Synthesizing Evolving Symbolic Representations for Autonomous Systems

Gabriele Sartor[1*], Angelo Oddi[2], Riccardo Rasconi[2], Vieri Giuliano Santucci[2], Rosa Meo[1]

[1*]Computer Science Department, University of Turin, Via Verdi 8, Turin, 10124, , Italy.
[2]Institute for Cognitive Sciences and Technologies, Via G. Romagnosi 18A, Rome, 00196, , Italy.

*Corresponding author(s). E-mail(s): gabriele.sartor@unito.it;
Contributing authors: angelo.oddi@istc.cnr.it;
riccardo.rasconi@istc.cnr.it; vieri.santucci@istc.cnr.it; rosa.meo@unito.it;

Abstract

In recent years, Artificial Intelligence (AI) systems have made remarkable progress in various tasks. Deep Reinforcement Learning (DRL) is an effective tool for agents to learn policies in low-level state spaces to solve highly complex tasks. Recently, researchers have introduced Intrinsic Motivation (IM) to the RL mechanism, which simulates the agent's curiosity, encouraging agents to explore interesting areas of the environment. This new feature has proved vital in enabling agents to learn policies without being given specific goals.
However, even though DRL intelligence emerges through a sub-symbolic model, there is still a need for a sort of abstraction to understand the knowledge collected by the agent. To this end, the classical planning formalism has been used in recent research to explicitly represent the knowledge an autonomous agent acquires and effectively reach extrinsic goals. Despite classical planning usually presents limited expressive capabilities, Probabilistic Planning Domain Definition Language (PPDDL) demonstrated usefulness in reviewing the knowledge gathered by an autonomous system, making explicit causal correlations, and can be exploited to find a plan to reach any state the agent faces during its experience.
This work presents a new architecture implementing an open-ended learning system able to synthesize from scratch its experience into a PPDDL representation and update it over time. Without a predefined set of goals and tasks, the system integrates intrinsic motivations to explore the environment in a self-directed way, exploiting the high-level knowledge acquired during its experience. The system

explores the environment and iteratively: (a) discover options, (b) explore the environment using options, (c) abstract the knowledge collected and (d) plan. This paper proposes an alternative approach to implementing open-ended learning architectures exploiting low-level and high-level representations to extend its own knowledge in a virtuous loop.

Keywords: artificial intelligence, abstraction, PPDDL, open-ended learning

1 Introduction

In the last few years, new AI systems have solved incredible tasks. These tasks include real-world games, such as chess [1] and Go [2–4], videogames such as Atari [5], Dota [6], and different robotics tasks [7–10]. These results have been mostly achieved through the intensive use of Reinforcement Learning (RL, [11]) with the rediscovered technology of neural networks and deep learning [12]. Usually, "standard" RL focuses on acquiring policies that maximise the achievement of fixed assigned tasks (through reward maximisation) with a predefined collection of skills. New approaches have been proposed to enrich RL, allowing the agent to extend its initial capabilities over time inspired by neuroscience and psychology. Indeed, studies on animals [13–15] and humans [16–18] have explored the inherent inclination towards novelty, which is further supported by neuroscience experiments [19–21]. The field of intrinsically motivated open-ended learning (IMOL [22]) tackles the problem of developing agents that aim at improving their capabilities to interact with the environment without any specific assigned task. More precisely, Intrinsic Motivations (IMs [23, 24]) are a class of self-generated signals that have been used to provide robots with autonomous guidance for several different processes, from state-and-action space exploration [25, 26], to the autonomous discovery, selection and learning of multiple goals [27–29]. In general, IMs guide the agent in acquiring new knowledge independently (or even in the absence) of any assigned task to support open-ended learning processes [30]. This knowledge will then be available to the system to solve user-assigned tasks [31] or as a scaffolding to acquire new knowledge cumulatively [32–34] (similarly to what has been called curriculum learning [35]).

The option framework has been combined with IMs and "curiosity-driven" approaches to drive option learning [32] and option discovery [36–39]. In the hierarchical RL setting [40], where agents must chunk together different options to properly achieve complex tasks, IMs have been used to foster sub-task discovery and learning [41–43], and exploration [26]. Autonomously learning and combining different skills is a crucial problem for agents acting in complex environments, where task solving consists of achieving several (possibly unknown) intermediate sub-tasks that are dependent on each other. An increasing number of works are tackling this problem [29, 44, 45], most focused on low-level, sub-symbolic policy learning [46], in turn combined in a hierarchical manner using some sort of meta-policy [47]. While promising, these approaches necessarily face the problem of exploration, which becomes slower and less efficient as the space of states and actions increases.

In contrast to sub-symbolic methods, symbolic approaches like Automated Planning [48, 49] enable the use of higher-level objects (referred to as symbols), resulting in quicker execution, facilitating the composition of complex sub-task sequences, and making the agent's internal knowledge interpretable. However, Automated Planning approaches require that the high-level representation of the planning domain is appropriately defined in advance. Generally, planning requires prior knowledge of the world in which the agent operates expressed in terms of both the preconditions necessary for the execution of the actions as well as the effects that follow from executing them. The need to be provided with an *ad-hoc* symbolic representation of the environment limits the utilization of high-level planning for artificial agents in unknown or highly unstructured settings, where the acquisition of new knowledge and new skills is the progressive result of the agent's autonomous exploration of the environment. However, some works tried to improve classical planning with autonomous model learning [50], suggesting to add new symbols to the symbolic representation supported by the human [51] and using a cognitive layer to manage an intermediate representation [52].

Recently, some ideas have appeared in the literature proposing methodologies for integrating sub-symbolic and symbolic approaches, or more generally, low-level and high-level modules [53]. On the one hand, some works tried to reconcile deep learning with planning [54, 55], goal recognition [56] and the synthesis of a symbolic representation of the domain [57, 58]. On the other hand, the integration has also been performed through a specific algorithm designed to produce an automated symbolic abstraction of the low-level information acquired by an exploring agent [59] in terms of a high-level planning representation such as the PDDL formalism [60], which explicitly describes the context necessary to execute an action on the current state (i.e., the *preconditions* and the *effects*) making use of symbols. This algorithm has been used as a module in architectures that integrate abstraction, planning and intrinsic motivations, such as IMPACT [37, 61].

This work presents an approach to the integration of low-level skills and high-level representations that allows to continuously update the set of low-level capabilities and their corresponding abstract representations. Both the creation and the extension of the agent's knowledge is based on the implementation of two forms of intrinsic motivation (IM) which, respectively, (i) drive the agent to learn new policies while exploring the environment and (ii) encourage it to use its skills to reach less explored states, the rationale being that exploring unknown states increases the likelihood to learn new skills. Then, the data collected by the agent's sensors before and after the execution of its skills are used by a specific algorithm to synthesize an updated abstract representation which can be used to plan the execution of sequences of low-level skills to reach more complex goals. The main contribution of this study is to create a framework that, virtually starting from zero symbolic knowledge, produces an abstraction of the low-level data acquired from the agent's sensors, whose enhanced expressiveness can be exploited to plan sequences of actions that reach more and more complex goals.

2 Background

To reach a high level of autonomy, an agent acting in the low-level space, sensing the environment with its sensors and modifying it through its actuators must implement a series of layers of abstraction over its state and action spaces. As human beings reason over both simple and complex concepts to perform their activities, so robots should be able to build their own abstract representation of the world to deal with the increased complexity, using *labels* to refer to actions and events to be recognized and reasoned upon. In this paper, two level of abstractions are applied: the first one, from primitive actions to *options* [11, 62] and the second one, from options to classical planning [49].

2.1 From primitives to options

As discussed before, at the lowest level, the agent sees the world with its sensor's values and changes it through the movement of its actuators. The most common formalism at this stage to deal with this type of representation is the Markov Decision Process (MDP), which models the environment as the tuple:

$$(S, A, R, T, \gamma), \tag{1}$$

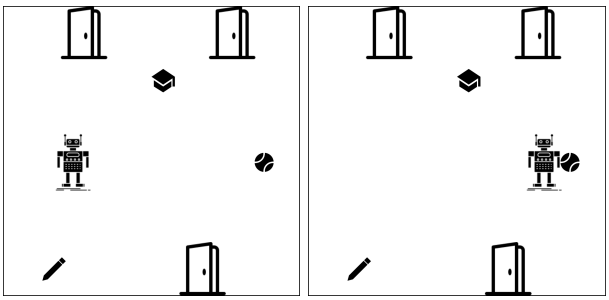
in which S represents the set of possible high-dimensional states where each $s \in S$ is described by a vector of real values returned by the agent's sensors, A describes the set of low-level actions $a \in A$ in some cases also called *primitives*, R the reward function where $R(s, a, s')$ is a real value returned executing a from state s achieving s', T the transition function describing for $T(s'|s, a)$ the probability of reaching the state s' executing a from s, and the discount factor $\gamma \in (0, 1]$ describing the agent's preference for immediate over future rewards. Usually, in this setting, the goal is to maximize *return*, defined as

$$\mathcal{R} = \sum_{i=0}^{\infty} \gamma^i R(s_i, a_i, s_{i+1}). \tag{2}$$

However, dealing with the state and action spaces of the formulation (1) is, in certain cases, impractical due to the high dimensional spaces considered. An effective formalism introduced to reduce the complexity of the problems is the *option* framework [62]. The *option* is a temporally-extended action definition which employs the following abstracted representation:

$$o = (I_o, \pi_o, \beta_o), \tag{3}$$

where the option o is defined by an *initiation set* $I_o = \{s|o \in O(s)\}$ representing the set of states in which o can be executed, a *termination condition* $\beta_o(s) \rightarrow [0, 1]$ returning the probability of termination upon reaching s by o, and a policy π_o which can be run from a state $s \in I_o$ and terminated reaching s' such that the probability $\beta_o(s')$ is sufficiently high. A *policy* is a function defining the behavior of the agent, mapping the perceived state of the environment to the action to be taken [11]. It is worth noting that options create a temporally-extended definition of the actions [62].



(a) (b)

Fig. 1: Representation of the option *reach the ball*.

Indeed, the option is an abstraction defining an action as a repeated execution of policy π from a state $s \in I_o$ to another s' in a maximum amount of time steps τ.

For instance, Figure 1 depicts an environment before and after executing the option *"reach the ball"*, respectively Figure 1a and 1b. When the agent wants to execute this option, it checks whether its current state belongs to the *initiation set* of the option. In our case, and assuming that the option's policy simply consists in moving towards the ball, the agent can execute the option and runs the policy π_o until the *termination condition* returns a sufficiently high probability of success or τ time steps are reached. In the Figure 1b the options successfully terminates getting close to the ball.

Passing from low-level actions to options reduces the agent's action space. Adopting options in the MDP formalism implies moving to the semi-Markov Decision Process (SMDP):

$$(S, O, R, P, \gamma), \tag{4}$$

where S is the original state space, $O(s)$ is the set of options executable from state s, $R(s, \tau|s, o)$ describes the reward expected executing $o \in O(s)$ from state s reaching s' after τ time steps, $P(s', \tau|s, o)$ returns the probability of reaching state s' after τ time steps executing $o \in O(s)$ from state s, and the discount factor $\gamma \in (0, 1]$. Using options entails moving in the low-level state space S with abstracted actions permitting the agent to perform extended and more complex behaviors, reducing the number of actions to achieve a certain task and simplifying the problem.

2.2 Options and Classical Planning

The option formalism and its way of abstracting the dynamics of an environment share common characteristics with *classical planning*, in which the world is described in a simplified formal description considering only the aspects necessary to solve the agent's task [49]. Planning is the field of research studying formal methods to automatically find solutions, also called plans, to tasks requiring a sequence of actions $[\alpha_1, \ldots, \alpha_n]$ to reach a goal state s_g from an initial state s_{init}. The plan ω is obtained by giving in input a model of the environment dynamics and the problem definition to a planner, returning a solution applying general optimization algorithms.

Classical planning is a particular instance of this methodology exploiting a logical language (i.e. propositional logic, first-order logic, etc.) to capture an abstract symbolic description of the world [49]. The *symbol*, which is the core of classical planning, is a name given to a certain set of states $s \in S$ satisfying a specific condition in the sensorimotor space. This *mapping* from the symbol to the real world states is called *grounding*. Specifically, a symbol $\sigma_Z \in \Sigma$, with Σ set of the available symbols, is the name given to a *test* τ_Z, and the corresponding set of low-level states where the test is satisfied $Z = \{s \in S \mid \tau_Z(s) = True\}$, with the high-dimensional low-level state space S [59]. To determine whether or not a low-level state s_i belongs to the state set Z, the $\sigma_Z(s_i)$ test is run, which will return either $True$ or $False$. Again, both Z and σ_Z provide the *semantics* of the symbol; Z is the symbol's *grounding set*, while σ_Z is its *grounding classifier*. Symbols can be combined using *operators* having the following meaning:

- $\neg\sigma_Z$ corresponds to the *negation* of symbol σ_Z;
- $\sigma_X \vee \sigma_Y$ corresponds to the *union* of symbols σ_X and σ_Y (i.e., the union of their respective *grounding sets*);
- $\sigma_X \wedge \sigma_Y$ corresponds to the *intersection* of symbols σ_X and σ_Y (i.e., of their respective *grounding sets*).

In classical planning, operators and symbols are used to describe actions in the following form

$$\alpha_i = (pre_i, eff_i^+, eff_i^-), \tag{5}$$

meaning that the action $\alpha_i \in \mathcal{A}$ can be executed when all the symbols $\{\sigma|\sigma \in pre_i\}$, also called *preconditions*, are *True*, and executing α_i produces the changes of the value of some symbols, also called *effects*, implying that all symbols $\{\sigma|\sigma \in eff_i^+\}$ assume the value *True* and symbols $\{\sigma|\sigma \in eff_i^-\}$ become *False*.

Finally, using symbols Σ and high-level actions $\mathcal{A}$ as building blocks, it is possible to define the model of environment $\mathcal{D}$, also called *domain*, and the *problem* $\mathcal{P}$ to solve. A classical planning domain can be defined as

$$\mathcal{D} = (\Sigma, \mathcal{A}, \Gamma), \tag{6}$$

using a set of symbols Σ, actions $\mathcal{A}$ and a *state-transition function* $\Gamma : \hat{\Sigma} \times \mathcal{A} \to \hat{\Sigma}$, where $\hat{\Sigma}$ is the set of possible subsets of Σ. The *state-transition* function $\Gamma(\hat{\Sigma}_s, \alpha_i) = (\hat{\Sigma}_s - eff_i^-) \cup eff_i^+$, if α_i is applicable to $\hat{\Sigma}_s$, where $\hat{\Sigma}_s$ is the set of symbols whose *grounding set* intersection defines the state $s \in S$. These elements are sufficient to describe the dynamics of the environment, over which the planner can reason and create chains of actions to reach a final goal. The function Γ encapsulates the transition model of the environment in each action model described by (5), defining the possible way to build sequences of them. Instead, the problem can be formalized as

$$\mathcal{P} = (\hat{\Sigma}_{s_{init}}, \hat{\Sigma}_{s_g}), \tag{7}$$

where $\hat{\Sigma}_{s_{init}}$ is the set of symbols whose *grounding set* intersection describes s_{init} and $\hat{\Sigma}_{s_g}$ the set of symbols whose *grounding set* intersection characterizes s_g. Then, the

plan solving the problem $\mathcal{P}$ is expressed as the sequence of actions

$$\omega = [\alpha_1, ..., \alpha_n], \tag{8}$$

resulting in a sequence of state transitions

$$[\hat{\Sigma}_{s_0}, \hat{\Sigma}_{s_1}, \ldots, \hat{\Sigma}_{s_n}], \tag{9}$$

such that $\Gamma(\hat{\Sigma}_{s_0}, \alpha_1) = \hat{\Sigma}_{s_1}, \Gamma(\hat{\Sigma}_{s_1}, \alpha_2) = \hat{\Sigma}_{s_2}, \ldots, \Gamma(\hat{\Sigma}_{s_{n-1}}, \alpha_n) = \hat{\Sigma}_{s_n}$ with initial state $\hat{\Sigma}_{s_0} = \hat{\Sigma}_{s_{init}}$ and final state $\hat{\Sigma}_{s_n} = \hat{\Sigma}_{s_g}$. In particular, the formulation presented based on a finite set of symbols Σ and state-transition system $\mathcal{T} = (\Sigma, \mathcal{A}, \Gamma)$ is called a *set-theoretic representation* [49]. In order to use classical planning systems, it is necessary to describe both the domain of the considered world and the tackled problem using a planning definition language. In this work we will use the Probabilistic Planning Domain Definition Language (PPDDL) [63]; once defined, both the domain and the problem definitions will be provided in input to the planner, which will eventually return a solution ω.

It is worth noting that the option o and the planning action α share some similarities. Indeed, o can be converted in its corresponding high-level action α finding the right set of symbols pre, eff^+, eff^- whose grounding sets satisfy the classifiers I_o as preconditions and β_o as effects, for the execution of π_o. The similarity between options and set-theoretic planning actions has been exploited to create automatic abstraction procedures able to convert the options' execution data into a working high-level planning description, thus allowing a solution that integrates ating low-level and high-level information. In this work we build upon the abstraction procedure implemented by Konidaris et al. [59] to convert sensors raw data into a PPDDL representation.

2.3 Intrinsic Motivation

The impulse to drive the agent away from the monotony of its usual activities, which psychologists and cognitive scientists have studied under the name of *intrinsic motivation*, is one of the most important elements enabling Open Ended Learning (OEL). The research in the field of Intrinsic Motivation (IM) concerns the study of human behaviors not influenced by external factors but characterized by internal stimuli (i.e. curiosity, exploration, novelty, surprise). In the case of artificial agents, we can summarize such aspect as anything that can drive the agent's behavior which is not directly dependent on its assigned task.

The insights provided by the IMs gave the researchers new ideas to model the stimuli of the agent (e.g. curiosity). Indeed, some models have been implemented using the prediction error (PE) in anticipating the effect of agent's actions (and more precisely the improvement of the prediction error [64, 65]) as an IM signal. A formal definition of agent driven by its curiosity has been formulated by Schmidhuber [64] as

simply maximizing its *future success* or *utility*, which is the conditional expectation

$$u(t) = E_\mu \left[\sum_{\tau=t+1}^{T} r(\tau) \middle| h(\leq t) \right], \tag{10}$$

over $t = 1, 2, \ldots, T$ time steps, receiving in input a vector $x(t)$, executing the action $y(t)$, returning the reward $r(t)$ at time t, taking into consideration the triplets $h(t) = [x(t), y(t), r(t)]$ as the previous data experienced until time step t (also called *history*). The conditional expection $E_\mu(\cdot|\cdot)$ assume an unknown probability distribution μ from M representing possible probabilistic reactions of the environment. To maximize (10), the agent also has to build a predictor $p(t)$ of the environment to anticipate the effects of its actions. The reward signal is defined as follows

$$r(t) = g(r_{ext}(t), r_{int}(t)), \tag{11}$$

which is a certain combination g of an external reward $r_{ext}(t)$ and an intrinsic reward $r_{int}(t)$. In particular, $r_{int}(t+1)$ is seen as *surprise* or *novelty* in assessing the improvements in the results of p at time $t+1$

$$r_{int}(t+1) = f|C(p(t), h(\leq t+1)), C(p(t+1), h(\leq t+1))|, \tag{12}$$

where $C(p, h)$ is a function evaluating the *performance* of p on a history h and f is a function combining its two parameters (e.g. in this case, it could be simply the improvement $f(a, b) = a - b$). It is important to notice that, as a baby does, an intrinsically motivated agent needs to find regularities in the environment to learn something. Consequently, if something does not present a *pattern*, there is nothing to learn and this becomes boring for both an agent and a human being.

In literature, IMs have also been categorized into different typologies [66–68]. An important discriminant aspect is the kind of signal received by the agent which can be of two types: *knowledge-based* (KB-IMs), which depends on the prediction model of the world (e.g. [64]), and *competence-based* (CB-IMs), which depends on the improvement of the agent's skills (e.g. [27]). In the framework presented in the next section, both these typologies are employed. CB-IM is used at a lower level to learn new skills and KB-IM at higher level to push the system to focus on the frontier of the visited states, from which it is more likely to discover novel information (e.g., find new states and learn new actions).

3 System Overview

This section presents a new framework of an open-ended learning agent which, starting from a set of action primitives, is able to (i) discover options, (ii) explore the environment using them, (iii) create a PPDDL representation of the collected knowledge and (iv) plan to improve its exploration while reaching a high-level objective set by the game.

The aim of this study is to assess the potential of *abstraction* in autonomous systems and propose a new approach for planning systems, extending them with learning capabilities and behaviors driven by IMs. IMs are employed for discovering new options in a *surprise-based* manner at low-level and continuously exploring new states driven by *curiosity* at high-level. By the term abstraction, we simply mean mapping a certain problem space into a simpler one (e.g. converting a continuous domain into a discrete domain). In the proposed architecture, the abstraction is applied at two levels: a) passing from the *primitive action space* to the *options action space* and b) converting *low-level data* collected during the exploration into a *high-level domain representation* suitable for high-level planning, thus from raw sensors' data to a PPDDL representation.

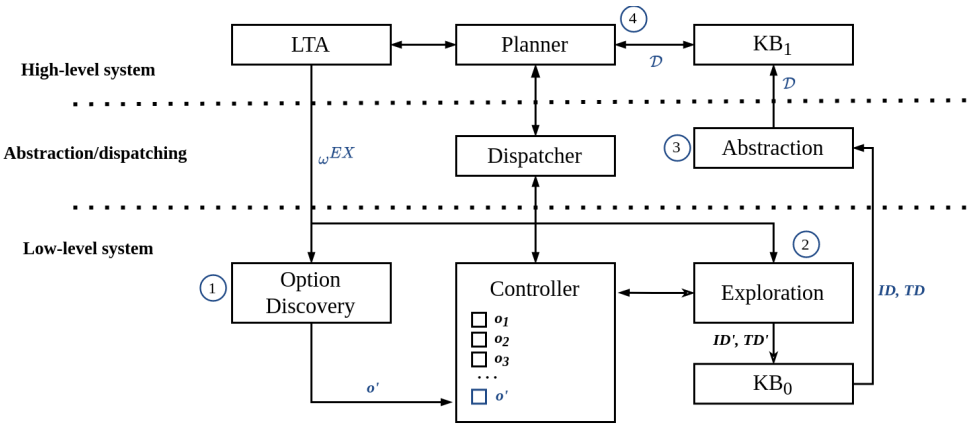


Fig. 2: The architecture of the system. The system, equipped with five primitive actions, iteratively (1) learns some options from scratch, (2) uses the options to explore the environment and gather low-level states data, (3) creates a PPDDL representation of the collected experience, (4) plan to solve the game and to improve its exploration.

Performing the pipeline depicted in Figure 2, the system creates different layers of abstraction, enriching the agent's knowledge with causal correlations between options and enabling more efficient reasoning (i.e. using classical planning). Symbols can be seen as knowledge building blocks that can be used to search for interesting states and find new knowledge in a virtuous loop.

3.1 Architecture description

As depicted in Figure 2, the system can be seen as a three-layered architecture: (i) the higher level contains the explicit agent's knowledge, (ii) the middle layer maps the high-level actions to their controllers and convert the raw sensors data into an explicit representation, and (iii) the lower level containing the components to sense the environment and interact with it. Mainly, the system executes the following pipeline:

1. **Option Discovery:** using a set of primitives $A = \{a_1, a_2, ...\}$ belonging to the agent, the system combines them to create higher level actions, in this case, the *options*;
2. **Exploration:** the set of options $O = \{o_1, o_2, ...\}$ discovered in the previous step is used to explore the environment. In the meantime, the visited low-level states data are collected into two datasets: the *initiation data* ID and *transition data* TD containing, respectively, the samples of states in which an option can be run and the transition between states executing it;
3. **Abstraction:** datasets ID and TD of the visited low-level states until that moment are processed by the algorithm of abstraction, generating a PPDDL domain $\mathcal{D}$ of the knowledge collected;
4. **Planning:** the PPDDL representation $\mathcal{D}$ is used to assess whether the final goal of the task s_g can be reached with the currently synthesized knowledge, and to generate a plan ω^{EX} to explore interesting areas of the domain. This plan is suggested by the **Goal Selector**, function included in the Long-Term Autonomy (LTA) module.
5. The system, coordinated by the LTA, will execute again the loop from step 1, exploiting ω^{EX} to improve the Option Discovery and Exploration phases.

Algorithm 1 Discover-Plan-Act algorithm

procedure DISCOVER_PLAN_ACT($cycles, dpa\_eps, dpa\_steps, d\_eps, d\_steps$)
  $c \leftarrow 0$ //Cycle initialization
  $O \leftarrow \{\}$ //Option set initialization
  $ID \leftarrow \{\}$ //Initiation Data initialization
  $TD \leftarrow \{\}$ //Transition Data initialization
  $\omega^{EX} \leftarrow \{\}$ //Initially, the high-level plan is empty
  while $c < cycles$ **do** //For each cycle
    $O\_new \leftarrow DISCOVER(d\_eps, d\_steps, \omega^{EX})$ //Learning the available options
    $O \leftarrow O \cup O\_new$
    $ID_{new}, TD_{new} \leftarrow Collect\_Data(dpa\_eps, dpa\_steps, O, \omega^{EX})$
    $ID \leftarrow ID \cup ID_{new}$
    $TD \leftarrow TD \cup TD_{new}$
    $\mathcal{D} \leftarrow Create\_PPDDL(ID, TD)$
    $s_{target} \leftarrow Get\_Target\_State()$
    $\mathcal{P}_{target} \leftarrow Generate\_PPDDL\_Problem(s_{target})$
    $\omega^{EX} \leftarrow Plan(\mathcal{D}, \mathcal{P}_{target})$
    $Check\_PPDDL\_Validity(\mathcal{D})$
    $c \leftarrow c + 1$
  end while
end procedure

In this setting, the agent is initially only endowed with a set of primitive movements $A = \{a_0, ..., a_m\}$, and the world $s \in S$ is represented in terms of a vector $(v_0, ..., v_n)$ of low-level variables $v_i \in \mathcal{R}$, whose values as retrieved by the agent's sensors.

The iterative utilization of this framework allows the synthesis of an emerging abstract representation of the world from the raw data collected by the agent, which continuously undergoes a refinement process over time, as it gets enriched with new

actions and symbolic concepts. In the following subsections, all the process funcionalities will be individually explained, with reference to the pseudocode depicted in Algorithm 1.

3.1.1 Option Discovery

In this section, we will analyze the **Option Discovery** module (Algorithm 1, line 8) in greater detail. As previously anticipated, the discovery of new options is considered to be driven by the agent's *surprise* in finding out that new primitives are available for execution, during the agent's operations. When the agent encounters a change in the availability of its primitive abilities, it stores this event as a low-level skill that can be re-used later to explore the surrounding environment.

By executing the algorithm, the agent can discover a set of options O from scratch; such options are generated by repeatedly executing a certain primitive $a \in A$ among the available ones and collecting the produced changes in the environment. This procedure is intentionally implemented in a simplified way, given that the focus of this work is on the architecture for extensible symbolic knowledge to be reusable to reach intrinsic and extrinsic goals autonomously; more sophisticated stategies to discover new policies are left to future works.

In more detail, the agent creates new options considering the following modified definition of option:

$$o(a^p, a^t, I, \pi, \beta), \tag{13}$$

where a^p and a^t are primitive actions such that: (i) $a^p \neq a^t$, a^p is used by the execution of π, (ii) a^t stops the execution of π when it becomes available, (iii) π is the policy applied by the option, consisting in repeatedly executing a^p until it can no longer be executed or a^t becomes available, (iv) I is the set of states from which a^p can run; and (v) β is the termination condition of the option, corresponding to the availability of the primitive a^t or to the impossibility of further executing a^p. For the sake of simplicity, in the remainder of the paper the option's definition will follow the more compact syntax

$$o(a^p, a^t) \tag{14}$$

meaning that I is the set of states in which a^p can run, β checks the following two conditions: a^t becomes available or a^p is no longer available, and π is the policy corresponding to repeatedly executing a^p until β verifies.

Algorithm 2 describes in further details the option discovery procedure previously described. At the beginning of each discovery episode, the plan ω^{EX} is executed to reach a new area where to start learning new options (line 7-9). Then, for a maximum number of episodes and steps, the agent saves the current state s and randomly selects a primitive a^p which can be executed in s (line 10-12). a^p is repeatedly executed towards reaching the state s' until either a^p is no longer available or new primitives beyond a^p become available. If $s \neq s'$, the procedure creates a new option o where $o = o(a^p, a^t)$ if a new primitive a^t has become available, or $o = o(a^p, \{\})$ in the opposite case. In either way, in case o has not been discovered before, it is added to the other collected options. It is important to note that the options are independent on the state where the agent is, and are defined by the primitives' availability. This definition makes options reusable on different floors and with different objects, just

depending on the agent's abilities. The procedure can discover a subset of the options available in the original implementation[1] sufficient to solve the entire game problem.

Algorithm 2 Option Discovery

```
1: procedure OPTION_DISCOVERY(d_eps, d_steps, ω^EX)
2:     O_new ← {}
3:     ep ← 0
4:     while ep < d_eps do //For each episode
5:         T ← 0
6:         Reset_Game()
7:         for option in ω^EX do // Execute IM plan
8:             Execute(option)
9:         end for
10:        while T < d_steps do //For each step
11:            s ← Get_State()
12:            a^p ← Get_Available_Primitive()
13:            while Is_Available(a^p) and not (New_Available_Prim()) do
14:                Execute(a^p)
15:                s' ← Get_State()
16:            end while
17:            if s ≠ s' then
18:                if New_Available_Prim() then
19:                    a^t ← Get_New_Available_Prim()
20:                    o ← Create_New_Option(a^p, a^t)
21:                else
22:                    o ← Create_New_Option(a^p, {})
23:                end if
24:                O_new ← O_new ∪ o
25:            end if
26:            T ← T + 1 //End For each step
27:        end while
28:        ep ← ep + 1 //End For each episode
29:    end while
30:    return O_new
31: end procedure
```

3.1.2 Exploration

After discovering a set of valid options O as explained in the previous section, the system exploits them to explore the environment, collecting data about the reached low-level states (Algorithm 1, line 10). Considering that the abstracted representation of the world does not change significantly with a small amount of new data, the function *Collect_Data()* is in charge of executing *d_steps* options for *d_eps* episodes, in which the agent starts its exploration from the initial configuration of the environment.

At each timestep, the agent attempts to perform an option $o \in O$ from a certain low-level state $s \in S$. The selection of the action o during the exploration can follow different strategies, which are described in the subsection 3.1.4. In case the execution of

[1] Link to the original implementation of [59]: https://github.com/sd-james/skills-to-symbols

the option changes the low-level variables of the state s and, consequently, the *mask*[2] m is not null, the system registers two types of data tuple (Algorithm 1, line 10): the *initiation data* tuple id and the *transition data* tuple td. The multiple instances of these tuples are stored, respectively, in the datasets ID, for the initiation data, and TD, for transition data. A single *initiation data* tuple id_i has the following structure

$$id_i = (s, o, f(s, o)), \tag{15}$$

where the function $f(s, o)$ returns the feasibility of executing o from s ($True$ if $s \in I_o$ and $False$ otherwise). The *transition data* tuple td_j takes the following structure

$$td_i = (s, o, r, s', g, m, O'), \tag{16}$$

where s' is the state reached after executing option o from the state s, g is a flag stating whether the final objective of the task has been reached, m is the *mask* of the option and O' is a list defining the options that can be executed from s'. When all the steps of the episode have been executed, the environment is reset and the next episode is started until reaching the maximum number of allowed episodes d_eps, where the $Collect\_Data()$ procedure terminates and the stored data are added to the existing datasets ID and TD (line 11-12).

3.1.3 Abstraction

The datasets collected in the previous step are then used as input for the function $Create\_PPDDL()$ (Algorithm 1, line 13), returning a symbolic representation $\mathcal{D}$ of the agent's current knowledge expressed in PPDDL formalism (PPDDL domain). The main advantage of the obtained PPDDL representation is that it makes explicit the causal correlations between operators that would have remained implicit at the option level. In the following, we provide a summary description of the abstraction procedure; for further details, the reader is referred to the original article [59].

The abstraction procedure executes the following five steps:

1. **Options partition**: this step is dedicated to partitioning the learned options into *abstract subgoal options*[3], a necessary assumption of the abstraction procedure. Abstract subgoal options are characterized by a single precondition and effect set. However, given the uncertainty of the actions' effects in the environment, the operators' effects will be modelled as *mixture distributions* over states. This phase utilizes the transition dataset TD collected before, as it captures the information about the domain segment the option modifies. Basically, the transition dataset is divided into sets of transition tuples presenting the same option o and mask m. Then, for each set, the partitions are ultimately obtained by performing clustering on the final states s' through the DBSCAN algorithm [69]. If some clusters overlap in their

[2]The *mask* is the list of all the state variables that are changed by the execution of a specific option. See details in [59].

[3]An abstract subgoal option o is characterized by a list of indices of the low-level variables, called *mask*, which are changed with the execution of o, without modifying other variables. In addition, the changing variables' values do not depend on their initial value.

initiation set, a unique partition is created with different effects and occurrence probabilities.

2. **Precondition estimation**: this step is dedicated to learning the classifiers that will identify the *preconditions* of the PPDDL operators. In order to have negative examples of the initiation set classifier for each operator, this operation utilizes the initiation dataset ID considering all the samples with option o and $f(s, o) = False$. The positive examples comprise instead the initial states s taken from TD tuples belonging to the same partition. The initiation set classifier of the option is computed using the Support Vector Machines (SVM) [70]. The output of this phase is the set of all the *initiation set* classifiers of all the operators.
3. **Effect estimation**: analogously, this step is dedicated to learning the symbols that will constitute the *effects* of the PPDDL operators. The effects distribution is modelled through the Kernel density estimation [71, 72], taking in input the final states s' of each partition.
4. **PPDDL Domain synthesis**: finally, this step is dedicated to synthesising the PPDDL domain, characterized by the complete definition of all the operators associated with the learned options in terms of preconditions and effect symbols. This step entails the simple mapping of all the data structures generated during the previous steps in terms of symbolic predicates to be used as preconditions and/or effects for every operator.

The produced PPDDL domain can be potentially used to reach any subgoal that can be expressed in terms of the available generated symbols at any point during the Discovery-Plan-Act (DPA) loop. One interesting aspect of the proposed DPA framework is that the semantic precision of the abstract representation and its expressiveness increase as the DPA cycles proceed, as will be described in the experimental section.

3.1.4 Goal Selector

This module aims at simulating the *intrinsic motivations* driving the agent towards interesting areas to satisfy its *curiosity* and optimize its exploration. In particular, one of the most fascinating aspects of this system is the capability of setting its high-level goals, which potentially could be a combination of symbols defining a state that the agent has never experienced before. In other words, abstract reasoning could be the driving criterion for *using the imagination to explore an unknown environment*. Despite the previous goal is rather ambitious and still the object of future work, we will demonstrate in this work that the abstract reasoning can indeed be used for the more "down-to-earth" task of devising rational criteria to make more efficient the exploration of unexplored parts of the environment.

The selection of the target state $s_{target} \in S$ to be reached in the next exploration cycle is performed at line 14 of Algorithm 1, calling the procedure $Get\_Target\_State()$. The *Goal Selector* suggests such state to the system, following an internal strategy which can be, in this case, *Action Babbling*, *Goal Babbling* and *Distance-based Goal Babbling*.

Action Babbling is the simplest strategy of the system for the exploration, consisting of a pure random walking of the agent. This strategy returns $s_{target} = NULL$, so

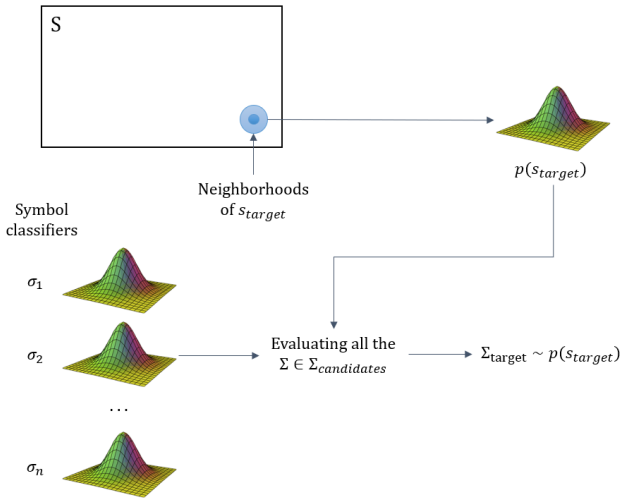


Fig. 3: The conceptual depiction of the selection of the symbols necessary to represent the goal s_{target}. The set of symbols having the distribution most similar to the goal will define the goal of the PPDDL problem file.

no plan ω^{EX} is generated and executed in the exploration phase on the subsequent cycle. *Goal Babbling* and *Distance-based Goal Babbling* differ in the way they implement IMs (such as curiosity). More specifically, in our system curiosity is formalised as an interest to reach (i) a random goal among those already achieved, and (ii) the border of the already acquired knowledge.

The first strategy is represented by *Goal Babbling*, consisting in randomly selecting a low-level state in the environment and trying to reach it [73]. Usually, the assumption of *Goal Babbling* is that all the goals which can be set belong to the world's low-level states that are reachable; each goal is formalised as the configuration of joints or position to be reached with the robot's actuators [27]. Since in general not all the states $s \in S$ are valid (e.g. the agent can't move inside the wall), this strategy selects a random state s_{target} among the visited ones (line 14). Subsequently, s_{target} is translated into a set of propositional symbols $\{\sigma_1, ..., \sigma_k\}$, as described in the following subsection 3.1.5, which represent the high-level goal to be reached using an off-the-shelf PPDDL planner. The capability of translating low-level states into symbols gives the agent a chance to reason on causal dependencies and, consequently, plan. It is important to notice that a pure *Goal Propositional Babbling*, consisting of the selection of a random subset of high-level symbols, would not be an effective strategy because only a limited number of combinations of symbols conjunctions are valid goals. Consequently, *Goal Propositional Babbling* is not taken into consideration.

The second strategy (*Distance-based Goal Babbling*) is implemented as a modified version of the *Goal Babbling*, and models the curiosity towards the less explored states as being influenced by the goal's distance from its starting location s_{init}. In this case,

the curiosity level for a state is defined as

$$\eta(s) = \|s_{init} - s + \mathcal{Z}\|, \tag{17}$$

consisting of the norm of the difference between the state vector of the agent at the beginning of each episode s_{init} and the visited state s. The low-level s_{target} state is selected between the farthest visited states, as follows:

$$s_{target} = \max_{s} \eta(s), \forall s \in S_{visited}. \tag{18}$$

More precisely, the low-level state s, target of the exploration, is the state that maximizes the distance $\eta(s)$. In the computation of s_{target}, a Gaussian noise $\mathcal{Z} \sim \mathcal{N}(0, 1)$ is added, to facilitate the reaching of different states. The *Goal Selector* driven by η continuously pushes the agent towards the border of the already acquired knowledge (similarly to the idea presented in [74, 75], but with a different implementation) and is thus the main responsible for the knowledge increase of the agent. Moving towards the frontier states, the agent is more likely to encounter novel states thus maximizing the probability to acquire new symbolic knowledge, which can in its turn be used to synthesize new (e.g., more expressive) high-level goals to be eventually reached through planning, ultimately creating a virtuous loop in which the agent's capabilities of increasing its knowledge through environment exploration are iteratively enhanced.

3.1.5 Translating low-level states into symbols

The peculiarity and, simultaneously, the biggest challenge of this software architecture is thus to use an abstract symbolic representation to manage the evolution of the agent's knowledge. Using a symbolic representation to describe a desired environment configuration is a powerful tool for an efficient exploration of the world.

After selecting s_{target} (line 14), the purpose of the planning module is twofold. On the one hand, it can be employed as usual to verify the reachability of the goal s_g of the environment (i.e. get the treasure and bring it "home" in the Treasure Game, in line 17) and, on the other hand, it can be exploited to generate a plan driving the agent towards states, in our case s_{target}, relevant to extend the knowledge of the system (line 16). In both cases, to take advantage of planning it is necessary to transform a low-level state into a high-level one. This operation requires finding the combination of propositional symbols

$$\hat{\Sigma}_{target} = \{\sigma_1, ..., \sigma_k\} \tag{19}$$

that best represent the portion of state space including s_{target}. In other words, we look for a subset of symbols $\hat{\Sigma}_{target}$ whose *grounding* is s_{target}. These symbols make it possible to generate the definition of a planning problem (line 15) which, together with the domain definition, can be used to perform planning and solve the problem (line 16).

In order to select the right symbols conjunction, the system creates the classifier $Cl_{target} \sim p(s_{target})$ approximating a distribution over s_{target}. Cl_{target} is a SVM

classifier trained on the states

$$\{s| \, \|s_{target} - s\| \leq \epsilon\}, \forall s \in S_{visited}, \tag{20}$$

representing, as positive samples, all the neighbours of s_{target} within a maximal distance ϵ and, as negative samples, all the remaining states encountered in the agent's experience. Once the classifier is generated, all the propositional symbols whose *mask* is equal to a *factor*[4] contained by Cl_{target} are collected as candidates $\hat{\Sigma}_{candidates}$. Then, all the subsets of symbols $\hat{\Sigma}_i \subset \hat{\Sigma}_{candidates}$, *whose respective masks do not overlap* are evaluated as representations of s_{target}. From each $\hat{\Sigma}_i$, m state samples $S_{\Sigma_i} = \{s_1, ..., s_m\}$ are generated and the score of the subset $\hat{\Sigma}_i$ is calculated as

$$score(\Sigma_i) = \frac{1}{m} \sum_{s \in S_\Sigma} Cl_{target}(s) \tag{21}$$

where $Cl_{target}(s)$ returns the probability that s belongs to the positive class of the classifier Cl_{target}. Then, the subset of symbols $\hat{\Sigma}$ maximizing the *score* function is used as a goal in the problem definition $\mathcal{P}_{target}$.

3.1.6 Planning

At the end of each cycle, the planning process generates a plan to reach either s_{target} and s_g. In both cases, in order to create a PDDL problem $\mathcal{P}$, it is necessary to find the set of symbols $\hat{\Sigma}_{init}, \hat{\Sigma}_g$ and $\hat{\Sigma}_{target}$, describing the most suitable high-level state representation for s_{init}, s_g and s_{target} respectively, as described in the previous subsection 3.1.5. Indeed, the couples $(\hat{\Sigma}_{init}, \hat{\Sigma}_g)$ and $(\hat{\Sigma}_{init}, \hat{\Sigma}_{target})$ define the problems $\mathcal{P}_g$ and $\mathcal{P}_{target}$. At line 15 of Algorithm 1, $\mathcal{P}_{target}$ is generated as previously discussed and the plan to solve it, ω^{EX}, is generated by the planner (line 16).

Before moving to the next cycle, the system tries to solve also the problem $\mathcal{P}_g$, performing the function $Check\_PPDDL\_Validity$ (line 17). The resulting plan ω^g is only used internally by the system to keep track of the success ratio of the planner with the evolution of the synthesized knowledge of the agent.

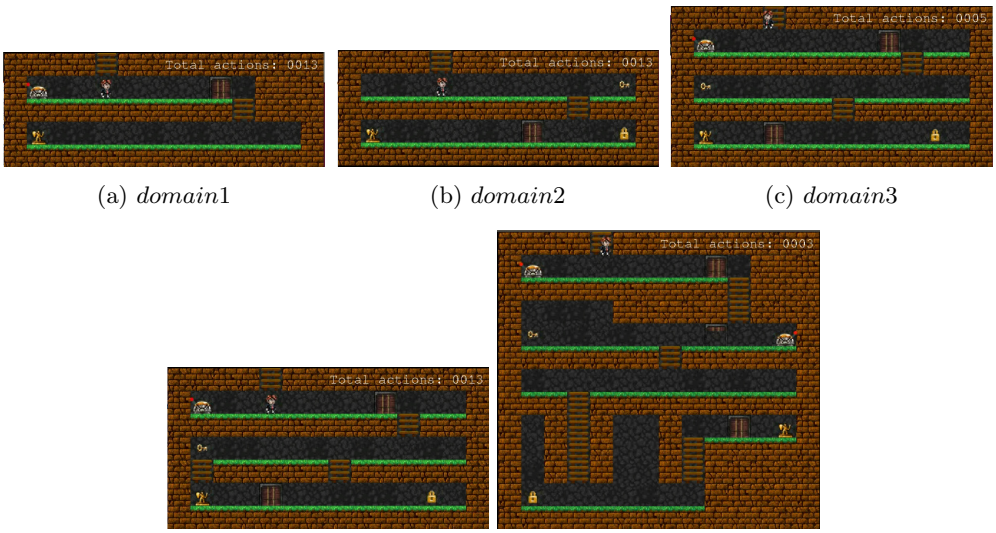
4 Experiment and Results

This section, dedicated to the experimental analysis, will first describe the dynamics of the environment, followed by an example of the system's cycle execution with its outputs and, finally, the overall results collected over different environment configurations.

4.1 Environment setup

The implemented system has been tested in the so-called Treasure Game domain [59]. In such an environment, an agent can explore the maze-like space by moving through

[4] *"Sets of low-level state variables that, if changed by an option execution, are always changed simultaneously" [59].*



(a) *domain*1 (b) *domain*2 (c) *domain*3

(d) *domain*4 (e) *domain*5

Fig. 4: All the domain configurations used in the experimental analysis. The game's purpose is to get the treasure at the bottom of the maze and bring it back to the top ladder.

corridors and doors, climbing stairs, interacting with handles (necessary to open/close the doors), bolts, keys (necessary to unlock the bolts) and a treasure. The agent starts its activity from the ladder on top of the maze (home location) and its overall task is to find the treasure and bring it back to the starting location. In our experimentation, the agent starts endowed with no previous knowledge about the possible actions that can be executed in the environment; the agent is only aware of the basic motion primitives at its disposal $A = \{go\_up, go\_down, go\_left, go\_right, interact\}$, respectively used to move the agent up, down, left or right by 2-4 pixels (the exact value is randomly selected with a uniform distribution) and to interact with the closest object. The interaction with a lever changes the state (open/close) of the doors associated with that lever (both on the same floor or on different floors) while the interaction with the key and/or the treasure simply collects the key and/or the treasure inside the agent's bag (in the bottom-right corner of the screen). The interaction with the bolt opens the door next to the treasure and it is feasible only when the agent has the key in its bag. The state $s \in S$ is defined in terms of the following low-level variables:

$$s = (x_{agent}, y_{agent}, \theta_1, \theta_2, ..., x_{key}, y_{key}, x_{bolt}, x_{treasure}, y_{treasure}) \tag{22}$$

in which x_{agent}, y_{agent} is the (x,y) position of the agent, θ_i is the angle of the lever i, x_{key}, y_{key} is the (x,y) location of the key, x_{bolt} is the state of the bolt (1 if open and 0 if locked) and $x_{treasure}, y_{treasure}$ is the (x,y) location of the treasure.

The system is tested in five different configurations of the environment, of increasing complexity: two small-sized (Figure 4a and 4b), two medium-sized (Figure 4c and 4d) and one complete instance (Figure 4e). For example, in the setting depicted in Figure 4e the agent has to: (i) pull the two levers on the top of the maze to open the doors, (ii) get the key which is used to (iii) unlock the bolt, (iv) get the treasure and (v) bring it back on top of the environment. The obstacles that pertain to the other configurations are shown in Table 1.

Domain	Levers	Keys	Bolts	Notes
*domain*1	1	0	0	None.
*domain*2	0	1	1	None.
*domain*3	1	1	1	None.
*domain*4	1	1	1	Shortcut available.
*domain*5	3	1	1	2 levers going to the treasure, 1 going home.

Table 1: Obstacles to be solved to end the game.

4.2 The cycle

For exemplificatory purposes, a complete execution cycle of the system is briefly described in the following, showing the output of each phase of an intermediate cycle (cycle n. 10) performed on *domain*3 (see Figure 4c) in the *Goal Babbling* strategy case.

Option Generation

At the beginning of each cycle, the agent executes Algorithm 2 to collect some options exploiting the agent's primitives, before the exploration can commence. In the Treasure Game environment selected for this work, the agent executes $d\_eps = 1$ episodes, composed by $d\_steps = 200$ primitive actions. After the execution of the plan ω^{EX}, the random exploration is reprised using the primitives contained in A (see Section 2.1). The result is the following set of learned options (11 in total) following the formalization (14):

```
O = { (go_up,{}), (go_down,{}), (go_left,{}), (go_left,go_up),
      (go_left, go_down), (go_left,interact), (go_right,{}),
      (go_right,go_up), (go_right,go_down), (go_right,interact),
      (interact,{}) }.
```

It is important to note that, in general, the discovered options are not all the options that may be possibly discovered in the environment, but only those experienced by the agent during the exploration. This procedure is incremental, adding options to the set O each iteration of the Algorithm 1. As described in section 3.1.1, this procedure leverages IMs *at low level*, capturing the curiosity of the agent when it discovers to have new available primitives to exploit.

```
(define (domain TreasureGame)
    (:requirements :strips :probabilistic-effects :rewards)

    (:predicates
        (notfailed)
        (symbol_0)
        (symbol_1)
        (symbol_2)
        ...
        (symbol_25)
    )

    (:action option-0-partition-0-0
        :parameters ()
        :precondition (and (notfailed) (symbol_5) (symbol_22)
                    (symbol_14))
        :effect (and (symbol_6) (not (symbol_5)) (decrease
                (reward) 36.00))
    )

    ...

     (:action option-10-partition-3-715
        :parameters ()
        :precondition (and (notfailed) (symbol_20) (symbol_22)
                    (symbol_14) (symbol_15) (symbol_0))
        :effect (and (symbol_13) (not (symbol_20)) (decrease
                (reward) 90.00))
    )
)
```

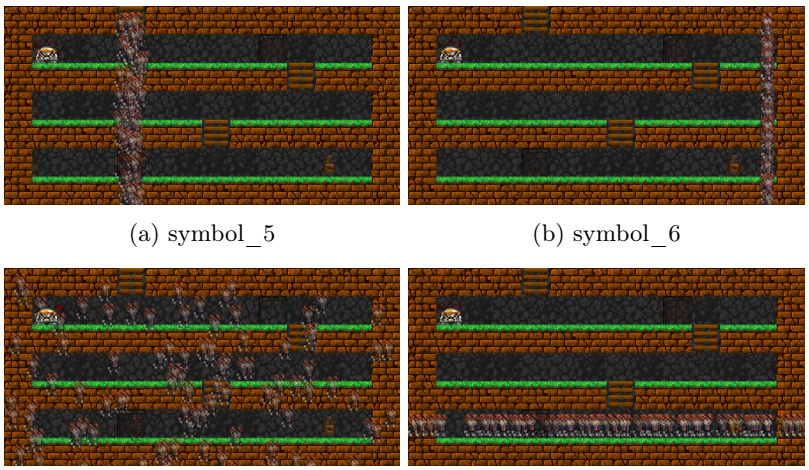
Fig. 5: An extract of the PPDDL generated by the abstraction procedure. Notice that `option-0-partition-0-0` can be explained by looking at the symbols of Figure 6. In fact, the effect is to change the x position replacing *symbol*_5 with *symbol*_6, maintaining invariant the presence of *symbol*_14 and *symbol*_22. Consequently, it is the option moving the agent on the right on the last floor.

Exploration

In this step, the plan ω^{EX} is again executed before starting the random walk over the available options O. In this particular example, 600 data entries have been collected in the ID dataset and 6600 in the TD dataset. It is important to remember that the number of collected data over the cycles is not constant because when an option does not produce effects on the environment, no data is collected.

Abstraction

Figure 5 shows a part of the output of the PPDDL domain obtained from the abstraction procedure. As visible, the figure does present a valid PPDDL domain description, upon which the planner may reason. Moreover, any subset of the produced domain predicates (symbols) can be used to define high-level goals the planner may try to plan for. In this specific case, the system generated 26 symbols and 716 operators.



(a) symbol_5 (b) symbol_6

(c) symbol_14 (d) symbol_22

Fig. 6: Semantics of the symbols employed by the first operator of the generated PPDDL: (a) and (b) mean being in a certain x position, (c) mean the lever is pulled and (d) being on the bottom floor.

Planning

The produced PPDDL domain is used by the system to (1) solve the entire game, or task, and (2) improve the exploration. Figure 7 illustrates two PDDL problems: the problem on top refers to the general goal of solving the game ($\mathcal{P}_g$) while the problem at the bottom refers to the goal dynamically synthesized by the *Goal Selector* module, which in this example relates to reaching the left corner of the middle floor ($\mathcal{P}_{target}$). The solution for both problems is presented in Figure 8. The first solution, ω^g, solves the game problem in 21 moves and the second one, ω^{EX}, reaches the *Goal Selector* goal in 6 moves.

4.3 Results

In this section, the overall results of the system over different domain instances are described. First, the setting of the environment is discussed, then the adopted baseline, as well as the other strategies enabling the planning exploration. Subsequently, some charts are presented, that highlight the system's performance in terms of success ratio on the planning task. Finally, some issues worth being underscored are commented, and the limitations of the employed technologies are discussed.

For our purposes, the Treasure Game[5] has been used with five different mazes (see Figure 4), to focus on the performances of the system on smaller domains, highlighting pros and cons of the symbolic approach proposed. The system has been executed, following the workflow described in Algorithm 1, in the cited five mazes configurations using parameters suitable to solve the tasks, which are described later. To summarize,

[5]Github repository: https://github.com/sd-james/gym-treasure-game

```
(define (problem task_goal)
    (:domain TreasureGame)

    (:init (notfailed) (symbol_0) (symbol_1)
           (symbol_2) (symbol_3)
           (symbol_4) (symbol_5) )

    (:goal (and (notfailed) (symbol_4)
           (symbol_17) (symbol_18)) )
)


(define (problem im_goal)
    (:domain TreasureGame)

    (:init (notfailed) (symbol_0)
           (symbol_1) (symbol_2)
           (symbol_3) (symbol_4)
           (symbol_5) )

    (:goal (and (symbol_13) (symbol_24)
           (symbol_14) (symbol_2)
           (notfailed)) )
)
```

Fig. 7: On the top, the problem of solving the task is synthesized by the system $\mathcal{P}_g$ and, on the bottom, the problem generated by the *Goal Selector* module $\mathcal{P}_{target}$.

the system iteratively (i) looks for new options O_{new} performing d_steps primitives for d_eps episodes, (ii) explores for dpa_eps episodes the maze executing dpa_steps, (iii) creates a symbolic abstraction of the domain $\mathcal{D}$, (iv) creates a plan ω^{EX} to optimize the exploration of the successive cycle $c + 1$. It is important to note that the autonomously discovered options successfully produced a valid high-level model, usable to build correct plans. To assess the accuracy of the current abstraction $\mathcal{D}$, we evaluate its ability to reach the goal s_g at the end of each cycle. This is done by requesting the planner to solve the problem $\mathcal{P}_g$ using the currently produced symbolic domain description $\mathcal{D}$.

The first strategy used in the experiments is the random walk, which is called *Action Babbling* [50] (see Section 3.1.4). This strategy simply executes the Algorithm 1 without using planning because the *Goal Selector* returns $s_{target} = NULL$ and no plan ω^{EX} is generated. This simple strategy is used as a baseline for the experimental analysis, and it is necessary to fully appreciate the advantages of using the symbolic approach. The exploration is equivalent to the one used by Konidaris [59], and we use it to observe the development of the symbolic model over time.

To support the use of symbolic planning, two other strategies have been considered: *Goal Babbling* and *Distance-based Goal Babbling* (see Section 3.1.4). These three strategies could be seen as having an increasing complexity and effectiveness in the exploration:

- *Action Babbling* selects completely random actions;
- *Goal Babbling* selects random goals, reaching them with a planned sequence of actions;

```
PLAN TASK GOAL:

[ 1:(go_down,{}),           ; climb down the stairs
  2:(go_left,go_down),      ; go left until it can go down
  3:(interact,{}),          ; pull the lever
  4:(go_right,go_down),     ; go right up to the stairs to go down
  5:(go_down,{}),           ; climb down the stairs
  6:(go_right,{}),          ; go right up to the end of the corridor
  7:(go_left,{}),           ; go left up to the end of the corridor
  8:(interact,{}),          ; take the key
  9:(go_right,go_down),     ; go right up to the stairs
  10:(go_down,{}),          ; climb down the stairs
  11:(go_right,interact),   ; go right up to the bolt
  12:(interact,{}),         ; unlock the bolt
  13:(go_left,go_down),     ; go left until it is possibile
  14:(interact,{}),         ; get the treasure
  15:(go_right,go_up),      ; go right up to the stairs
  16:(go_up,{}),            ; go upstairs
  17:(go_right,{}),         ; go right up to the end of the corridor
  18:(go_left,go_up),       ; go left up to the stairs
  19:(go_up,{}),            ; go upstairs
  20:(go_left,go_up),       ; go left up to the stairs
  21:(go_up,{}) ]           ; go up to home

PLAN IM GOAL:

[ 1:(go_down,{}),           ; climb down the stairs
  2:(go_left,go_down),      ; go left up to the lever
  3:(interact,{}),          ; pull the lever
  4:(go_right,go_down),     ; go right until it can go down
  5:(go_down,{}),           ; climb down the stairs
  6:(go_left,interact) ]    ; go left up to the key
```

Fig. 8: The plans generated ω^g and ω^{EX}.

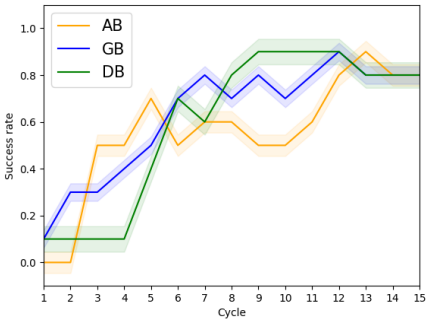
- *Distance-based Goal Babbling* selects the farthest goals and reaches them by using planning (see subsection 3.1.4).

It is reasonable to conjecture that in smaller domains where the reasoning is less necessary and purely random exploration may suffice, we do not expect to observe much difference among the previous strategies; while in the bigger domains the time to reach the goal may significantly change, depending on the strategy employed.

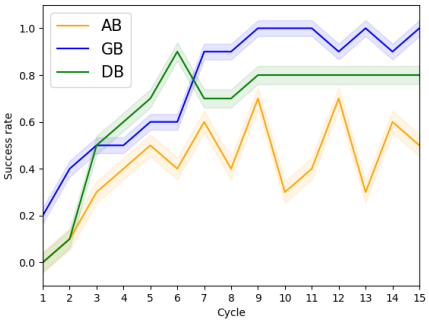
The five configurations considered are depicted in Figure 4, and the parameter values used in the exploration function $Collect\_Data$ for each configuration are shown in Table 2. Smaller domains $domain1$ and $domain2$ required the execution of 50 options per episode, $domain3$ required 150, $domain4$ required 200, and finally 800 options were executed in $domain5$.

The main results of the system are depicted in Figure 9. Precisely, in the graphs, it is shown the probability of success in solving the game over time using different strategies. Such success entails the generation of a sufficiently mature domain $\mathcal{D}$ and a correct problem formalization of $\mathcal{P}_g$, resulting in a correct plan ω^g.

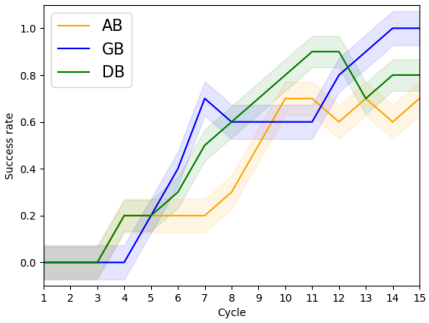
The system has been run with $cycles = 15$, assessing at the end of each cycle whether ω^g was able to solve the game using the current synthesized PPDDL domain representation $\mathcal{D}$. This mechanism has been performed for ten trials. Consequently, in



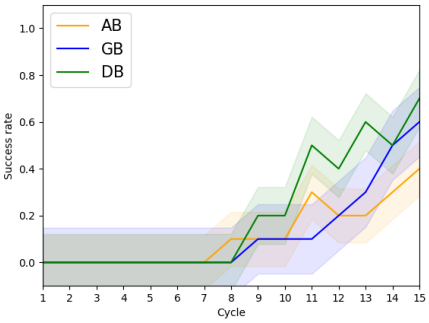
(a) domain 1



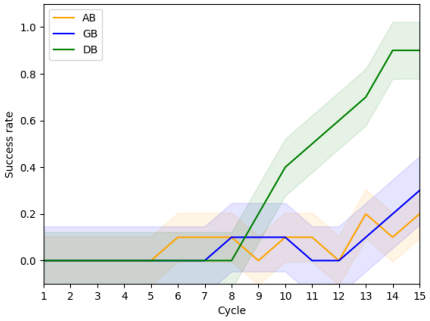
(b) domain 2



(c) domain 3



(d) domain 4



(e) domain 5

Fig. 9: The figure depicts the success rate of ω^g using different exploration strategies over time. In figure 9a, 9b, 9c, 9d and 9e, we have respectively the results of the domain of figures 4a, 4b, 4c, 4d, 4e over 15 cycles of exploration.

Domain	*dpa_eps*	*dpa_steps*	minimal solution steps
*domain*1	4	50	11
*domain*2	4	50	13
*domain*3	4	150	19
*domain*4	4	200	15
*domain*5	4	800	31

Table 2: Main parameters used in the different environment configurations.

the charts of Figure 9 are depicted cycles on the *x-axis*, and the percentage of trials that have succeeded in solving the game in the specific cycle on the *y-axis*.

In the smaller domains *domain*1, *domain*2, *domain*3 and *domain*4, the planning contribution is limited and sometimes not visible at all. Especially in the simplest domain *domain*1 the three strategies present similar performances, solving in the last cycles 80-90% of the trials (Figure 9a). In domains *domain*2 and *domain*3, presenting slightly higher complexities, the advantages of executing plans start being visible (Figure 9b and 9c). It is evident that *Goal Babbling* behaves almost perfectly in the last cycles, demonstrating the usefulness of collecting transition data with reasonable sequences of actions. The combination of randomness in selecting the goal to be reached and the reasoned transitions speed up the exploration and the consequent maturity of the PPDDL representation of the environment. Instead, the *Distance-based Goal Babbling* seems better than a completely random search but less effective than the precedent. This result entails that applying an exploration focused on the frontier's goals is not convenient in smaller domains. In fact, in the conclusive cycles of *domain*2 and *domain*3, *Goal Babbling* maintains 90-100% of success ratio, *Distance-based Goal Babbling* around 80% and *Action Babbling* between 60-70%.

Although the problem faced by *domain*3 seems easier than *domain*4, presenting respectively minimal solutions of 19 and 15 steps (see Table 2), the system needs more steps per episode to solve *domain*4 using the same amount of episodes used by *domain*3. The reason of this behaviour is due to the higher complexity of synthesizing a PPDDL domain $\mathcal{D}$ for this scenario. In fact, the scenario *domain*4 presents a sort of "shortcut" to reach the treasure, which does not require collecting the key and opening the door. From the point of view of the abstraction procedure, the shortcut represents a branch in the possible choices of the agent, thus presenting the agent with additional concepts to be abstracted in order to synthesize a complete representation. Consequently, the system generates additional symbols and operators, requiring more experience to strengthen the transition model captured by the PPDDL and provide satisfying plans.

The significance of the frontier exploration emerges in the *domain*5 configuration, where planning evidently boosts the results of the agent. Indeed, being bigger than other domains, *domain*5 is more difficult to be solved and planning results to be efficient in driving the exploration of the agent immediately towards the borders of its knowledge. The main result highlighted by the charts is that in bigger domains (Figure 9e), the impact of using planning is evident. Planning is able to easily drive the agent towards interesting visited states, where it can continue exploring. Statistically, after collecting the transition data for 15 cycles, the system struggles to solve the problem,

never exceeding 10% of success. Similar behaviour for *Goal Babbling*, reaching most 20% of success. Instead, *Distance-based Goal Babbling* results being extremely effective, reaching 90% of success. Conceptually, the strategy continuously pushes the agent towards the frontiers of the visited states, increasing the probability of encountering new unexplored areas and reducing redundant data.

4.4 Discussion and Future Work

As shown in Figure 9, the results are not deterministic and change over time. On the one hand, the system is continuously evolving in terms of knowledge and, on the other hand, ML techniques introduce further stochasticity to the final representation generated. During the abstraction procedure, described in section 3.1.3, some statistical tools are employed to create data structures to represent preconditions, effects and symbols.

For instance, an erroneous[6] clustering phase could generate unexpected effects on symbols and operators. An example highlighting this fact is the *noisiness of some symbols*, interfering with the generation of a correct formalization of $\mathcal{D}, \mathcal{P}$ and, consequently, the resulting plan ω. In Figure 10 it can be seen the graphical representation of the symbol "on the highest y-axis position", meaning "on the top ladder" because it is the only possible state with such y-axis value (it is not allowed to move inside the walls). In some cases, it could happen that the initial state of the game is interpreted

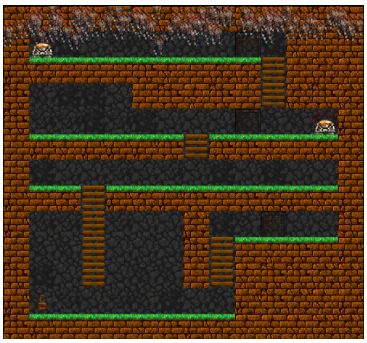


Fig. 10: The graphical representation of the symbol "on the higher y-axis position". It can be seen that it is significantly noisy because some samples of the agent are depicted on the top y-axis and other samples almost on the lower floor.

as being at the top floor under the ladder and, consequently, it is not necessary to climb down the ladder to execute the agent's task. Then, although almost the whole plan ω is correct to complete the game, without the option of climbing down the ladder as the first action, the plan is incomplete, and the trial is considered unsuccessful. On the one hand, this is a drawback of using ML tools, which can introduce noise in

[6]Statistical methods are not mistaken because they just create a representation based on the data provided. However, for our purposes, some models' instances could be obstacle on the reaching of our goal.

the symbolic representation. On the other hand, it is the strong point of the probabilistic approach, always proposing solutions, even though presenting a certain degree of error.

The system proposed is limited by the *propositional logic* representation, which does not permit the inclusion of arguments in operators and symbols. Such limitation could be overtaken by developing a new abstraction procedure following other logics (e.g. First Order Logic (FOL)). Recently, an implementation of an object-centric abstraction procedure has been proposed [76], demonstrating that it is possible to create a lifted representation suitable for transfer the knowledge to new tasks.

Another aspect to be analyzed deeper is the *management of the knowledge data*. Unfortunately, the abstraction module has no real incremental nature, but the abstraction procedure is executed over all the data, each cycle from scratch. This way of operating is not computationally efficient and does not reflect the learning process of the human being. However, given that in our domains all the necessary knowledge is discovered in a certain amount of time and does not change, a first improvement to be applied to the system could be maintaining a maximum of transition and initiation data discarding over-sampled transitions. Therefore, a sort of filter in the knowledge acquisition could stop the growth in terms of the abstraction procedure's time in stationary and limited domains but also reduce it for all the other cases.

Then, the abstraction time seems being linear with respect to the number transition tuples (in the worst-case exponential according to Konidaris [59]). Consequently, it would be fundamental to find solutions to mitigate this aspect. Possible actions could be to filter the acquired data or structure the collected knowledge in another form more efficiently for the abstraction process and, eventually, produce different complementary representations which are used according to the task to be tackled.

Finally, a natural evolution of this work lies in the neuro-symbolic approach. The integration of explicit knowledge inside sub-symbolic systems makes the learning process more effective and efficient, taking advantage of both the approaches. One of most promising extension of this work should embrace this new techniques, as some other preliminar works have already done [54, 55, 57].

5 Conclusions

In this paper, a novel approach for open-ended learning in autonomous agents based on intrinsically motivated planning is presented. This approach integrates two powerful paradigms, intrinsic motivation and classical planning, to enable agents to continuously learn and improve their knowledge and skills without relying on external supervision or rewards.

This work suggests an alternative or complementary approach to the advanced and popular sub-symbolic methods, demonstrating interesting features. First, it allows agents to explore and learn in a self-directed and open-ended manner without being limited to a predefined set of goals or tasks. Second, it enables agents to represent and reason about their knowledge and skills in a structured and formal way, which can facilitate planning and generalization to new situations. Third, it can incorporate intrinsic motivations that drive the agent to explore and learn beyond extrinsic goals,

which can enhance the agent's adaptability, robustness, and creativity. However, there are still several challenges and opportunities for future research in this area to enable the systems to perform complex activities in a relevant operational environment.

Overall, we believe that our approach represents a promising step towards more autonomous and intelligent agents that can continuously learn and improve in an open-ended and self-directed manner.

Conflict of Interest. The authors declare that they have no conflicts of interest.

Data availability. The implementation of the system is available at https://github.com/gabrielesartor/discover_plan_act.

References

[1] Campbell M, Hoane AJ, hsiung Hsu F. Deep Blue. Artificial Intelligence. 2002;134(1):57–83. https://doi.org/https://doi.org/10.1016/S0004-3702(01)00129-1.

[2] Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science. 2018;362(6419):1140–1144.

[3] Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Van Den Driessche G, et al. Mastering the game of Go with deep neural networks and tree search. nature. 2016;529(7587):484–489.

[4] Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, et al. Mastering the game of go without human knowledge. nature. 2017;550(7676):354–359.

[5] Oh J, Guo X, Lee H, Lewis RL, Singh S. Action-conditional video prediction using deep networks in atari games. Advances in neural information processing systems. 2015;28.

[6] Berner C, Brockman G, Chan B, Cheung V, Dębiak P, Dennison C, et al. Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:191206680. 2019;.

[7] Koch W, Mancuso R, West R, Bestavros A. Reinforcement learning for UAV attitude control. ACM Transactions on Cyber-Physical Systems. 2019;3(2):1–21.

[8] Nguyen H, La H. Review of deep reinforcement learning for robot manipulation. In: 2019 Third IEEE International Conference on Robotic Computing (IRC). IEEE; 2019. p. 590–595.

[9] Ibarz J, Tan J, Finn C, Kalakrishnan M, Pastor P, Levine S. How to train your robot with deep reinforcement learning: lessons we have learned. The International Journal of Robotics Research. 2021;40(4-5):698–721.