SAPIENZA
UNIVERSITÀ DI ROMA

Final Project

for Electives in: Human-Robot Interaction

and Reasoning Robots

Title: “Improving exploration in Reinforcement Learning with LTL advices provided by humans”

Students’ names: Mahmoud, Elbarbari

Jim Martin Catacora Ocana

José Vicente Jaramillo

1. Introduction

In MDPs, the agent has a complete model of the environment and knows the reward function, so it can use this information to derive a correct behavior, but this model is not always available. The task of reinforcement learning is to use observed rewards to learn an optimal policy for the environment assuming no prior knowledge about the environment. To learn how to behave in the world, the agent will act randomly, as it collects the reward from the environment, will continuously improve its policy until it achieves a policy that maximizes a long term cumulative reward. This interaction with the world can be expensive and unfeasible in real-world environments, or even in simulators, finding an optimal (or nearly optimal) policy in complex worlds can require costly computational resources.

Human-robot collaboration has won attention in the artificial intelligence literature, mainly because as robots become more common in every day's life, the interaction between them can be useful to both parties in doing tasks in a more efficient way. This article uses the assumption that robots can use information given by humans to act properly. Humans will interact by giving an "advice" to the agent (see Section 3) so that it quickly finds useful ways of acting.

The method proposed in this project, follows the work of Toro Icarte et. al [1], and makes use of Linear Temporal Logic for providing advice to an agent (see Section 4), while the R-MAX algorithm is employed as the framework for reinforcement learning (see Section 5), because it allows to easily take advantage of advices for guiding exploration.

It is demonstrated in the experimental section that this approach can improve the learning phase by reducing the number of training steps needed to learn a good policy (see Section 6).

2. Grid-world problem

The example problem considered in this project for testing the performance of the proposed method is described next. The learning agent is placed at some initial location within a grid world. Within the world there is one key, one door, several walls and nails, in addition, the world itself is surrounded by walls. The agent can move deterministically in the four cardinal directions, unless there is a wall or locked door in the way. The agent can only go through the door when it has a key. The agent automatically picks up a key whenever it visits a location with a key. The agent receives a reward of -1 for every action, unless it enters a location with nails (reward of -10) or reaches the door with a key (reward of +1000, and the episode ends).

The following two advices were considered in order to test the method presented below: 1) "Get the key and then go to the door", 2) "Avoid nails".

3. Human Robot Interaction

Humans do learn from trial and error, but certainly not only- Humans also learn from taking advices from other humans. In this project, we construct a

RL agent that can take advices from a non-expert user to guide the agent exploration for a satisfying policy. We thereby design an interface which the non-expert user can use to communicate the advice to the agent using written natural language. Typically, in domain-constrained speech recognition systems, grammar based language models are used. A grammar is an automaton that accepts or rejects word sequences. Using regular-expression syntax, it allows to define the exact utterances the system is to recognize.

Humans usually communicate their advice to other humans in the following four formats:

- *Do something.*
- *Don't do something.*
- *You should do something.*
- *You shouldn't do something.*

So we would expect the advice from the non-expert user in one of the preceding formats. We then look at what kind of words/structure a non-expert user would use. Since we consider only two type of advices: first, avoid the nails, and second, eventually get the key and then eventually get to the door. Here are some examples a non-expert user would use to communicate these two advices to the agent:

- *Avoid the nails.*
- *Don't step on the nails.*
- *You shouldn't walk over the nails.*
- *Go for the key and then go for the door.*
- *You should get the key and then reach the door.*

We use BNF, Backus–Naur form , to define the grammar below.

- <advice> ::= <recommendation> <verb> <preposition> <article> <noun> <EOL> | <recommendation> <verb> <preposition> <article> <noun> "and" "then" <advice>
- <recommendation> ::= "" | "Don't" | "You" "should" | "You" "shouldn't"
- <verb> ::= "avoid" | "go" | "get" | "reach"| "walk" | "step"
- <article> ::= "" | "the"
- <noun> ::= "key" | "door" | "nails"

The above grammar specifications is an automaton that either accepts or rejects a sequence of words /advice the user enters. But this does not tell us which of the pre-defined advices the user is trying to communicate to the agent. So one solution is to build an automata that, on the contrary to the above case, has more than one accepting state. Each accepting state corresponds to one of the predefined advices. And everytime the automata process a sequence of words, a user advice, we check which of the accepting states the auomata terminates. In this case, the advice the user is

trying to communicate is the advice corresponding to the accepting state the automaton terminates.

Another appealing approach is using an RNN e.g. LSTM to classify the user advice input written in natural language (sequence of words) to one of pre-defined advices used in the domain. We also add another class to represent an unknown/unrecognized advice . One concern about this approach is the dataset on which we train the network. Actually on domains similar to the one we use in this project we do not expect more than 10 different advices to be defined, for each advice we may need something like 25 different examples, which means the whole dataset will contain something like 250 instances. This suggests that the approach is a good realistic solution to solve the problem.

After we recognize the advice the user is trying to communicate to the agent. We can use the LTL formula corresponding to the user advice for futher processing and finally guiding the agent exploration using the user advice.

Providing advice based on LTL formulation

3.1. Preliminaries

3.1.1. Signature

Since the aim of this work is to provide advices to learning agents, the first step is to define a signature, i.e. a vocabulary from which an advice can be articulated. A signature is defined as a tuple containing constant and predicate symbols, denoting objects and their properties in the world, and also information about the arity of predicates. Formally, a signature is represented as $\Sigma = \langle \Omega, C, arity \rangle$, where Ω is a finite set of predicate symbols, C is a finite set of constant symbols and $arity : \Omega \to N$ is a mapping that assigns an arity to each predicate.

For the problem dealt with in this project, the signature was composed of only the at predicate having arity 1, where the truth value of $at(c)$ indicates whether the agent is currently at the same location of object c or not. Furthermore, the signature contained one ‘key’ object ($key\,0$), one ‘door’ object ($door\,0$), and 18 ‘nail’ objects ($nail\,0, \ldots, nail\,17$).

Afterwards, given the signature, advices can be specified as LTL formulas, by first forming ground atoms, i.e. instantiating predicates with suitable objects, and then combining ground atoms by means of first-order linear temporal logic (FOLTL) operators.

3.1.2. Labelling function

To work with advices, it is necessary to evaluate at least parts of their corresponding LTL formulas at every state. Because formulas are made up of ground atoms, then it is convenient to create a labelling function, which is essentially a mapping from ground atoms to truth values (i.e. a truth assignment), given the current state.

Formally, the set of all ground atoms associated to a signature is defined as $GA(\Sigma)=\{P(c_1,\ldots,c_{arity(P)}) \vee P\in\Omega, c_i\in C\}$. For convenience, we can also define the set of all ground literals (ground atoms and their negation) as $lit(\Sigma)=GA(\Sigma)\cup\{\neg p : p\in GA(\Sigma)\}$. Then, a truth assignment τ is the set of ground literals ($\tau\subseteq lit(\Sigma)$), where for each element $a\in GA(\Sigma)$ either the ground atom a or its negation $\neg a$ belongs to τ (one of the two must belong). The space of all truth assignments is denoted $T(\Sigma)$. Finally, the labelling function of an MDP with Σ as signature and S as the finite set of all states is defined as $L:S\rightarrow T(\Sigma)$.

In practice, the implementation of the labelling function consists simply in checking the truth value of every ground atom for the current state. In this project, this is even simpler, as it is only relevant to know the current position of the agent (x_R, y_R) and compare it with the position of every object in the world: $\{at(c)=T, if\ (x_R, y_R)=(x_c, y_c); at(c)=F, otherwise \vee ¿\ c\in C\}$.

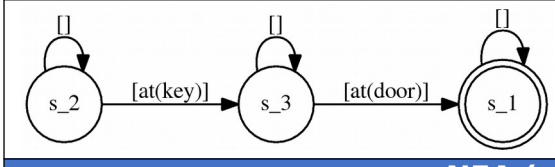
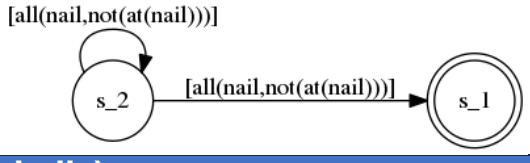
3.2. From LTL to Automata

An LTL formula can be converted to a non-deterministic finite automaton (NFA), so that, said formula is true in the current state iff its associated NFA accepts the sequence of visited states.

The LTL-to-NFA converter developed by Baier and McIlraith [2] was employed in this work, which can be downloaded from [3] (software developed in Prolog). This converter was specifically created to deal with first-order LTL, which is the language needed for giving advice due to its broad expressivity. It is also worth mentioning that this converter, instead of producing a single large NFA, generates several smaller NFAs.

The table below shows graphical and symbolic representations of 2 NFAs produced by the Baier-McIlraith converter for the advices: “Get the key and then go to the door” and “Avoid nails”:

Table 1. Two advices and their corresponding NFAs.

Linguistic advice	
Get the key and then go to the door	Avoid nails
LTL formula	
$\Diamond(at(key\,0)\wedge\bigcirc(\Diamond at(door\,0)))$	$\Box\forall(n\in nails).\neg at(n)$
NFA (graphical)	
	
NFA (symbolic)	

```
digraph automaton_for_ltl {
    initial_state=s_2
    final_states={s_1}
    input_symbols={'[at(key)]','[at(door)]'}
    s_2->s_2 [label='[]'];
    s_2->s_3 [label='[at(key)]'];
    s_3->s_3 [label='[]'];
    s_3->s_1 [label='[at(door)]'];
    s_1->s_1 [label='[]'];
}
```

```
digraph automaton_for_ltl {
    initial_state=s_2
    final_states={s_1}
    input_symbols={'[all(nail,not(at(nail)))]'}
    s_2->s_2 [label='[all(nail,not(at(nail)))]'];
    s_2->s_1 [label='[all(nail,not(at(nail)))]'];
}
```

As seen above, each NFA has its own set of symbols, i.e. inputs to the automaton, which in general correspond to FOLTL subformulas found in the main advice formula. NFAs read only those particular symbols as inputs; hence, it was necessary to create a specific function for each NFA that given the truth assignment τ for the current state s (extracted from the labelling function L), indicates which symbol is sent to said NFA at that timestep, if any.

For example, for the advice “Get the key and then go to the door”, when the agent reaches the key, the symbol $at(key0)$ is sent to the associated NFA, but when the agent is neither at the key nor at the door, no new symbol is sent to the automata. These assignment-to-symbol functions simply evaluate for each symbol (by carrying out Boolean operations), if their corresponding subformulas are true in view of the known current assignment, and if so, the symbol is sent.

In this project, the NFAs generated by the Prolog converter as symbolic representation were used to manually initialize corresponding NFAs with the Python library Automata, which can be found in [4]. Said NFAs instances begin with a state set containing only the NFA initial state; then, they are fed at each timestep with an up-to-date sequence of symbols in order to: 1) check whether the sequence is accepting or not, and 2) if it is, to compute the current NFA state set. The Automata library automatically produces the previous two outputs given that the NFA receives a proper sequence of symbols.

The above procedure may lead to an empty NFA state set on some NFAs and state sequences. For example, for the advice “Avoid nails”, there is no transition to follow after the agent enters a state with a nail. In other words, the advice formula can no longer be satisfied during the current learning episode. Such situations are called NFA dead-ends.

However, since the advice may still be useful even if it has been violated, NFA dead-ends are handled as follows: if an NFA state set becomes empty, the corresponding input sequence is not updated with the last symbol. That is, bad transitions are simply ignored, and everything continues as if the agent never made a mistake.

3.3. Heuristic associated with an advice

3.3.1. Background Knowledge Functions

For an advice to be useful, it has to be associated with a heuristic function, that gives the agent a little intuition on how to proceed (which action to take) towards the goal. This project considered a bottom-up approach, such

that heuristics are initially defined for literals, and then heuristics for advices are constructed by combining lower-lever heuristics.

For this project, heuristics associated with literals are called Background Knowledge Functions. Their goal is to optimistically inform how many primitive actions might be needed to make the corresponding literal true. Formally, a background knowledge function is defined as $h_B: S \times A \times lit(\Sigma) \to N$, where $h_B(s,a,l)$ is an estimate of the number of primitive actions expected before reaching a state where the literal l is true.

In this work, heuristic functions were formulated for $at(c)$ and $\neg at(c)$, where c is some object in the world. For the case of $at(c)$, the following heuristic was employed:

$$h_B(s,a,at(c)) = |pos(agent,s).x - pos(c,s).x| + |pos(agent,s).y - pos(c,s).y| + \Delta$$

Where $pos(\cdot,s)$ provides the planar coordinates of some object in state s, and Δ is equal to -1 if action a points towards c from the agent's position and 1 if it does not. Furthermore, the heuristic formulated for $\neg at(c)$ is shown below:

$$h_B(s,a,\neg at(c)) = \begin{cases} 1, & if\ h_B(s,a,at(c)) = 0 \\ 0, & otherwise \end{cases}$$

3.3.2. Combining heuristics

The background knowledge functions associated with every literal conforming an arbitrary advice formula $\varphi \in L_\Sigma$ can be combined into a single heuristic $h: S \times A \times L_\Sigma \to N$ according to the following recursive rules:

$$h(s,a,l) = h_B(s,a,l)\ for\ l \in lit(\Sigma)$$

$$h(s,a,\psi \wedge \chi) = max\ \{h(s,a,\psi), h(s,a,\chi)\}$$

$$h(s,a,\psi \vee \chi) = min\ \{h(s,a,\psi), h(s,a,\chi)\}$$

3.4. Advice-Based Action Selection

3.4.1. Advice guidance

Suppose the agent is at a state s in the MDP, with NFA state sets $q^{(0)},\ldots,q^{(m)}$, one per each automaton generated by the LTL-to-NFA converter. Intuitively, following the advice in s involves having the agent take actions that will move the agent through the edges of each NFA towards its accepting states. Therefore, in order to define a more accurate heuristic function for an advice, useful edges were identified and combined into an advice guidance formula $\hat{\varphi}$.

The set of useful edges in the i-th NFA is defined as $(q,\beta,q') \in useful(q^{(i)})$, where q belongs to the current state set $q^{(i)}$, there exists an edge from q to q' labelled by the formula β, and an edge is marked as useful if q is not an accepting state and there exists a path in the NFA from q' to an accepting state that does not have q along it. Then, $\hat{\varphi}$ can be expressed as follows:

$$\hat{\varphi} = ¿\, i=0 \,¿\, m \left[¿\, (q,\beta,q') \in useful(q^{(i)})\ ¿_{DNF}(\beta) \right]$$

Where, $to_{DNF}(\beta):2^{L_\Sigma} \to L_\Sigma$ converts the formula β to disjunctive normal form. In the end, the heuristic associated with the guidance will be computed, $\hat{h}(s,a)=h(s,a,\hat{\varphi})$.

3.4.2. Advice warning

Similarly, to avoid NFA dead-ends, an advice warning formula $\hat{\varphi}_w$ is constructed as follows:

$$\hat{\varphi}_w=¿i=0¿m\left[¿q\in q^{(i)}\wedge(q,\beta,q')\in\delta^{(i)}¿_{DNF}(\beta)\right]$$

Where $\delta^{(i)}$ denotes all edges (q,β,q') of the i-th NFA. Then, the previous formula is used to define the set $W(s)=\{a\in A(s):h(s,a,\hat{\varphi}_w)\neq 0\}$ that contains the actions that will lead to NFA dead-ends.

4. Incorporating advice in R-MAX

Reinforcement Learning methods often require extensive exploration of the environment, which can be infeasible in real-world cases where exploration can be very computationally expensive at the very least. Thus, we investigate the use of human advice as a means of guiding exploration. R-MAX is a model-based RL algorithm that solves MDPs by exploring the environment by assuming that unknown transitions give maximal reward Rmax. In practice, if Rmax is big enough, this means that the agent plans towards reaching the closest unknown transition.

We propose a simple variant of an R-MAX algorithm that can take advantage of human advice. Instead of always planning towards the closest unknown transition, we execute the action with minimum heuristics value (advice guidance heuristic), if it executes an unknown transition; and if there are no unknown transition that can be executed at this time step, we plan toward the closest unknown transition. We use simple BFS to plan to the nearest unknown transition and Value Iteration to compute the optimal policy over the learned model. The pseudocode of this algorithm is shown below:

Box 1. Pseudo of R-MAX algorithm.

```
Function Rmax with advice(S, A, L, h_B, φ_advice, N):
    T_unknown ← Ø
    p̂ ← initialize empty model()
    for s, a ∈ S × A do
        T_unknown ← T_unknown ∪ (s, a)
    t ← 0; π ← Ø; s ← get initial state();
    while t < N do
        t ← t + 1
        if is terminal state(s) or cannot be reached then:
            s ← get initial state()

        if any of a ∈ A executes unknown transition then:
            a ←choose a with min heuristics
        else:
            π ← policy towards nearest unknown transition
            a ← π(s)
        s_prime, r ← execute action(s, a)
        p̂ ← update model
        T_unknown ← T_unknown \ (s, a)
        s ← s_prime
return computed optimal policy(p̂)
```

5. Experiments

5.1. Settings

We compare two learning methods over the grid-world problem defined earlier: 1) using standard R-MAX, and 2) using R-MAX with human advice. In each case, we try two different pieces of advice: *"Get the key and then go to the door"* and *"Avoid nails"*. We perform around 50 experiments on 50 different 20x20 grid worlds for each method and advice.

The worlds are randomly generated with a probability 0.6 for each cell to be empty, 0.2 to have a nail and 0.2 to have a wall. In each experiment, we run 2000 training steps maximum. Every 200 training steps, we do an evaluation step by performing value iteration and look at the best policy we can get given the model the agent has learned so far.

Afterwards, results from 100 experiments were first averaged, and then normalize between 0 and 1, where 1 represents the maximum reward possible. Note that, if the agent cannot learn a policy that brings him to the final goal, we assume the reward to be 0.

5.2. Results

Figure 1 shows the results of using the *"Get the key and then go to the door"* advice against standard R-MAX. It is clear from the results below that this advice allows the agent to quickly find a policy that solves the problem and then slowly converges to an optimal policy. We therefore believe that advice shall play an important role in RL, since some tasks are too big to be tackled using only trial and error approach.

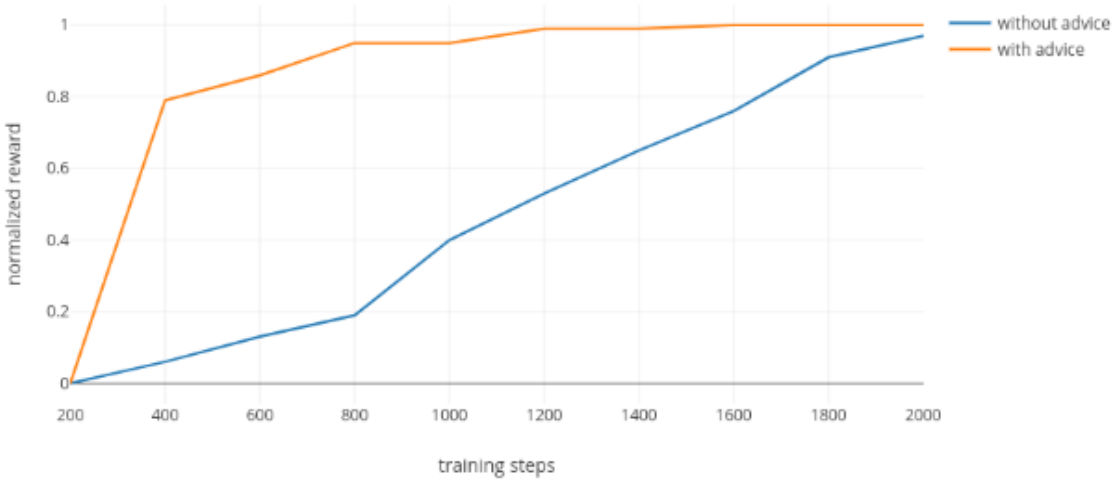


Figure 1. Advice "Get the key and then go to the door".

Figure 2 shows the results of using the *"Avoid nails"* advice against standard R-MAX. Although this advice seems well-intended, it is actually not a great advice, because during the first episodes it often prevents the agent from discovering any policy leading to the final goal. We, therefore, note that as the quality of the advice decreases, the agent's initial performance also does, but the agent still converges to the optimal policy at the end. The latter is a result of the fact that advices are treated like soft constraints that can be violated, instead of hard constraints that cannot be violated.

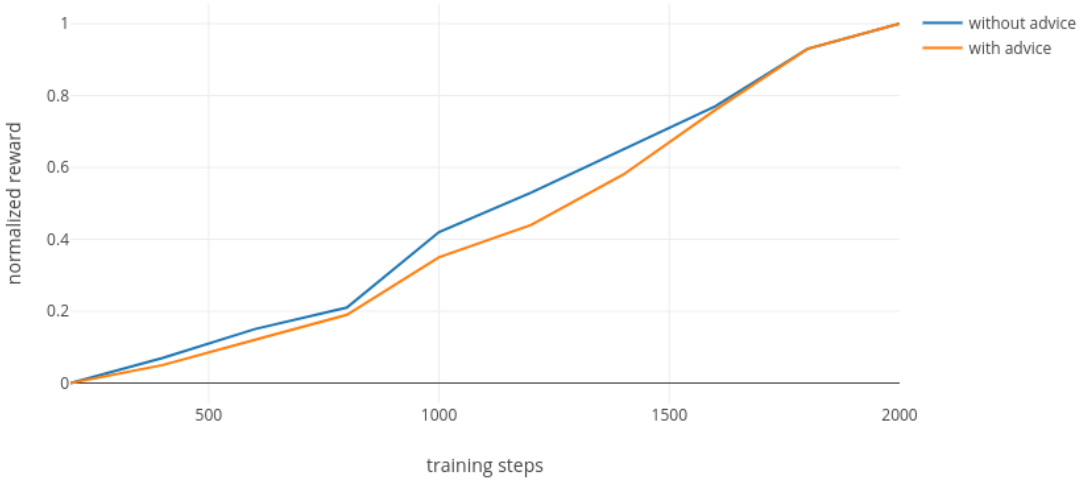


Figure 2. Advice "Avoid nails".

6. Conclusions

From the human-robot-interaction perspective, the goal of this project was to experiment with a method that allows human designers to provide and incorporate advices into the learning process of an agent in order to mainly reduce its learning time. In this way, designers could now guide or warn agents through the difficulties of the given problem by means of suggestions, but without constraining them to any specific execution. We observe that this method could be particularly valuable for handling problems with sparse rewards (where rewards are usually given at the end of a long episode), as designers might be able to manually divide the problem into a series of simpler problems and define an associated list of sequential sub-goals, which could then be encoded within an LTL formula to form a very informative advice.

With respect to the LTL formulation, human-advices were incorporated into learning by: 1) establishing a signature, i.e. a common vocabulary that can be understood by humans and agents made out of predicates and world-objects (advices can then be expressed in first-order LTL language given this signature), 2) constructing heuristic functions associated with these advices, composed from background knowledge functions defined at the level of predicates (heuristics guide agents into following the corresponding advices).

Signatures and background knowledge functions have to be specified manually by the designers according to the proposed method. Clearly, an exhaustive definition (having every conceivable predicate) would be quite cumbersome or downright intractable for very complex problems involving sophisticated agents. Therefore, from a practical perspective, it is convenient to, first come up with the advices and then develop a reduced signature, expressive enough to handle only those advices. This puts in evidence a limitation of this method, as the LTL formulation has to be manually extended for every new advice one wishes to try out that contains predicates not considered so far in the signature. Likewise, the applicability of the method is confined to designers; leaving out end-users, who would like to provide any reasonable piece of advice to a learning agent (hence, an exhaustive signature would be mandatory in that case).

Finally, from the reinforcement learning point of view, our experiments show that the use of advices can considerably accelerate learning towards a high-performing policy (depending heavily on the quality of the advice); however, learning the optimal policy takes even longer than without advices. This points out that the proposed method is best applied over large, computationally demanding problems, where finding the optimal policy is less important than obtaining a sufficient approximate solution.

7. References

[1] Toro Icarte, R., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: Advice-Based Exploration in Model-Based Reinforcement Learning. In: Advances in Artificial Intelligence, Canadian AI. (2018).

[2] Baier, J., McIlraith, S.: Planning with first-order temporally extended goals using heuristic search. In: AAAI. (2006) 788-795.

[3] http://www.cs.toronto.edu/~jabaier/planningTEG/

[4] https://github.com/caleb531/automata