Sub-goal Discovery in Reinforcement Learning

Juliette Rengot
Ecole des Ponts ParisTech - Master MVA
`juliette.rengot@eleves.enpc.fr`

Yonatan Deloro
Ecole des Ponts ParisTech - Master MVA
`yonatan.deloro@eleves.enpc.fr`

Abstract

This project explores reinforcement learning problems where sub-goals can be identified to speed up the learning of the main task. We study and implement two cutting-edge methods: Macro Q-Learning with Diverse-Density-based options ([4]), which exploits commonalities across successful paths throughout learning, and HEXQ algorithm ([5]), which creates automatically a hierarchy of objectives. We test them on problems inspired by taxi domain and compare them to the standard Q-learning algorithm on its flat MDP formulation.

1 Introduction

Enabling an autonomous agent to explore and interact efficiently with an environment is a difficult task. In Reinforcement Learning (RL), the agent learns behaviours on the basis of reward signals provided only when it reaches desired goals. Unfortunately, this standard approach does not scale well for large and complex tasks. One possible solution is to learn sub-goals that can be reused in different contexts. Indeed, many environments present a repetition of sub-structures, but this information is not used in classical RL solutions.

In this project, we focus on sub-goal discovery, and especially on two different state-of-the-art approaches: Macro Q-learning with Diverse-Density based options (we will refer to as DDMQ) and the HEXQ algorithm. We implement these methods in the particular context of the taxi problem. First, we will describe the considered environments. Then, we will present the details of the chosen methods and finally we will focus our attention on the results.

2 Problem definition

The original taxi domain : The taxi task, as originally described by Dietterich ([3]), consists in exploring a 5-by-5 grid world (Figure 1 left): a taxi starts at a random location and navigates to pick-up passengers at their source location and to put them down at their destination location. There are four possible locations (taxi ranks) for passenger source and destination. We suppose that the destination and the location of the passengers are known information. The taxi agent has 6 possible actions : ”up”, ”down”, ”right”, ”left”, ”pick up” and ”put down” (called primitive actions). The rewards are fixed: a successful passenger delivery gives $+20$, a wrong pick up or a wrong put down gives -10, a navigation steps gives -1. This problem has a sub-goal (pick up the people) and information could be reuse (navigation from one rank to another).

A simplified taxi domain : We chose to illustrate the benefits of DDMQ on a simplified problem. We adapt the previous environment so that, the world is a 7-by-7 grid without wall, and, at each episode, passengers appear at the same rank and go towards the same destination (Figure 1 right). The sub-goal consisting in picking up passenger remains. An approach which is indeed particularly suited to this simplified problem is the search for ”bottlenecks” in the observation space, which is one main principle of DDMQ.

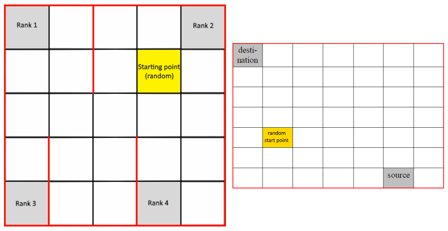


Figure 1: Visual representation of original (Right) and simplified (Left) environments.

3 Description of some possible solutions

3.1 "Basic Flat Q-learning" : an approach without any sub-goal discovery strategy

While very inefficient, the most basic idea to try to solve these problems is to solve the navigation sub-tasks as many times as it reoccurs in the different contexts. We can do so by formulating the problems as MDPs, where, for the original taxi domain, states are defined as triplets composed of the taxi position ($[\![0, 24]\!]$), the passenger position ($[\![0, 4]\!]$: in the taxi or in a rank) and the wished destination ($[\![0, 3]\!]$) ; while, for the simplified taxi domain, there are only two variables : the taxi position ($[\![0, 48]\!]$) and the passenger position $(0, 1)$. We then apply the standard Q-learning algorithm to solve theses MDPs. We will refer to this basic approach as the "basic flat Q-learning".

3.2 Macro Q-Learning with Diverse-Density-based options (DDMQ)

The first approach consists in "mining" the set of the trajectories saved by the agent to find potential sub-goals. To explore more efficiently and accelerate the learning, it allows to follow and assess built-in policies aimed at reaching these sub-goals, instead of primitive actions. The sub-goal discovery algorithm was introduced by Amy McGovern and Adrew Barto ([4]), and used Macro Q-Learning algorithm ([1]) to benefit from these identified potential sub-goals throughout learning.

3.2.1 Presentation of the method

1) Discovering sub-goals candidates : Multiple-Instance Learning and Diverse Density

To discover sub-goals candidates, we look for commonalities across trajectories successful with respect to the final goal. Indeed, sub-goals can be seen as "bottleneck" areas in the observation space. It means that the agent will visit frequently these areas during trajectories reaching the goal, but not on unsuccessful paths.

Modelling the search for bottlenecks as a multiple instance learning problem : The article [4] reformulates the problem of finding bottleneck regions as a multiple instance learning problem (MIL). An instance is defined as an observation made by the agent on its trajectory, typically a visited state. A trajectory is modelled as a bag of instances. Trajectories reaching the final goal within a given number of states are "positive bags" : we assume they contain at least one positive instance explaining their success, but they may contain many negative instances. We may also want to use unsuccessful trajectories as "negative bags", but these are assumed to contain only negative instances in the MIL framework. Looking for bottleneck regions casts to looking for type of instances, or concepts, which are responsible for the positivity of the bag they belong to : these are called "target concepts". If instances are defined as visited states, a concept can for instance simply consists in reaching a given state or it can be more abstract.

Solving the multiple-instance learning problem using diverse density : A solution to such MIL problems is introduced by [2] and consists in finding the region with instances from the most positive bags and the least negative bags. The paper suggests to consider the most diversely dense concept where the diverse density $d(t)$ of a concept t is defined as the probability that it is the target concept among a set of concepts candidates. More precisely, if a uniform prior is assumed on the location of the target concept, and if bags are independent conditionally given it, this definition amounts to : $d(t) := Pr(c_t|B_1^+, ..., B_n^+, B_1^-, ..., B_m^-) = \prod_{i=1}^n Pr(t|B_i^+) \prod_{i=1}^m Pr(t|B_i^-)$, where $Pr(t|B_i^+) =$

$1 - \prod_{1 \le j \le p}(1 - Pr(B_{ij}^+ \in c_t))$ and $Pr(t|B_i^-) = \prod_{1 \le j \le p}(1 - Pr(B_{ij}^- \in c_t))$, denoting $(B_i^+)_{1 \le n}$ the positive bags, $(B_i^-)_{1 \le m}$ the positive bags, and B_{ij} the j-th instance of bag i. $Pr(B_{ij} \in c_t)$ can be modelled based on the distance between the instance and the concept as $\exp -||B_{ij} - c_t||^2$, where the norm definition may or not involve scaling factors of the features if c_t is a vector. One may use exhaustive search or gradient descent to find the target concept (as well as the scaling factors if any).

Keeping only relevant diverse dense concepts : When looking for the diverse density peaks, the paper advises to exclude the states near the terminal states of any successful trajectory. As sub-goal candidate, it also selects peak concepts which appear early and persist throughout learning. To do so, it keeps a running average of how often each concept appears as a peak throughout learning : average m_c for each concept c is initialised to 0 at the first episode, and updated as $m_c \leftarrow \lambda(m_c + 1)$ (λ being strictly inferior to 1) if c is found as a diverse density peak for the set of bags updated with the new trajectory. c is selected as a sub-goal candidate only if m_c goes beyond a given threshold.

2) Learning with discovered sub-goal candidates : Options and Macro Q-Learning

How to use the sub-goals candidates selected at a given iteration from the study of the saved trajectories? We allow the agent to follow built-in policies aimed at reaching a given sub-goal candidate, instead of performing primitive actions. But, it will also need to assess throughout learning the value of taking such a policy from a given state.

Creating an option for a sub-goal candidate : To enable the agent to focus on reaching a particular sub-goal from a given state, the approach makes use of the concept of options. An option is defined as a temporally-extended action which the agent executes until a termination condition is satisfied. While executing the option, the agent chooses action according to the option's own policy. An option is defined by a triplet $< I, \pi, \beta >$ where I is the input set or the set of states in which the option can be initiated by the agent, π is the option's policy defined over the states where the option can execute, and β is the termination condition, defined by a probability $\beta(s)$ that the option terminate in each state s.

Once a sub-goal candidate is selected at a given iteration, the DDMQ algorithm creates an option aimed at reaching it, where: **i)** we add to I, for each saved trajectory performed by the agent, the set of states visited by the agent from time $t - n$ to time t if the sub-goal has been reached at time t, and where n is an hyper-parameter ; **ii)**π, the option's policy, is initialised by creating a value function using the same space of states as the main problem, and the policy value function is learnt using experience replay with the saved trajectories by the agent, and using, instead of the environment reward, a function specific to the option which rewards if the agent reached the sub-goal, and penalises otherwise (and potentially more if he left the input set), **iii)** β is set to 1 when the sub-goal is reached or when the agent is no longer in the input set, and 0 otherwise.

Using Macro-Q-Learning to learn with primitive actions and options : The algorithm to learn with the set of primitive actions augmented by the created options is the Macro Q-learning algorithm introduced by McGovern and Sutton in [1]. The state-action value function Q is defined over the set of states and over the set of the primitive actions augmented with the created options (a column is either an action or an option). For a primitive action a_t taken in state s_t rewarded r_{t+1} and reaching s_{t+1}, update is the same as in classic Q-learning :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$

where α is the learning rate and γ is the discount parameter.

For an option m_t initiated in state s_t, executed from t to $t + n$ where it terminated and reached the state s_{t+n}, and with the actions played according to the option rewarded by the environment $(r_{t+1}, ..., r_{t+n})$, update is the following :

$$Q(s_t, m_t) \leftarrow Q(s_t, m_t) + \alpha \left(\sum_{i=1}^{n} r_{t+i}\gamma^{i-1} + \gamma^n \max_{a'} Q(s_{t+n}, a') - Q(s_t, m_t) \right)$$

3.2.2 Implementation choices for the simplified taxi domain

We applied DDMQ to the simplified taxi domain. The pseudo-code is written in Figure 2.

```
PSEUDO-CODE OF MACRO Q-LEARNING WITH DIVERSE-DENSITY BASED OPTIONS :

INPUT
eps : exploration/exploitation trade-off schedule
N : number of episodes
Tmax : time horizon of an episode
n : number of steps before reaching a concept within a trajectory, considered for initializing the option input set
lambda : factor inferior to 1 such that 1-lambda is the preference to find a peak concept early
threshold_score_concept : threshold on the the running average of how often a concept appears as a peak
gamma_option : discount factor to compute the policy of an option

PROCESS
#initialization
env ← Environment of the taxi problem
Q ← random matrix of size (49*2, 6) such that Q[s, a]=-∞  if the action action is not possible from the state s
Options ← [ ]

memory ←[ ] #database of trajectories
score_concepts ← vector of zeros of size the number of concepts storing the running average of how often a
concept appears as a peak

for k in range(N):
    state ← new initial state
    trajectory ← [ ]
    #MACRO Q-LEARNING
    while (t<Tmax) and (env is not solved):
        save state as state_0
        chosen_action ← choose an action (primitive/option), randomly with a probability eps, otherwise
        the one that maximize Q[state, :]
        if chosen_action is an option :
            while (option is not terminated) and (env is not solved) : #input set left or reached goal
                play action according to option
            Q[state_0, chosen_action] ← updating using Macro Q-Learning formula
        else : #it is a primitive action
            play chosen_action
            Q[state_0, chosen_action] ← updating using classic Q-Learning formula
    add trajectory to memory

    #DIVERSE-DENSITY BASED SUBGOAL DISCOVERY TO CREATE NEW OPTION
    create positive bag from trajectory if successful
    compute the diverse density from all the saved positive bags, and search for diverse density peaks
    for each peak concept c found:
        score_concepts[c] ← lambda * (score_concepts[c]+1)
        if (score_concepts[c] > threshold_score_concept) and (c not in taboo states):
            mark concept as subgoal candidate
    select at random one concept c among the subgoal candidates #max one option created per episode
    create a new option o = <I, \pi, \beta> of reaching concept c
        - I ← union of the states of the trajectories in memory visited n steps before reaching c
        - Q_option ← Q-learning matrix specific to the option computed using experience replay with the
        trajectories in memory, and the reward option instead of the environment reward
        - \pi ← computed with the option value function inferred from Q_option
        - for each state s: \beta(s) ← 1 if s belongs to the concept or is out of the input set I, 0 otherwise
            add option o to Options
            add a column Qc to the Q matrix for the newly created option,  such that  such that Qc[s]
            =-∞  if the state in not in the option input set

pi ← compute policy from Q matrix

OUTPUT
  final policy : pi #associate the index of the primitive action or option to play from each possible state
  list of options : Options: #definitions of options (input set, option policy, terminal condition)
```

Figure 2: Pseudo-code of DDMQ

Considered concepts, metric, and bags : First, we consider as concepts the individual states, ie. the pair composed of the taxi location (x_{taxi}^X, x_{taxi}^Y) and the Boolean x_{people} indicating if the passenger is or not in the taxi. Secondly, we simply chose the norm-2 distance over the features $(x_{taxi}^X, x_{taxi}^Y, x_{people})$ when computing the probability of a state to belong to a concept. With more time, we would have tried to learn a different weight for the squared distance over the x_{people} variable, together with the peak concepts. Third, we only used positive bags (corresponding to trajectories reaching the final goal before the horizon is reached) to compute diverse density values of concepts (negative bags may indeed contain positive instances in our case). Finally, we prevented the agent to create options whose terminal states are the concepts for which the taxi location is in a norm-1 ball of radius 2 around the destination location, and for which the passenger is in the cab (we call these taboo concepts), and we limited the number of options created per episode to 1.

Computing the option's policy To initiate the option's policy whose terminal state is a selected concept, we used the Q-learning algorithm with experience replay over the last 100 trajectories collected by the agent, and the following option reward : 10 if the sub-goal was reached, -1 otherwise (goal not reached but still in the option input set, or left the input set).

3.3 The HEXQ algorithm

Now, we can study a second approach which, instead of building and assessing built-in policies at the same level of primitive actions, introduces a hierarchy of sub-tasks.

3.3.1 Presentation of the approach and its illustration on the taxi domain

The algorithm HEXQ was first presented by Bernhard Hengst ([5]). The main idea is to decompose a Markov Decision Process (MDP) by dividing the state space into nested sub-MDP regions. Sub-MDP are a generalisation of MDP when the time between decisions is variable. This decomposition is possible when :

- some of the variables in the state vector represent features in the environment that change at less frequent times intervals
- variables that change value more frequently retain their transition properties in the context of the more persistent variables
- the interface between regions can be controlled

For our taxi problem, the three state variables (taxi location, passenger location and destination) do not change at the same frequency: taxi location varies at almost all time step whereas the destination location is fixed for the considered trajectory. So, the first condition is satisfy. Moreover, the transitions properties of taxi trajectory are fully defined when it knows where to go and through which point. Those of people trajectory are also determined by the destination location. The second condition is verified. In our context, the last point is not a problem neither because we have only one region at the first level. Finally, we can decompose our taxi problem into several sub-MDP.

Automatic Hierarchical decomposition : Finding a good decomposition to complex problems is a critical stake as the state space size scales exponentially with the number of variables. We construct a hierarchy where the maximum number of levels is the number of state variables. For the taxi problem, we have 3 levels (Figure 3). The bottom level is associated with the variable that changes values most frequently. We first learn the bottom level, that is the only one interacting with the environment using primitive actions. In the taxi domain, the first level will be linked with the taxi location, the second one with the passenger location and the third one with the destination location.

For each level, we partition states into Markov regions. The boundaries between regions are represented by transitions called exits. To go to upper level, we abstract the region and consider it as an abstract state for the new level. Exits policies become abstract actions.

When the taxi will explore its environment, it will first be at level 2 to decide where is the destination rank, then it will be at level 1 to decide where is the people location. Those steps define sub-goals and enables the taxi to be more efficient in its trajectory choices.

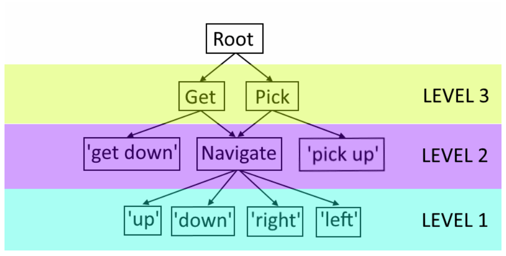


Figure 3: The hierarchy structure of the taxi problem

State abstraction : Abstract state at the level e is defined as the Cartesian product of the region labels and values of the next state variable in frequency order :

$$s^{e+1} = |r^e|x^j + r^e$$

with s^{e+1} the abstract state value at level $e+1$, $|r^e|$ the number of regions at level e, x^j the next most frequent state variable value and r^e the region label from level e. The frequency order is determined by exploring randomly the environment for a given period of time and analysing the obtained statistics. This random exploration is also used to determine exit states (their transitions are unpredictable).

In the taxi example, we only have one region at first level. So abstract states correspond to the passenger location at the second level and passenger destination at the third level.

Action abstraction : The abstract actions, available in each of these abstract states, are the policies leading to region exits at the level below. In the taxi problem, at second level, we have 8 exits :

- $(s^1 = 0, a = pickup)$
- $(s^1 = 4, a = pickup)$
- $(s^1 = 20, a = pickup)$
- $(s^1 = 23, a = pickup)$
- $(s^1 = 0, a = putdown)$
- $(s^1 = 4, a = putdown)$
- $(s^1 = 20, a = putdown)$
- $(s^1 = 23, a = putdown)$

Abstract actions can be interpreted as "going to a rank and interact with passengers". At third level, we have 4 exits :

- $(s^2 = 0, (s^1 = 0, a = putdown)$
- $(s^2 = 0, (s^1 = 4, a = putdown)$
- $(s^2 = 0, (s^1 = 20, a = putdown)$
- $(s^2 = 0, (s^1 = 23, a = putdown)$

Abstract actions can be interpreted as "going to pick up passenger and then put them in a rank".

The Value Function : The recursively optimal hierarchical exit Q-function $Q^*_{em}(s^e, a)$ at level e in the sub-MDP m is the expected value after completing the execution of (abstract) action a starting in (abstract) state s^e and following the optimal hierarchical policy thereafter :

$$Q^*_{em}(s^e, a) = \Sigma_{s'} T^a_{s^e s'} (R^a_{s^e} + V^*_{em}(s'))$$

with $T^a_{s^e s'} = Pr(s'|s^e, a)$, $R^a_{s^e} = E(primitive\, reward\, after\, a|s^e, a)$, $s' =$ hierarchical next state

The recursively optimal hierarchical value function decomposition is given by :

$$V^*_{em}(s) = max_a (V^*_{e-1, m^{e-1}(a)}(s) + Q^*_{em}(s^e, a))$$

with $m^{e-1}(a)$ the sub-MDP implementing action a.

The formulas are similar to the classic Q-learning update. The main difference is that we update on a whole trajectory leading to s^e and not at each time step. We use these formulas to update the Q matrix. We explore the environment at lower levels but we can collect information at upper levels.

3.3.2 Implementation choices for the original taxi domain

In our implementation (Pseudo-code: Figure 4), we suppose that the hierarchy is already constructed. In theory, at the beginning of the training, HEXQ should order the variables and find exits at the first level before it can improve its performances over the taxi domain. In the article, this work takes 41 episodes.

```
PSEUDO-CODE OF HEXQ ALGORITHM :
  INPUT
  eps : exploration/exploitation trade-off schedule
  N : number of episodes
  Tmax : time horizon for an episode

  PROCESS
    #initialisation
    env ← Environment of the taxi problem
    Q_level0 ← random matrice of size (2, 4, 25, 6) such as $\forall (i, j, k, l)$, Q_level0[i, j, k, l]=$-\infty$ if the action $l$ is not possible from the state $k$
    Q_level1 ← random matrice of size (4, 5, 8) such as $\forall (i, j, k)$, Q_level1[i, j, k]=$-\infty$ if the action $k$ is not possible from the state $j$
    Q_level2 ← random matrice of size (4, 4) such as $\forall (i, j)$, Q_level2[i, j]=$-\infty$ if the action $j$ is not possible from the state $i$

    for k in range(N):
      New initialisation of env
      while ((t<Tmax) and env is not solved):
        people ← boolean storing True is people are inside the taxi and False otherwise
        objective ← destination_rank if people are inside the taxi and source_rank otherwise
        state_level0 ← taxi location
        if current hierarchical_level is 0 :
          action ← choose a primitive action, randomly with a probability eps or the one that maximise Q_level0[people, objective, state_level0, :]
          apply the chosen action
          Q_level0[people, objective, :, :] ← Updating with Qlearning
          V_level0[people, objective, :] ← Value fonction associated to Q_level0

          if we reach an exit state of level 1 :
            Q_level1[people, objective, :, :] ← Updating, from Q_level0 and V_level0, with hierarchical formulas
          if we are in exit state of level 2 :
            Q_level2 ← Updating, from Q_level1 and V_level1, with hierarchical formulas

        if current hierarchical_level is 1 :
          state_level1 ← people location
          action ← choose an abstract action of level 1, randomly with a probability eps or the one that maximise Q_level1[objective, state_level1, :]
          Apply the chosen action
          Q_level1[objective, :, :] ← Updating with Qlearning
          V_level1[objective, :] ← Value fonction associated to Q_level1

          if we are in exit state of level 2 :
            Q_level2 ← Updating, from Q_level1 and V_level1, with hierarchical formulas

        if current hierarchical_level is 2 :
          state_level2 ← destination location
          action ← choose an abstract action of level 2, randomly with a probability eps or the one that maximise Q_level2[state_level2, :]
          Apply the chosen action
          Q_level2 ← Updating with Qlearning
          V_level2 ← Value fonction associated to Q_level2

        t ← t+1

  OUTPUT
  Q matrices : Q_level0, Q_level1, Q_level2
  associated value functions : V_level0, V_level1, V_level2
  associated policies: policy_level0, policy_level1, policy_level2
```

(a) Learning the policies.

```
PSEUDO-CODE OF TRAJECTOY SIMULATION WITH HEXQ MODEL :
  INPUT
  env = Environment for taxi domain
  Q = [Q_level0, Q_level1, Q_level2] : list Q matrices at different levels
  V = [V_level0, V_level1, V_level2] : list of value functions at different levels
  policy = [policy_level0, policy_level1, policy_level2] : list of policies at different levels
  Tmax = Time horizon for an episode

  PROCESS
    New initialisation

    # Looking for the destination and people locations
    action ← policy[2][index of destination rank]
    found_destination ← objective associated to action

    action ← policy[1][index of found destination rank][1 + index of source rank]
    found_people ← objective associated to action

    action ← policy[1][index of found people rank][0]
    found_destination ← objective associated to action

    #Navigation
    objective ← index of found people rank
    for i in range(Tmax):
      people ← variable storing 1 if people are in the taxi and 0 otherwise
      if people==1 :
        objective ← index of found destination rank

      action ← policy[0][people][objective][taxi location state]
      apply action
      visualise action
  OUTPUT
```

(b) Simulating a trajectory.

Figure 4: Pseudo-code for HEXQ Model

For level 0, we define a matrix Q_0 of size (2, 4, 25, 6). $Q_0[i, j, k, l]$ is the value of the state-action value function when $people\_inside = i$, $destination = j$, $taxi\_location = k$ and $action = l$. For level 1, we define a matrix Q_1 of size (2, 5, 8). $Q_1[i, j, k]$ is the value of the state-action value function when $people\_inside = i$, $people\_location = j$, $action = k$. For level 2, we compute a matrix Q_2 of size (4, 4). $Q_2[i, j]$ is the value of the state-action value function when $destination\_location = i$, $action = j$. To speed up the learning process, we also assumed that $Q_0[0, j, k, l] = Q_0[1, j, k, l]$ if k is not the people location and that $Q_1[0, j, k] = Q_1[1, j, k] = Q_1[2, j, k] = Q_1[3, j, k]$ if $k! = 0$ (the policy managing the navigation from one point to another is the same whether the taxi intends to pick up or put down the passenger)

4 The results and the comparison of the different algorithms

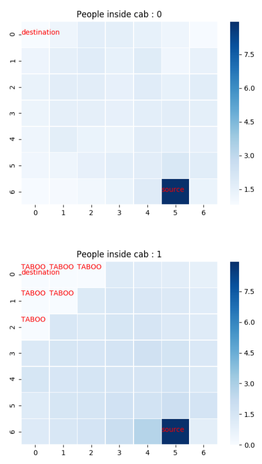
Git repository : Our implementation in Python of the three approaches (basic flat Q-learning, DDMQ, HEXQ) over the corresponding taxi domains can be found in the following repository.

4.1 DDMQ vs. basic flat Q-learning over the simplified taxi domain

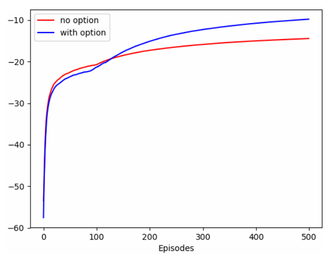
We first compared the Macro Q-Learning approach with Diverse-Density-based options ("MQDD") to the classic Q-learning algorithm using only primitive actions ("basic flat Q-learning") on the flat MDP formulation over the Simplified Taxi Domain (chosen discount factor: $\gamma = 0.95$).

For both algorithms, we used a number of episodes $N = 500$, a time horizon for a given episode $Tmax = 1000$, and an initial exploration-exploitation trade-off $\epsilon = 0.4$ which increases by 0.1 every 100 episodes until reaching 0.9. We also used an adaptive learning rate for the Q-value update of the pair (state,action), proportional to the inverse of the number of previous updates of such value.

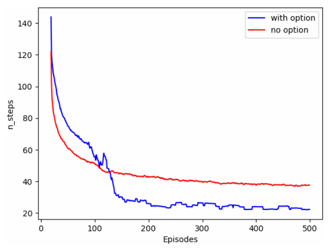
Results are reported in Figure 5. As we can observe, adding options to the set of primitive actions enables to accelerate learning. Acceleration happens around episode 100 where options begin to be created (Figure 5b, 5c). As expected, main sub-goal candidates for options correspond to reaching the source location with or without passengers (Figure 5a). As mentioned in [1], there are potentially two explanations for the benefit of using such relevant options : the agent spends more time exploring the relevant parts of the grid, and Macro Q-learning values propagates more quickly that one step at a time (as in classic Q-learning).



(a) Running average m_c of how often each concept c (taxi location, people presence in the cab) appeared as a peak at episode 50. Average over 30 runs of DDMQ



(b) For each algorithm, at a given episode, we plot the average of the discounted cumulative rewards across all previous episodes.



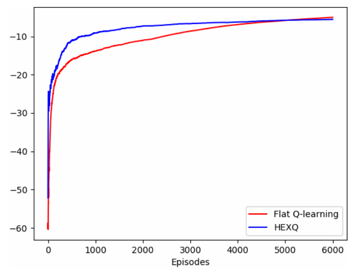
(c) Number of steps to reach the goal throughout the learning. Curbs are average over 30 runs for each of the two algorithms, and moving average over 20 episodes was then applied.

Figure 5: Comparison between DDMQ and basic flat Q-learning over the Simplified Taxi Domain

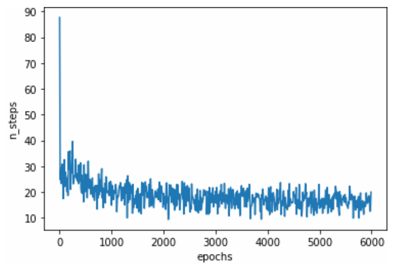
4.2 HEXQ vs. basic flat Q-learning over the original taxi domain

For both flat Q-learning and HEXQ, we use a number of episodes $N = 6000$, a time horizon for a given episode $Tmax = 1000$, a discount factor $\gamma = 0.95$, and an initial exploration-exploitation trade-off $\epsilon = 0.1$ which increases by 0.1 every 200 episodes until reaching 0.9. We use the same adaptive learning rate as in previous experiences.

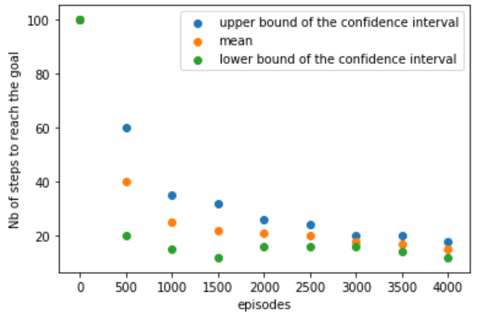
The average of the discounted cumulative rewards stabilises quicker with HEXQ than with the flat Q-learning (Figure 6a). Moreover, during the training process, the number of steps to reach the goal decreases quickly with HEXQ (Figure 6b). Finally, we obtained trajectories of length 15. We also plot the statistics, on 100 trajectories exploiting the HEXQ learnt policy with different number of episodes, of the number of steps needed to reach the goal (Figure 6c). We do the same for the flat Q-learning (Figure 6d) and thus observe the learning acceleration provided by HEXQ.



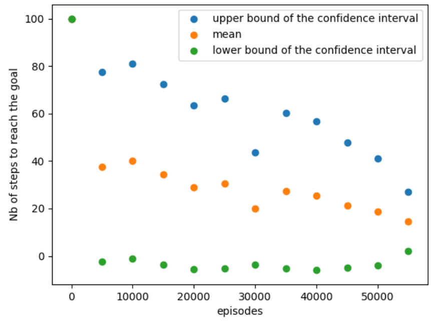
(a) For each algorithm, at a given episode, we plot the average of the discounted cumulative rewards across all previous episodes.



(b) Number of steps to reach the goal throughout the learning, for one run of HEXQ algorithm (moving average over 10 episodes was applied).



(c) HEXQ: Statistics for 100 trajectories, exploiting the learnt policy, of the number of steps to reach the goal. 68% confidence interval (+/- 1 std).



(d) Flat Q-learning: Statistics for 100 trajectories, exploiting the learnt policy, of the number of steps to reach the goal. 68% confidence interval (+/- 1 std).

Figure 6: Comparison between HEXQ and basic flat Q-learning over the Taxi Domain

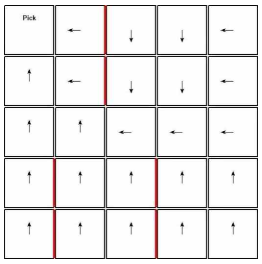
The obtained policies (Figure 7) are coherent and enable us to find relevant trajectories. For example, if people are waiting in 4 (green case) and want to go in 0 (blue case), the taxi (red dot), starting in 20 (yellow case) can reach the goal in 17 steps with a reward of -12.61 (view trajectory!).

In previous experiments, the probability that the chosen action leads to the expected state is equal to one. We try a stochastic case where such probability equals to 0.6. We reach the goal in 26 steps with a reward of -34.53 (view trajectory!). The task is harder but we still obtain satisfying results.

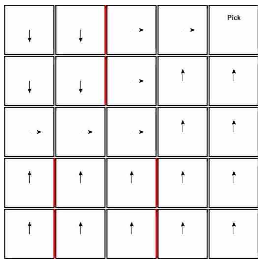
5 Conclusion

To conclude, we have studied and showed the benefits of using two different approaches to solve efficiently RL problems with sub-objectives.

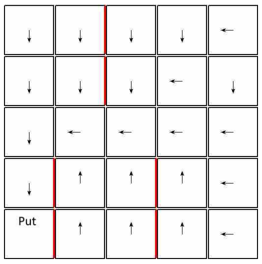
On the one hand, DDMQ seems very efficient for problems where bottlenecks can be found in the observation space. However, it assumes that the agent must be able to reach the overall goal using only primitive actions, which may not be the case in difficult environments. Moreover, defining an



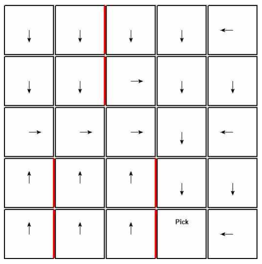
(a) Policy at level 0 when the objective is 0 and people are not in the taxi.



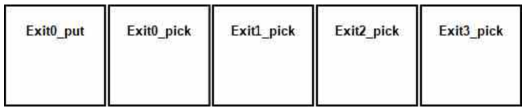
(b) Policy at level 0 when the objective is 4 and people are not in the taxi.



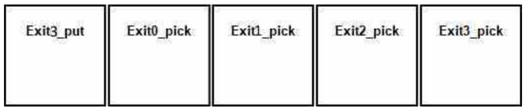
(c) Policy at level 0 when the objective is 20 and people are in the taxi.



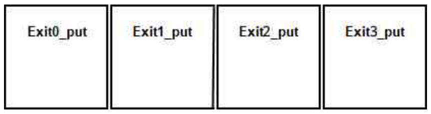
(d) Policy at level 0 when the objective is 23 and people are not in the taxi.



(e) Policy at level 1 when destination is 0.



(f) Policy at level 1 when destination is 23.



(g) Policy at level 2.

Figure 7: HEXQ policies

interesting distance between an instance and a concept, without introducing information about the problem, is a challenge. Finally, this method can suffer from its memory cost (to mine the saved trajectories by the agent) and from its quite large number of hyper-parameters.

On the other hand, the HEXQ exploits efficiently a hierarchical structure of tasks to speed up the training process. There are still some limitations. For example, we did not implement the construction of the hierarchy (frequency analysis and exit discovery) but it implies a computational cost. During a determined period of time, it does not learn the optimal policies, unlike flat Q-learning. In our basic problem, this construction is simple and quick, and the abstraction remains quite intuitive and comprehensible, but we could imagine more difficult tasks. Also, if we can't define some variables that vary at longer timescale than others, we can't use this method.

References

[1] Andrew H. Fagg Amy Mcgovern Richard S. Sutton. "Roles of Macro-Actions in Accelerating Reinforcement Learning". In: In Grace Hopper Celebration of Women in Computing (1997).

[2] Oded Maron and Tomás Lozano-Pérez. "A Framework for Multiple-Instance Learning". In: (1998).

[3] Thomas G. Dietterich. "Hierarchical reinforcement learning with the MAXQ value function decomposition." In: Journal of Articial Intelligence Research 13 (2000), pp. 227–303.

[4] Amy McGovern and Andrew G. Barto. "Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density". In: Computer Science Department Faculty Publication Series. 8. (2001).

[5] Bernhard Hengst. "Discovering Hierarchy in Reinforcement Learning with HEXQ". In: Proceedings of the Nineteenth International Conference on Machine Learning (July 2002), pp. 243–250.