choice. We conjecture that this is because the Boltzmann policy is Q-value aware, giving more probability to the best actions, which realize an exploitation-exploration trade-off. The ϵ-greedy policy is less efficient because it does not take into account the Q-values, and we conjecture that the UCB policy is too hard to tune or too slow to converge even if proven theoretically converging.

7 Extension to non constant-instance episodes with Deep RL

7.1 presentation of the method

We implemented deep Q learning with an epsilon-gray agent to explore the nonconstant-instance setting. In order to improve the stability of the learning phase (mandatory to obtain non-divergent results), we use two neural networks. The first, the "main network", is the one we train to approximate q values. The second, the "target network", is used only to generate the q-values of the S_{t+1} state in the fundamental Q-Learning equation (based on the Bellman equation):

$$q(S_t, a) = R_{t+1} + \gamma \max_i q(S_{t+1}, a_i)$$

Both networks share the same architecture. The target network is updated with the main network's parameters once all N actions have been chosen, with N large enough to give the main network time to learn its q values from the output of the target network. Furthermore, the main network is not updated after each action chosen, but rather after a batch of k actions (with $1 < k << N$). Here is a detailed description of our parameters:

- We use between 10000 and 15000 episodes for the training (usual time needed to assess the convergence of the algorithms on small and medium instances)
- The number of actions taken between two updates of the main network (i.e. the batch size) is equal to 4
- The number of actions taken between two updates of the target network is equal to 100
- For our neural networks, we use 3 fully connected layers, with hidden dimension equal to 500 and output dimension equal to the maximum possible number of actions to choose from.
- We use the Huber loss between $q(S_t, a)$ and $R_{t+1} + \gamma \max_i q(S_{t+1}, a_i)$ (not the mean square error), in order to increase stability
- We use Adam with learning rate 10^{-4}, and $(\beta_1, \beta_2) = (0.5, 0.99)$

We took into account the fact that only some actions were available at each step by considering the maximum only on available actions in $q(S_t, a) = R_{t+1} + \gamma \max_i q(S_{t+1}, a_i)$.We were also careful not to forget to impose $v(S_{t+1}) = 0$ when S_{t+1} is the terminal state of an episode.

7.2 Results

For this section, due to lack of time (the time required to run the deep learning algorithm being greater than for other algorithms), we focus on the Knapsack problem. Due to instability during training, we were unable to provide any meaningful results for large instances (100 items, 100 capacity, 2 average weight). However, we were able to run our deep learning algorithm on the medium instance (30 items, 30 capacity, 2 average weight) for constant-instance setting and on the small instance (10 items, 10 capacity, 2 average weight) for non-constant-instance setting. In each graph, we keep the plots of the classical q-learning and the random algorithm to allow a comparison of their performances.

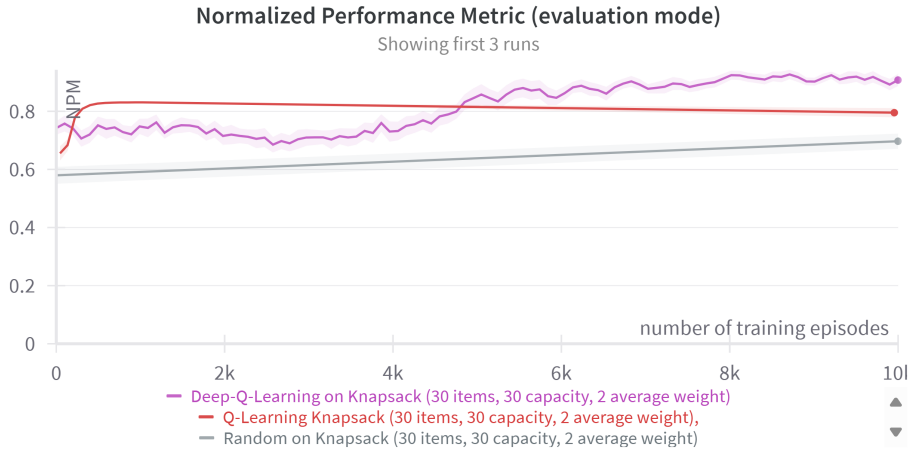

Fig. 14: Evaluation performance on Knapsack medium (30 items, 30 capacity, 2 average weight) for constant-instance setting, with training during 10000 episodes

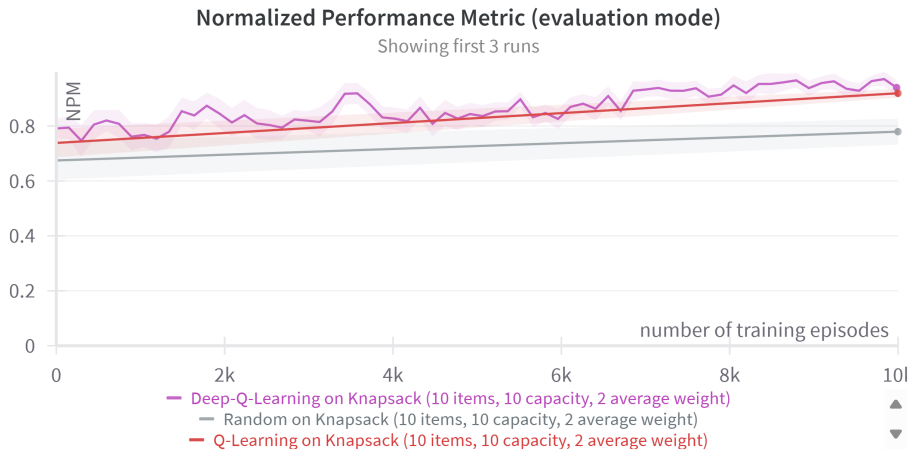

Fig. 15: Evaluation performance of Deep Q learning on Knapsack medium (10 items, 10 capacity, 2 average weight) for non-constant-instance setting with training during 10000 episodes. N.B.: The performance we show for classic Q learning is obtained in a constant-instance setting (it is shown only for comparison with Deep Q learning, which is evaluated in a non-constant-instance setting).

We can see that in both graphs, deep reinforcement learning is just as effective as classic RL. However, it is capable of handling environments whose instance changes during training: this is what we see on Knapsack medium. Once trained, Deep Q-Learning makes almost no errors on this environment. Conventional RL would not have been able to handle this case.

7.3 Limitations of Deep Q-Learning

We did not study Knapsack medium with nonconstant-instance setting because gradients tend to explode. This is a major limitation of our method (and gradient clipping does not produce any results). We would probably need more advanced algorithms (with policy gradient and actor critic) to stabilize learning further. But that is outside the scope of this project.

8 Conclusion

We presented in this report the results of our experiments on the Knapsack, Facility Location, and Bin Packing problems using Reinforcement Learning algorithms. Extensions could be made for extensions of these problems, such as the 2d and 3d knapsack problem solved by heuristic approaches for the moment. [7] We showed that it was possible to apply RL to optimization problems and that the performance of the algorithms was highly dependent on the problem and the size of the instance. We showed that the choice of the exploratory policy and the densification of the reward were crucial factors in the algorithms' performance. We also showed that the size of the instance had a significant influence on the performances of the algorithms and that the performances usually decreased with the size of the instance. These limitations could be overcome using Deep Reinforcement Learning applied to environments where the instance changes at every episode. However, Deep Q-Learning with an epsilon-greedy agent failed to solve the largest instances of our problems. Consequently, we assert that value approximation algorithms alone do not suffice for these scenarios. Incorporating deep learning to learn the policy is probably necessary for effectively exploring the vast state spaces in these cases. We defer this aspect for potential future research endeavors.

References

1. Berkey, J.O., Wang, P.Y.: Two-dimensional finite bin-packing algorithms. Journal of the operational research society **38**, 423–429 (1987)
2. da Costa, J.B.D., Meneguette, R.I., Rosário, D., Villas, L.A.: Combinatorial optimization-based task allocation mechanism for vehicular clouds. In: 2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring). pp. 1–5 (2020). https://doi.org/10.1109/VTC2020-Spring48590.2020.9128834
3. Eglese, R.W.: Simulated annealing: a tool for operational research. European journal of operational research **46**(3), 271–281 (1990)
4. Fabozzi, F.J., Huang, D., Zhou, G.: Robust portfolios: contributions from operations research and finance. Annals of operations research **176**, 191–220 (2010)
5. Heyman, D.P., Sobel, M.J.: Stochastic models in operations research: stochastic optimization, vol. 2. Courier Corporation (2004)
6. Hochbaum, D.S.: Complexity and algorithms for nonlinear optimization problems. Annals of Operations Research **153**, 257–296 (2007)
7. Jens Egeblad, D.P.: Jens egeblad, david pisinger,. Jens Egeblad, David Pisinger, **Volume 36**(Issue 4), Pages 1026–1049 (2009). https://doi.org/https://doi.org/10.1016/j.cor.2007.12.004
8. Lambora, A., Gupta, K., Chopra, K.: Genetic algorithm-a literature review. In: 2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon). pp. 380–384. IEEE (2019)
9. Martello, S., Toth, P.: Algorithms for knapsack problems. North-Holland Mathematics Studies **132**, 213–257 (1987)
10. Owen, S.H., Daskin, M.S.: Strategic facility location: A review. European journal of operational research **111**(3), 423–447 (1998)
11. Papadimitriou, C.H., Steiglitz, K.: Combinatorial optimization: algorithms and complexity. Courier Corporation (1998)
12. Raidl, G.R., Puchinger, J.: Combining (integer) linear programming techniques and metaheuristics for combinatorial optimization. Hybrid metaheuristics: An emerging approach to optimization pp. 31–62 (2008)
13. Salkin, H.M., De Kluyver, C.A.: The knapsack problem: a survey. Naval Research Logistics Quarterly **22**(1), 127–144 (1975)
14. Sbihi, A., E.R.: Combinatorial optimization and green logistics. Ann Oper Res 175 **159-175** (2010). https://doi.org/https://doi.org/10.1007/s10479-009-0651-z