Faster MIL-based Subgoal Identification for Reinforcement Learning by Tuning Fewer Hyperparameters

SAIM SUNEL, Department of Computer Engineering, Middle East Technical University, Ankara, Turkey
ERKIN ÇILDEN, RF and Simulation Systems Directorate, STM Defense Technologies Engineering and Trade Inc., Ankara, Turkey
FARUK POLAT, Department of Computer Engineering, Middle East Technical University, Ankara, Turkey

Various methods have been proposed in the literature for identifying subgoals in discrete reinforcement learning (RL) tasks. Once subgoals are discovered, task decomposition methods can be employed to improve the learning performance of agents. In this study, we classify prominent subgoal identification methods for discrete RL tasks in the literature into the following three categories: graph-based, statistics-based, and multi-instance learning (MIL)-based. As contributions, first, we introduce a new MIL-based subgoal identification algorithm called EMDD-RL and experimentally compare it with a previous MIL-based method. The previous approach adapts MIL's Diverse Density (DD) algorithm, whereas our method considers Expected-Maximization Diverse Density (EMDD). The advantage of EMDD over DD is that it can yield more accurate results with less computation demand thanks to the expectation-maximization algorithm. EMDD-RL modifies some of the algorithmic steps of EMDD to identify subgoals in discrete RL problems. Second, we evaluate the methods in several RL tasks for the hyperparameter tuning overhead they incur. Third, we propose a new RL problem called key-room and compare the methods for their subgoal identification performances in this new task. Experiment results show that MIL-based subgoal identification methods could be preferred to the algorithms of the other two categories in practice.

CCS Concepts: • **Computing methodologies → Sequential decision making;**

Additional Key Words and Phrases: Subgoal identification, expectation-maximization, diverse density, hyperparameter search, multiple instance learning, reinforcement learning

ACM Reference Format:
Saim Sunel, Erkin Çilden, and Faruk Polat. 2024. Faster MIL-based Subgoal Identification for Reinforcement Learning by Tuning Fewer Hyperparameters. *ACM Trans. Autonom. Adapt. Syst.* 19, 2, Article 10 (April 2024), 29 pages. https://doi.org/10.1145/3643852

1 INTRODUCTION

Solving **reinforcement learning (RL)** tasks with a divide-and-conquer strategy was found to be fruitful in the RL literature [6, 25, 27], and this has led to the proliferation of many task decomposition algorithms in recent decades. A decomposition technique divides a given RL task

This study was supported by Grant No. 2210-A (2019/2) National Scholarship Programme for MSc students of the Scientific and Technological Research Council of Turkey (TÜBİTAK).
Authors' addresses: S. Sunel and F. Polat, Department of Computer Engineering, Middle East Technical University, 06800 Ankara, Turkey; e-mails: saimsunel@ceng.metu.edu.tr, polat@ceng.metu.edu.tr; E. Çilden, RF and Simulation Systems Directorate, STM Defense Technologies Engineering and Trade Inc., 06530 Ankara, Turkey; e-mail: erkin.cilden@stm.com.tr.

ACM 1556-4665/2024/04-ART10
https://doi.org/10.1145/3643852

into smaller tasks and tackles each subtask individually. Solutions to these subtasks are combined to accomplish the main objective more efficiently. These smaller tasks are critical, since they are intermediate steps toward solving the problem.

If employed at the early stages of problem-solving, then task decomposition can accelerate the learning performance of RL agents by increasing exploration efficiency and shortening the time required to achieve goal [1–5, 16, 17, 19, 25]. Many decomposition methods seek to identify critical states on a given problem, called *subgoals*. After discovering these states, agents first attempt to learn policies to reach these subgoals. After obtaining solutions to these subgoals, agents can learn at more abstract policy levels. With such an abstraction, agents do not have to realize their learning solely with the low-level actions supported by tasks. They can learn to develop policies over policies and further blend low-level actions with more abstract behavior. For instance, an agent may be allowed to apply only four actions that move it forward, back, left, and right directions in a given task. After running a subgoal analysis, the agent may find that a particular state acts as a gateway and regard this state as a subgoal. To reach this subgoal, it can learn an abstract action, "move to the gateway," by analyzing its trajectories. Thus, the agent can execute this abstract action whenever it wishes to transition to the gateway state.

In the related literature, various studies have proposed methods to tackle the subgoal identification problem in discrete RL tasks by approaching it from different perspectives. Several studies use graph theory methods and metrics on a graph constructed by inspecting the state transitioning history of an agent [4, 17]. Another group of studies devises statistical metrics mainly depending on the occurrences of states in trajectories [3, 13]. One study takes a unique approach and treats the subgoal identification problem as if it is a problem of **multi-instance learning (MIL)** paradigm [16]. With these fundamental differences, we can group these studies into three main categories: graph-based, statistics-based, and MIL-based. Throughout this article, we advocate MIL-based subgoal identification algorithms thanks to their unique approach to the subgoal identification problem and the advantages they provide in practice.

Tuning hyperparameters of algorithms for learning tasks is of great importance in machine learning, because hyperparameters directly determine the behavior and learning capabilities of methods [21]. So, a hyperparameter search procedure has to be carried out to find the best-performing hyperparameter configuration in a particular problem. Reinforcement learning and subgoal identification methods are no exception [10, 24]. For every new RL task encountered, hyperparameters of methods have to be adjusted to acquire optimal performance. The effort to search for optimal hyperparameter values correlates with the size of the hyperparameter search space. Every hyperparameter constitutes one dimension of the search space, and as the number of hyperparameters increases, the number of hyperparameter configurations to test increases exponentially. Deciding values for hyperparameters can become demanding, especially for those that can take a value from an infinite set, since some values or value ranges must be opted among infinitely many options. The situation can worsen if the performance of a method is highly susceptible to its hyperparameters. In such cases, searching for an optimal value turns out to be looking for a needle in a haystack, as the exact hyperparameter values have to be determined for the algorithm to perform optimally. Thanks to their small number of hyperparameters with values from a finite set, MIL-based methods can easily be exempted from the intricacies that increase the effort for a hyperparameter search.

In addition to bolstering MIL-based studies, we propose a new MIL-based subgoal identification method in this study. Our method has been developed mainly by getting inspiration from a previously proposed MIL-based approach, which we refer to as subgoal identification with Diverse Density (SDD) [16]. The method is an adapted version of a well-known MIL algorithm called **Diverse Density (DD)** for the subgoal identification problem. The DD algorithm is one of the first techniques proposed in the MIL domain. As the successor of DD, the **Expectation-Maximization Diverse Density (EMDD)** algorithm [28] surpasses DD concerning both speed and accuracy performances. Thanks to its superiority, we have adapted EMDD to identify subgoals in discrete RL tasks. In the upcoming sections, we present our algorithm coined EMDD-RL in detail and provide experimental evidence to support its usefulness in several RL problems by comparing it with SDD for speed and accuracy performances.

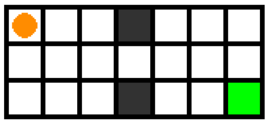
The availability of many methods for subgoal identification makes it hard to devise a unifying experimental setup under which all methods are tested for their accuracy performances in multiple RL tasks. It is because a hyperparameter search for each method has to take place for each task, and many RL tasks have to be designed, both of which require a significant amount of effort and time. Instead of comparing methods of the three categories over a wide range of RL problems, we first focus on the hyperparameter search overhead they incur while being employed for several discrete RL problems. Second, we compare their identification accuracy performances in a new RL task. To summarize, we follow three main directions throughout this article to compare subgoal identification methods. In the first direction, as EMDD-RL is a new MIL-based approach, we compare it with SDD. In the second direction, we conduct experiments to assess the hyperparameter tuning overhead of the methods in the three categories. Comparing the subgoal identification performance of the methods in the new RL task constitutes the third direction.

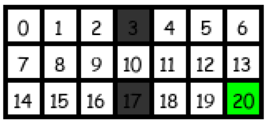
This work focuses mainly on the subgoal identification problem in discrete RL tasks. After identifying subgoals, a task decomposition method can divide the problem into smaller subproblems and learn the optimal policies for them, whose concerns and research questions differ from the methods presented in this article. Once a subgoal identification method and a decomposition method are selected, they can be combined to solve a given RL problem faster than the conventional RL methods (e.g., Q-learning [26], SARSA [22]). To this end, *options* framework [25] could be considered as the task decomposition algorithm, which can develop and learn abstract policies over primitive ones (actions).

The article's outline is as follows: in Section 2, we introduce the prominent subgoal identification techniques in the literature in a detailed manner by categorizing them depending on their approach to the subgoal identification problem. In the same section, along with SDD, we also introduce DD and EMDD algorithms to lay the foundations for our algorithm. In the following section (Section 3), we present our MIL-based subgoal identification method and explain the necessary algorithmic details and internals. Next, we outline our experimentation setup for the three directions mentioned previously and present the results of the experiments in Section 4. In Section 5, we discuss the experiment results and emphasize why the MIL-based methods should be preferred in practice for subgoal identification in discrete RL problems. Last, we finalize our study with concluding remarks and future directions in Section 6.

2 SUBGOAL IDENTIFICATION IN REINFORCEMENT LEARNING

Reinforcement learning constitutes the third paradigm in machine learning and aims to develop decision-maker entities (agents) that can solve sequential decision-making problems through trial and error. A mathematical framework called **Markov Decision Process (MDP)** [24] represents a reinforcement learning problem (environment). MDP is a four-tuple (S, A, R, T) where S is the set of states in which an agent can be, A is the set of available actions for the agent, R is the reward function that determines the feedback signal delivered to the agent, and T is the transitioning function that governs the probability of transitioning from one state to another. An agent interacts with this framework (environment) during learning and seeks an optimal action sequence (policy). The agent decides on an action and applies it to an environment; as a response, the environment delivers a reward score and the new state information to the agent. The main goal of the agent is


(a) A small version of the two-rooms environment



(b) Each cell with a number represents a state.

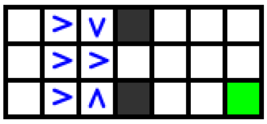

(c) A sample policy that takes the agent to the other room.

Fig. 1. Sample grid-world problem

to maximize the expected reward received ($\mathbb{E}[\sum_t \gamma^t R_t]$) over time, where R_t is the reward value received at time step t and γ is the discount factor regulating the importance of future rewards. Throughout the study, we refer to the environments whose state and action sets are finite as discrete tasks.

There have been plenty of studies to identify subgoal states to decompose a discrete RL problem. Depending on how methods approach identifying subgoals, there are different subgoal definitions. For instance, Reference [12] defines the states between strongly connected areas on the graph constructed via an agent's interaction history (trajectories) as subgoals. Different from this definition, [3] considers the states that enable an agent to transition to an unvisited part of the state space of an environment. In this article, we stick to the definition provided by Reference [16], where a subgoal is a state that is present in successful trajectories (an agent reaches a goal state in these trajectories) but not present in negative ones.

Most subgoal identification studies consider grid-world problems in which an agent has to pass through several gateways and obstacles to reach a goal state. A sample grid-world problem is illustrated in Figure 1(a). Each square represents a state, which is colored depending on its type (obstacle state is black, goal state is green, and visitable is white). In the problem, the obstacle states divide the state space into two rooms, and an agent represented with an orange circle aims to learn a sequence of actions that will lead it to the goal state located at the bottom right. Whenever it reaches the goal state, a new episode begins. At the beginning of each episode, the agent is randomly located at one of the states in the left-hand side room.

Figure 1(b) shows the numeric labels of the states. With the subgoal definition considered in this work, state 10 is a good candidate, because it must be present on successful trajectories (the agent has to pass through it to reach the goal state). These gateway states and the states around them are valuable as they facilitate an agent to transition to different parts of the state space of a problem. If the agent learns how to reach these states before an actual goal state, then it can navigate between rooms quickly and use its time to explore other rooms. Thus, the agent can get the goal state quickly and learn the action sequence to solve the problem faster.

The subgoal definitions presented in this work apply to grid-world problems and are valid for other types of discrete RL problems, because a transitioning graph can still be constructed. On this graph, we can still expect strongly connected regions to occur or some specific states to be present on successful trajectories. We prefer grid-world problems in this work as they facilitate visualization.

Enhancing the learning performance of a reinforcement learning agent requires more than just identifying subgoals. Employing **hierarchical reinforcement learning (HRL)** techniques with the identified subgoals is usually essential. In the literature, hierarchical reinforcement learning methods [6, 18, 25, 27] propose techniques to accelerate learning and improve the problem-solving capability of an agent. These methods decompose a given RL problem into sub-tasks and learn a hierarchy over the action sequences that solve these subtasks.

One of the well-known HRL methods is the *options* framework [25]. This framework requires subgoals to decompose a problem into smaller subtasks. It first solves these subtasks and then learns how to combine the solutions to achieve an actual goal. An *option* is a close-loop action sequence defined over a particular state-space region and solves a specific subproblem. Figure 1(c) shows a sample *option* that takes the agent to the subgoal state (state 10). This *option* facilitates the agent to transition to the right-hand-side room where the actual goal state is; hence, it is highly likely that the agent will reach the goal state faster by utilizing this *option*. In addition to their advantage of facilitating learning, when several tasks are similar to each other (e.g., they have similar state sets and problem structure), an *option* learned in a task can be reused for the others to shorten the learning duration for an agent further [16]. With this respect, transfer learning can be employed for new RL tasks with identified subgoals and learned *option*s.

The **multi-instance learning paradigm (MIL)** emerged due to the need for a solution to the drug activity prediction problem [7] and is a generalization of the conventional supervised learning paradigm. In this paradigm, data instances do not have individual labels. Instead, data instances belong to groupings called bags, and only the labels of these bags are known. So, datasets of this paradigm consist of bag-label pairs. DD [14] was the first MIL algorithm adapted for subgoal identification. As a successor to DD, EMDD [28] offers superior accuracy and speed performances in the MIL domain. To frame subgoal identification as an MIL problem, bag-label pairs should be acquired from an RL problem setting. While an agent interacts with an environment, it can store its interaction history (sequence consisting of rewards, states, and actions). If the problem is an episodic task, then the agent can create a new bag for each episode, and depending on a success criterion, it can label this bag as positive or negative. The agent can also follow a similar approach for non-episodic tasks (e.g., after reaching a particular state or obtaining a particular reward signal, it can create a new bag and label it according to some other success criterion.)

In the following subsections, we provide a detailed review of the prominent subgoal identification methods of the three categories (graph-based, statistics-based, and MIL-based) for discrete RL tasks.

2.1 Graph-based Subgoal Identification

Graph-based subgoal identification algorithms use graph theory metrics and algorithms to discover subgoals. By utilizing the interaction history of an agent with an environment, it is possible to create a transitioning graph. This graph can reveal the overall structure of the environment; hence, by employing graph theory algorithms (e.g., max-flow/min-cut, partitioning) and metrics (e.g., centrality metrics), subgoal locations can be discovered. Each algorithm differs in the technique/metric used and assumptions that they make, which naturally leads to having advantages and disadvantages over one another.

2.1.1 Local Graph Partitioning (L-Cut) . The L-Cut algorithm [4] constructs a transitioning graph by utilizing the recent trajectories of an agent. According to the study, a subgoal refers to the state between two regions on the graph where the chances of the agent transitioning from one region to another are less than the likelihood of it staying in those regions. In grid-world problems, these subgoals correspond to gateway locations that agents must pass through to transition between rooms. The algorithm uses a cut metric called NCUT [4] to partition the recent transitioning graph. After partitioning, states connecting the regions are considered subgoals. Periodically, the algorithm constructs a transitioning graph and partitions it. Accepting every identified state as a subgoal in this construct-partition cycle is inappropriate, because the agent may have followed a random trajectory. The study proposes a simple threshold mechanism to eliminate falsely identified subgoals.

The method requires four threshold values for states to be a subgoal. The first one is the t_c hyperparameter, which sets a threshold value on the cut metric value while partitioning the graph.

The method does not yield subgoals if a partitioning cannot attain a cut score above this threshold. The second one is the t_o hyperparameter, with which the algorithm checks whether an identified subgoal state has been observed in the trajectory history sufficiently many times since the beginning of the agent's interaction with the environment. The method uses the third hyperparameter, t_p, to allow or reject a state s to be a subgoal depending on whether the ratio $\frac{c_s}{i_s}$ is above the t_p value, where c_s is the number of times the agent is in a state s and i_s is the number of times the method identifies it as a subgoal. The fourth hyperparameter (h) regulates the length of the recent trajectory history of the agent.

The authors of the study have left a strategy to select hyperparameter values unmentioned. Since $t_c, t_p \in \mathbb{R}$, the number of the values that we can consider is theoretically infinite for both hyperparameters, whereas t_o and h hyperparameters can take values from a finite set.

2.1.2 Q-Cut. Like L-Cut, the Q-Cut algorithm [17] also constructs a transitioning graph. It regards states in the middle of highly connected subgraphs as subgoals. The algorithm treats the transitioning history as a flow graph and employs a Max-Flow/Min-Cut algorithm to identify subgoal locations.

The authors propose a relative visitation frequency metric over states for building the flow graph. Like L-Cut, the cut procedure can partition every given graph. So, the algorithm considers only the significant cuts to yield reliable subgoals. Checking the reliability of the cut is carried out by measuring a metric (ratio cut). If a cut attains a ratio cut score above a certain threshold, then the algorithm considers it valid and regards the states on the cut location as subgoals.

The algorithm runs iteratively. Once an agent has interacted with an environment for a designated time, the algorithm initiates the cut process. Mainly, it requires two hyperparameters for its operations. The first one is the cut threshold value (t_c, $t_c \in \mathbb{R}$), and the second one is the step threshold (t_s), which determines the number of actions the agent must take in an environment before executing the cut procedure. The t_c hyperparameter can have any value from an infinite set, while t_s takes an integer value. Like L-Cut, the authors have left a well-defined strategy to ascertain optimal values for the hyperparameters undescribed.

2.1.3 Segmented Q-Cut. The Segmented Q-Cut [17] algorithm is proposed to circumvent the problem of identifying multiple subgoals in an environment with Q-Cut. Unlike the original Q-Cut, the algorithm constructs local graphs. With the cut-checking mechanism of Q-Cut, it decides whether to partition a local flow graph.

The algorithm starts with a single local graph constructed by inspecting the trajectory history of an agent, and it checks the cut quality on the graph regularly. If a cut is reliable, then this local graph is partitioned into two smaller local graphs. While the agent continues interacting with the environment, the algorithm expands these two graphs via trajectories depending on which graph region the agent is visiting. It repeats this procedure to identify multiple subgoals.

Besides the t_c and t_s hyperparameters of Q-Cut, Segmented Q-Cut introduces a distance threshold (t_d, $t_d \in \mathbb{N}$) for picking the source or target node in the cut procedure. Depending on the t_c, t_s, and t_d values, the application frequency of the cut procedure changes. The t_c hyperparameter value should be picked from an infinite set, whereas t_d and t_s can take a value from a finite set of numbers. Like previous methods, no well-defined procedure to determine hyperparameter values is available in the study.

2.1.4 Strongly Connected Components (SCC). Similar to previous methods, this study [12] regards states that are in the middle of densely connected regions as subgoals and employs the *strongly connected components* identification procedure to find these subgoal locations. First, it constructs a directed transitioning graph from the transitioning history of an agent. Then, an algorithm (the authors call **Strongly Connected Components (SCC)**-Inspector) is run over the graph to obtain strongly connected components. The method considers the states bridging strongly connected components as subgoals.

To obtain reliable connected components, the authors impose a threshold value (t_t, $t_t \in \mathbb{N}$) on the edge weights of the transitioning graph. Those edges that pass the threshold test remain on the graph for connected component analysis. This threshold value is the only hyperparameter of the method. The authors propose to inspect the edge weight histogram of the transitioning graph to set an optimal value for the hyperparameter.

2.1.5 *Betweenness.*

The method [23], different from Q-Cut and L-Cut, calculates a centrality metric (betweenness) over states for subgoal identification. In a problem, if an agent often has to pass a state or a group of states, then they possess high importance to achieve the goal; hence, they are highly likely to be subgoals. The betweenness metric accentuates these central locations. The algorithm constructs a transitioning graph by inspecting trajectory histories. The method computes the betweenness centrality value for each state on the graph and picks the states that achieve the local maximum score as subgoals.

Calculating the betweenness metric on the whole graph incurs significant computation overhead, especially for problems with large state-space sizes. To remedy this problem, the authors propose an incremental version of the algorithm that constructs local subgraphs and performs the search for the most central state on these small graphs via the betweenness metric. The incremental version introduces a couple of hyperparameters to get reliable results cleansed from the noise that might emerge due to randomness in trajectories. The authors inherit some hyperparameters ($t_p \in \mathbb{R}$ and $t_o \in \mathbb{N}$) of L-Cut for this purpose. Once a state attains the highest local betweenness score on the local graph, it must meet the threshold values set by t_o and t_p to become a subgoal. In addition, it is crucial to determine how often the states on the local graph should get evaluated for the subgoal criteria. The incremental version introduces a step threshold ($t_s \in \mathbb{N}$) to govern the evaluation frequency, similar to the L-Cut study. No well-defined technique to choose hyperparameter values for new problems is available in the study.

2.2 Statistics-based Subgoal Identification

Unlike graph-based algorithms, statistics-based methods calculate metrics on states without constructing a graph. They mainly depend on the occurrences of states in trajectory histories. Thus, these methods generally have better time and space complexities than graph-based methods for subgoal identification.

2.2.1 *Relative Novelty (RN).*

The **relative novelty (RN)** method [3] approaches the subgoal identification problem from a different perspective. It aims to identify the locations where an agent transitions to a previously unvisited part of the state space by measuring a metric called relative novelty. The authors define the novelty score of a state as the square root of its observation frequency in trajectory history. To infer whether the agent transitions to a new state-space region, the algorithm calculates the relative novelty scores of states, where relative novelty is the square root of the summation of novelty values of a group of states.

The authors take the relative novelty score ratio between two groups of states into account for subgoal identification. The first group consists of the states that precede a particular state s (predecessors), and the other consists of the states that are present in trajectories after state s (successors). If the ratio of novelty scores of predecessors and successors of a particular state is high, then this state is likely responsible for the agent to transition to a new state-space region; hence, it is a subgoal candidate.

To calculate the relative novelty ratio, the method requires the number of predecessors and successors as a hyperparameter (novelty lag, $l_n \in \mathbb{N}$). After calculating its ratio value, the authors employ Bayesian decision theory to accept a state as a reliable subgoal, which requires the following hyperparameters: t_{RN}, p, q, $\lambda_{fa}/\lambda_{miss}$, $p(N)/P(T)$.

The authors provide a well-defined procedure to set hyperparameter values for t_{RN}, p, q, $\lambda_{fa}/\lambda_{miss}$, $p(N)/P(T)$, which requires experience data collection from various RL tasks. However, a hyperparameter search must be performed for the novelty lag hyperparameter.

2.2.2 *Frequency-Distance.* The objective of the study in Reference [13] is to remove three prerequisites of the SDD algorithm: knowledge of the distance between states, static filtering to exclude certain states from calculations, and the requirement for negative trajectories (bags). To this end, the authors introduce two numerical measures calculated for every state in trajectory histories: frequency and distance. The frequency measure calculates the normalized occurrence frequency for a state and aims to emphasize states that occur frequently in trajectories. Similarly, using a Gaussian-like function, the distance measure calculates a transformed distance score for each state. This measure eliminates the need for static filtering (state elimination) and prioritizes states in the middle of trajectories. The algorithm uses the multiplication of these two measures to identify subgoals. Transformed distance calculations require two hyperparameters ($a, b \in \mathbb{R}$). Unfortunately, the study lacks a well-defined procedure to assign optimal hyperparameter values. A serious disadvantage of the method is that it requires many trajectories to solve a problem.

2.3 MIL-based Subgoal Identification

MIL-based subgoal identification methods mainly convert the subgoal identification problem into a MIL problem. Thanks to the MIL paradigm, they inherently seek what is helpful for an agent to achieve its goal. They label each trajectory positively or negatively depending on a success criterion and aim to find out what causes the discrimination between positive and negative bags.

When applied for subgoal identification in an RL problem, these methods require a dataset consisting of positive and negative bags. In an RL setting, the primary data source is an agent's trajectory history. We can label a trajectory as positive or negative depending on a criterion. A simple criterion can be checking whether the agent has reached a goal state in a trajectory (if it has reached, we can label the trajectory as positive; otherwise, negative). For instance, suppose that the following trajectories have been gathered from the sample environment in Figure 1: {{0, 1, 2, 9, 10, 11, 4, 5, 6, 13, 20}, {7, 0, 7, 8, 15, 16}, {8, 9, 10, 11, 18, 19, 20}, {2, 9, 10, 11, 4, 12}}. Since the first and the third trajectories have ended in the goal state, we label these two trajectories as positive. Similarly, the second and fourth trajectories are labeled as negative.

2.3.1 *Diverse Density.* The DD algorithm [14, 15] calculates a probability score (diverse density) that signifies how important a data instance is to distinguish between positive and negative bags. The data instance with the highest DD value is called the concept instance, and the algorithm uses it to label new unseen bags.

Geometrically, the algorithm considers each bag as a path in a multi-dimensional space where each data instance is a point. It aims to pinpoint a specific location (the location of the concept instance) in the space that must satisfy the following conditions: (1) most positive bags should have instances close to this location, and (2) instances of negative bags should be far from this location. A data instance that appears in many positive bags but not in negative bags yields a high DD score.

A search over all data instance space must be performed to find the location with the highest DD value for a given dataset. Since it is computationally intractable, especially in a continuous space, the authors suggest initiating the search procedure from individual data instances. More formally, the algorithm maximizes the DD value by starting from each data instance t in positive bags, which is mathematically expressed as

$$DD(t) = \max_y P\left(y = t | B_1^+ B_2^+ B_3^+ \cdots B_1^- B_2^- B_3^- \cdots\right), \tag{1}$$

where B_i^+ and B_j^-represent the ith positive bag and the jth negative bag, respectively. After applying the Bayesian formula two times and assuming: (1) all data instances have the same prior probability scores, (2) bags are conditionally independent for a given data instance t, the Equation (1) becomes (the normalization factor is the same for all data instances):

$$DD(t) = \max_y \prod_i^{n^+} P(y = t|B_i^+) \prod_j^{n^-} P(y = t|B_j^-), \tag{2}$$

where n^+ and n^- are the total number of positive and negative bags, respectively. For representing the probability of a data instance with given bags, the noisy-or model can be employed:

$$\begin{aligned} P(y = t|B_i^+) &= 1 - \prod_k^{n_i^+} (1 - P(y = t|B_{ik}^+)), \\ P(y = t|B_j^-) &= \prod_l^{n_j^-} (1 - P(y = t|B_{jl}^-)), \end{aligned} \tag{3}$$

where B_{ik}^+ is the kth instance in the ith positive bag, B_{jl}^+ is the lth instance of the jth negative bag; n_i^+ and n_j^- are total instance counts of the ith positive bag and the jth negative bag, respectively. $P(y = t|B_{ik}^+)$ (and also $P(y = t|B_{jl}^-)$) models the probability of the instance t for a given other instance, and it is represented via a Gaussian-like distribution:

$$P(y = t|B_{mn}) = \exp(-sd\|y - B_{mn}\|^2), \tag{4}$$

where sd is a scaling factor. Equation (1) is maximized via the gradient ascent algorithm for each data instance in positive bags. Among all the final maximization results attained, the one with the highest DD score is considered to be the concept instance:

$$\begin{aligned} C &= \{B_{ij}^+ | 1 \le i \le n^+, 1 \le j \le n_i^+\}, \\ t^* &= \arg\max_{x \in C} DD(x). \end{aligned} \tag{5}$$

2.3.2 Expectation Maximization Diverse Density (EMDD). The EMDD algorithm [28] combines the **expectation-maximization (EM)** technique with the Diverse Density algorithm. It regards the concept instance as an unknown entity and employs EM to estimate it. It starts with an initial guess for the concept instance. The initial guess is updated, and its associated DD score is improved gradually by employing the EM technique. In the **expectation (E)** step, the algorithm picks a single instance from every bag depending on how close data instances are to the current guess. A probability distribution model measures the closeness between instances. These selected instances constitute the dataset for the DD maximization procedure in the M step (every bag has a single data instance). After maximization, the resulting data point becomes the updated concept instance. These EM steps are repeated till the DD value of the concept instance stops improving.

To find the concept instance, DD uses all instances in positive bags as initial starting points while searching. Unlike DD, the EMDD algorithm considers a relatively small set of these instances. In addition, during the DD value maximization step, fewer instances are used to calculate gradient information. DD uses all the instances in bags, whereas EMDD considers only a single instance from each bag. Thus, EMDD incurs less computation overhead compared to DD. Thanks to the EM algorithm, it outperforms DD also with respect to accuracy performance in the MIL domain.

2.3.3 SDD Algorithm. The SDD algorithm [16] approaches the subgoal identification problem as a MIL domain problem. Each episode history of an agent corresponds to a bag, and visited states are the instances in those bags. SDD adapts the DD algorithm for the discrete nature of tasks with a discrete state space by discarding gradient ascent calculations. Instead, it calculates DD scores of states without the maximum operator in Equation (1). The method picks the state attaining the highest DD value as a subgoal.

At the early stages of learning, the state with the highest DD score may not be a true subgoal. To alleviate this problem, the authors propose a counting mechanism to eliminate falsely identified states. In the study, after identifying subgoal states confidently, the *options* framework [25] is employed to decompose and solve problems efficiently.

The algorithm can pick the states that are spatially very close to a goal state in trajectories as subgoals, which tend to have high DD values. It applies static filtering [11] to eliminate these states from DD calculations.

Algorithm 1 provides the pseudo-code for the SDD algorithm. The pseudo-code only involves the identification of a subgoal (all *options* framework-related parts of the method proposed in Reference [16] are discarded). This is because, in this study, we mainly focus on the subgoal identification accuracy performances of methods. The `SDD` function requires positive bags, negative bags, a transitioning graph, and a list of states. First, it eliminates some states from positive and negative bags for static filtering (line 2). Next, it scans all the positive bags and gathers their instances into a set (lines 3–8). It then calculates the DD scores by iterating through each instance within the set (lines 9–17) and returns the instance with the highest DD score as a subgoal (line 18).

The function `DD` returns the adapted DD score of a given instance (Equation (2) without the maximum operator). It calculates two conditional probability scores for the given instance via positive bags (lines 26–28) and negative bags (lines 29–31). Then, it combines the results of both parts into a single value via multiplication (lines 27 and 30). Conditional probability calculations are carried out via two functions `PrPositive` and `PrNegative`. These functions implement Equation (3). The instance-instance similarity is measured via the `PR` function, which implements Equation (4). For instance-instance similarity calculations, a distance value is required to measure the proximity of two states (line 22). To this end, the method utilizes a transitioning graph. This graph can be constructed using the states in positive and negative bags.

3 EMDD-RL ALGORITHM

This section describes our novel EMDD-RL algorithm and provides its pseudo-code along with a basic execution flow diagram (Figure 2).

The EMDD-RL algorithm (Algorithm 2) adapts EMDD for the subgoal identification problem in discrete RL tasks. It has two main stages: seed instance selection and EM loop. Like SDD, EMDD-RL initially applies static filtering on bags (line 2). Then, the algorithm selects seed instances and employs the EM algorithm. Unlike EMDD, which randomly chooses multiple positive bags and uses their instances as seed instances, EMDD-RL picks the positive bag with the fewest instances and uses its instances as seeds (line 4, "Initial Seed Selection" block in Figure 2). EMDD-RL considers this particular positive bag for seed selection because of two main reasons. First, this bag corresponds to the shortest trajectory in which an agent achieves its goal; thus, it has fewer instances than other positive bags. With fewer seed instances, EMDD-RL incurs less computation overhead. Second, since it is a positive bag, one of its instances should have the highest DD score.

Further, EMDD-RL removes some of the selected seeds to reduce computation overhead (lines 5–8, "Skip some instances" block). The algorithm specifies the number of instances to be removed

ALGORITHM 1: SDD Algorithm

```
1: function SDD(positive bags $B^+$, negative bags $B^-$, transitioning graph $G$, array $eliminated\_states$)
2:     Remove each $x \in eliminated\_states$ from all bags
3:     $all\_positive\_instances = set()$
4:     for each $B_i^+ \in B^+$ do
5:         for each instance $j \in B_i^+$ do
6:             add $j$ to $all\_positive\_instances$
7:         end for
8:     end for
9:     $highestDD = -\infty$
10:    $c^* = null$
11:    for each instance $j \in all\_positive\_instances$ do
12:        $dd\_j$ = DD($j$, $B^+$, $B^-$, $G$)
13:        if $dd\_j > highestDD$ then
14:            $c^* = j$
15:            $highestDD = dd\_j$
16:        end if
17:    end for
18:    return $c^*$
19: end function
20: function PR(instance $i$, instance $j$, transitioning graph $G$, dimension scalar $sd = 1.0$)
21:    Calculate distance $d$ between $i$ and $j$ on $G$
22:    return $\exp(-sd.d^2)$
23: end function
24: function DD(instance $i$, positive bags $B^+$, negative bags $B^-$, transitioning graph $G$)
25:    $product = 1.0$
26:    for each $B_i^+ \in B^+$ do
27:        $product *= $ PrPositive($i$, $B_i^+$)
28:    end for
29:    for each $B_i^- \in B^-$ do
30:        $product *= $ PrNegative($i$, $B_i^-$)
31:    end for
32:    return $product$
33: end function
34: function PrPositive(instance $i$, positive bag $B_j^+$, transitioning graph $G$)
35:    $product = 1.0$
36:    for each instance $b_i^+ \in B_j^+$ do
37:        $value$ = PR($i$, $b_i^+$, $G$)
38:        $product *= (1.0 - value)$
39:    end for
40:    return $(1.0 - product)$
41: end function
42: function PrNegative(instance $i$, negative bag $B_j^-$, transitioning graph $G$)
43:    $product = 1.0$
44:    for each instance $b_i^- \in B_j^-$ do
45:        $value$ = PR($i$, $b_i^-$, $G$)
46:        $product *= (1.0 - value)$
47:    end for
48:    return $product$
49: end function
```

through its hyperparameter k. Mainly, this hyperparameter has a direct effect on the total number of operations performed. With this basic removal strategy, EMDD-RL aims to pick several instances in a state-space region by eliminating neighboring states, because DD scores in a state-space region are close to each other. After determining the seed instances, the EM loop is executed for each of them individually (lines 12–18, "EMDD" block). During the EM loop, the goal is to find a new instance with a higher DD value than a given seed.

In the loop, initially, a given seed instance is considered to be the current concept instance (line 25). This current concept instance gets updated as long as its DD value improves. For calculating DD scores, the DD function of SDD is utilized. In the E step (lines 31–39, "Expectation" block), the algorithm picks one instance closest to the current concept instance from each bag using a Gaussian-like distribution. Later in the M step (lines 41–69, "Maximization" block), it uses these selected instances to obtain a new concept instance that maximizes bag label probabilities, which is the objective function of the M step. During maximization, EMDD-RL discards gradient ascent calculations of EMDD. Instead, it calculates the objective function score for each instance $a \in testset$ (lines 48–69). Then, it picks the state with the highest objective score as the new current concept instance (line 72). The content of $testset$ is determined according to the S hyperparameter of the method. EMDD algorithm considers the instances selected from a positive bag in the E step. EMDD-RL follows this behavior when S = S1 (line 43). However, EMDD-RL can also consider all instances in positive bags (when S = S2, line 46) for identifying the instance with the highest objective score. The reasoning behind this new behavior is that testing all instances in positive bags could expedite the detection of the instance with the highest DD score in the EM loop. Thus, EMDD-RL could complete calculations faster. To model bag label probabilities, the authors of EMDD propose two

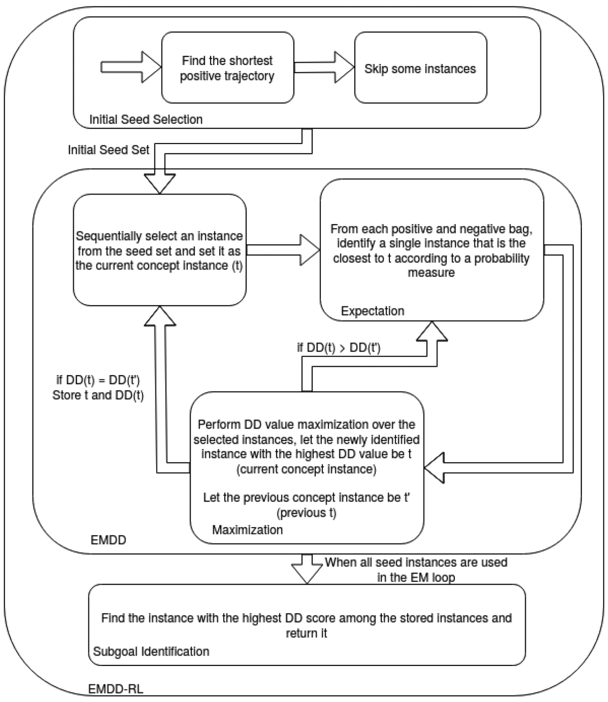

Find the shortest positive trajectory
Skip some instances
Initial Seed Selection
Initial Seed Set
Sequentially select an instance from the seed set and set it as the current concept instance (t)
From each positive and negative bag, identify a single instance that is the closest to t according to a probability measure
Expectation
if DD(t) > DD(t')
if DD(t) = DD(t') Store t and DD(t)
Perform DD value maximization over the selected instances, let the newly identified instance with the highest DD value be t (current concept instance)
Let the previous concept instance be t' (previous t)
Maximization
EMDD
When all seed instances are used in the EM loop
Find the instance with the highest DD score among the stored instances and return it
Subgoal Identification
EMDD-RL

Fig. 2. Execution flow diagram for EMDD-RL. The algorithm prepares a seed instance set for DD value maximization, and for each seed instance, it executes the expectation-maximization procedure. Each seed instance is a starting point for the maximization calculations. As long as the DD value is improved, expectation-maximization steps are repeated. After processing every seed instance, the method returns the instance with the highest DD score as a subgoal.

distribution models: **linear (LIN)** and **exponential (EXP)**. Therefore, selecting the appropriate distribution model is another hyperparameter for EMDD-RL, which is indicated by *model* in the pseudo-code (lines 50–56, lines 57–63).

EMDD calculates the objective function of the M step by multiplying probability scores, which can suffer from floating-point number issues in practice. EMDD-RL applies the logarithm function to circumvent these issues (lines 52, 55, 59, and 62). After processing all seed instances and completing maximization calculations (lines 11–18, "Subgoal Identification" block), it returns the instance with the highest DD value score as a subgoal (line 20). Similar to SDD, EMDD-RL requires similarity information between instances for its calculations. To this end, it utilizes the PR function of SDD, which requires a transitioning graph.

Compared to SDD, which uses all instances as seed instances in positive bags for DD maximization calculations, EMDD-RL considers fewer instances thanks to the EM loop. The EM loop identifies a concept instance by incurring less computation overhead. Moreover, since it dynamically searches the instance space until no DD score improvement occurs, it is likely to encounter more accurate concept instances. Thus, EMDD-RL can identify subgoals more accurately compared to SDD.

3.1 SDD and EMDD-RL Worst-case Time Complexity Analysis

This part provides a detailed worst-case run-time analysis of SDD and EMDD-RL for an episodic discrete RL task. Since both are MIL-based methods, they require the same input information:

ALGORITHM 2: EMDD-RL

```
1:  function EMDD-RL(positive bags $B^+$, negative bags $B^-$, transitioning graph $G$, skipping factor $k$, probability model $model$, maximization instance set $S$ = S1, array $eliminated\_states$)
2:      Remove each $x \in eliminated\_states$ from all bags
3:      // Construct initial seed set
4:      Find the smallest positive bag, $b^+_{smallest}$
5:      Let the size of $b^+_{smallest}$ be $l$
6:      // 0, k, 2k, 3k ... $l$
7:      $indices = \{i \mid 0 \leq i < l, i = i + k\}$
8:      $seeds = \{b^+_{smallest}[i] \mid i \in indices\}$
9:      $maxddvalue = -\infty$
10:     $t^* = null$
11:     // Find the state that has the highest DD value
12:     for $seed \in seeds$ do
13:         $< t, ddvalue >= $ EM($seed, B^+, B^-, G, sd$)
14:         if $ddvalue > maxddvalue$ then
15:             $maxddvalue = ddvalue$
16:             $t^* = t$
17:         end if
18:     end for
19:     // Return the state with the maximum DD value as the identified subgoal
20:     return $t^*$
21: end function
22: function EM(instance seed, positive bags $B^+$, negative bags $B^-$, transitioning graph $G$, probability model $model$, maximization instance set $S$ = S1)
23:     $nldd0 = \infty$
24:     $nldd1 = -\log(\mathrm{DD}(seed, B^+, B^-, G))$
25:     $peakstate = seed$
26:     $highestDD = -1 * nldd1$
27:     // Keep updating the current guess as long as the DD value improves
28:     // EM loop
29:     while $nldd1 < nldd0$ do
30:         // E step
31:         $p^*_+ = set(), p^*_- = set()$
32:         for each bag $B_i \in B^+$ do
33:             $p = \arg\max_{B_{ij} \in B_i} \mathrm{PR}(B_{ij}, peakstate, G)$
34:             append $p$ to $p^*_+$
35:         end for
36:         for each bag $B_k \in B^-$ do
37:             $p = \arg\max_{B_{kl} \in B_k} \mathrm{PR}(B_{kl}, peakstate, G)$
38:             append $p$ to $p^*_-$
39:         end for
40:         // M step
41:         $h' = null$; $maxhvalue = -\infty$
42:         if $S$ = S1 then
43:             $testset = \{p_i | p_i \in p^*_+\}$
44:         else if $S$ = S2 then
45:             // All instances in positive bags
46:             $testset = \{p_i | p_i \in B^+_i, B^+_i \in B^+\}$
47:         end if
48:         for each instance $t_i \in testset$ do
49:             sum = 0.0
50:             if $model$ = linear then
51:                 for $p_i \in p^*_+$ do
52:                     $sum+ = \log(1 - |1 - \mathrm{PR}(p_i, t_i, G)|)$
53:                 end for
54:                 for $p_l \in p^*_-$ do
55:                     $sum+ = \log(1 - \mathrm{PR}(p_l, t_i, G))$
56:                 end for
57:             else if $model$ = exponential then
58:                 for $p_i \in p^*_+$ do
59:                     $sum+ = \log(\exp(\mathrm{PR}(p_i, t_i, G) - 1))$
60:                 end for
61:                 for $p_l \in p^*_-$ do
62:                     $sum+ = \log(\exp(\mathrm{PR}(p_l, t_i, G)))$
63:                 end for
64:             end if
65:             if $sum > maxhvalue$ then
66:                 $sum = maxhvalue$
67:                 $h' = t_i$
68:             end if
69:         end for
70:         $nldd0 = nldd1$
71:         $nldd1 = -\log(\mathrm{DD}(h', B^+, B^-, G))$
72:         $peakstate = h'$
73:         $highestDD = -1 * nldd1$
74:     end while
75:     return $< peakstate, highestDD >$
76: end function
```

positive bags, negative bags, and a transitioning graph. Here, we assume that the task imposes a certain step limit for an agent to form a negative bag. If the agent reaches a goal state without exceeding the step limit in an episode, then this episode's history is considered a positive bag; otherwise, it is a negative bag. After the agent finishes interacting with the task, bags are formed with trajectory histories. The number of bags and the step limit determine the input size for both methods.

We consider the following definitions: θ is the set of states of the task, n is the number of states of the task ($n = |\theta|$), n^+_{bag} is the number of positive bags, n^-_{bag} is the number of negative bags, $step$ is the step limit value imposed in the task, B^+_i is a positive bag where $B^+_i = \{s_j | j = 1..O(step), s_j \in \theta\}$, B^+ is the set of positive bags ($B^+ = \{B^+_i | i = 1..n^+_{bag}\}$), B^-_m is a negative bag where $B^-_m = \{s_k | k = 1..step, s_k \in \theta\}$ and B^- is the set of negative bags ($B^- = \{B^-_m | m = 1..n^-_{bag}\}$). Each bag B_u (positive or negative) consists of $\alpha_u * n$ instances where each α_u is a constant factor (for negative bags $\alpha_u = step/n$, the same equation is valid for positive bags in the worst case). In the following subsections, we first provide the analysis for common parts of the methods, and then the individual algorithm analyses are presented.

3.1.1 Common Algorithmic Parts. The DD function (Algorithm 1) is common to both algorithms. It invokes the PrPositive function n^+_{bag} times and the PrNegative function n^-_{bag} times. Both PrPositive and PrNegative functions call the PR function, which first measures the distance between given instances (states) and uses the result in a mathematical expression. We can consider Dijkstra's algorithm [8] for distance calculations. The algorithm has a time complexity of $O(|E| + |V| \log |V|)$ [9] per distance measurement, where E and V are the edge set and vertex set of a transitioning graph, respectively. The graph can have $O(n)$ vertices and $O(n^2)$ edges at most.

Both PrPositive and PrNegative functions require $O(step) * O(|E| + |V| \log |V|)$ time (in the worst case), since they iterate through a positive or negative bag. Hence, the DD function has a worst-case time complexity of $n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|)$.

3.1.2 SDD Algorithm. The SDD function (Algorithm 1) identifies all distinct instances in positive bags in $n^+_{bag} * O(step)$ time and calculates a DD score for each of them. Since all the states can be distinct in the positive bags, the SDD function has a worst-case time complexity of $n^+_{bag} * O(step) * (n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|)) + n^+_{bag} * O(step)$.

3.1.3 EMDD-RL Algorithm. The EMDD-RL function (Algorithm 2) first finds the positive bag with the smallest size, then constructs a seed instance set, and iterates through seed instances by calling the EM function (Algorithm 2). The EM function first invokes the DD function for the initial seed instance and performs the expectation-maximization procedure as long as the DD score improves. In the E step, from every positive bag, a single instance is picked (n^+_{bag} instances), in $n^+_{bag} * O(step) * O(|E| + |V| \log |V|)$ time, by invoking the PR function. Later, the same procedure is applied for the negative bags (n^-_{bag} instances, in $n^-_{bag} * step * O(|E| + |V| \log |V|)$ time). The test instances in the M step are determined depending on the hyperparameter S. For each test instance ($n^+_{bag} + n^-_{bag}$ instances for S = S1, $n^+_{bag} * O(step)$ instances for S = S2), the PR function is invoked. Thus the M step completes in $(n^+_{bag} + n^-_{bag}) * (n^+_{bag} + n^-_{bag}) * O(|E| + |V| \log |V|)$ time or $(n^+_{bag} * O(step)) * (n^+_{bag} + n^-_{bag}) * O(|E| + |V| \log |V|)$ time depending on the hyperparameter S. Finally, the DD function is called once more at the end of the main loop. If we represent the maximum number of times the main loop executes with *iteration*, then the EM function has a worst-case time complexity of $n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|) + iteration * (n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|) + (n^+_{bag} + n^-_{bag}) * (n^+_{bag} + n^-_{bag}) * O(|E| + |V| \log |V|) + n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|))$ for S = S1, and $n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|) + iteration * (n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|) + (n^+_{bag} * O(step)) * (n^+_{bag} + n^-_{bag}) * O(|E| + |V| \log |V|) + n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|))$ time for S = S2.

As EMDD-RL picks a single positive bag for seed instances, in total it has a worst-case time complexity of $O(step) * (n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|) + iteration * (n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|) + (n^+_{bag} + n^-_{bag}) * (n^+_{bag} + n^-_{bag}) * O(|E| + |V| \log |V|) + n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|)))$ for S = S1, and $O(step) * (n^+_{bag} * O(step) * O(|E| + |V| \log |V|) + n^-_{bag} * step * O(|E| + |V| \log |V|) + iteration *$

$(n_{bag}^{+} * O(step) * O(|E| + |V| \log |V|) + n_{bag}^{-} * step * O(|E| + |V| \log |V|) + (n_{bag}^{+} * O(step)) * (n_{bag}^{+} + n_{bag}^{-}) * O(|E| + |V| \log |V|) + n_{bag}^{+} * O(step) * O(|E| + |V| \log |V|) + n_{bag}^{-} * step * O(|E| + |V| \log |V|)))$ for S = S2.

Lemma 3.1. *Let the number of positive bags be n_{bag}^{+}, the number of negative bags be n_{bag}^{-}, the step limit imposed in a discrete RL task be step, the edge set and vertex (state) set of the transitioning graph constructed by interacting with the task be E and V, respectively. The SDD algorithm has a worst-case time complexity of $O(n_{bag}^{+} * step) * O(n_{bag}^{+} + n_{bag}^{-}) * O(step) * O(|E| + |V| \log |V|)$ to identify a subgoal.*

Proof. (By construction) Sections 3.1.1, 3.1.2 dissect the SDD algorithm and provide partial and overall time complexity analyses. □

Lemma 3.2. *Let the number of positive bags be n_{bag}^{+}, the number of negative bags be n_{bag}^{-}, the step limit imposed in a discrete RL task be step, the maximum number of iterations in the M step of EMDD-RL be iteration, the edge set and vertex (state) set of the transitioning graph constructed by interacting with the task be E and V, respectively. The EMDD-RL algorithm has a worst-case time complexity of $O(n_{bag}^{+} + n_{bag}^{-} + step) * O(iteration) * O(n_{bag}^{+} + n_{bag}^{-}) * O(step) * O(|E| + |V| \log |V|)$ time for S = S1 and $O(step * n_{bag}^{+}) * O(iteration) * O(n_{bag}^{+} + n_{bag}^{-}) * O(step) * O(|E| + |V| \log |V|)$ time for S = S2 to identify a subgoal.*

Proof. (By construction) Sections 3.1.1 and 3.1.3 dissect the EMDD-RL algorithm and provide partial and overall time complexity analyses. □

4 EXPERIMENTS AND RESULTS

In the previous section, we have introduced various methods of the three categories to identify subgoals in discrete RL tasks. Searching for the best-performing algorithm is an intricate endeavor, since each method makes different assumptions and approaches to the subgoal identification problem from various aspects. Conducting a thorough analysis of all these techniques in various RL tasks entails considerable time and effort. Instead, we have designed and conducted three main experiments to compare the methods.[1]

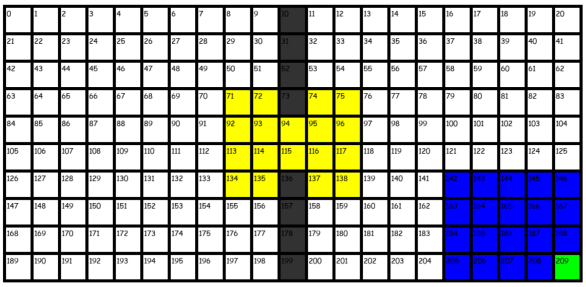
Our first experiment setup concentrates on comparing MIL-based methods. Specifically, it assesses the accuracy and speed capabilities of EMDD-RL and SDD in two RL tasks (two-rooms, four-rooms). In the second experiment setup, we evaluate the methods of the three categories considering their practical hyperparameter search overhead for achieving optimal performance in three tasks (two-rooms, four-rooms, and two-rooms5×). Finally, we compare all the methods in a new challenging RL task called key-room in the third experiment setup, focusing on their subgoal identification accuracies. The following sections provide detailed experiment setups and present the results.

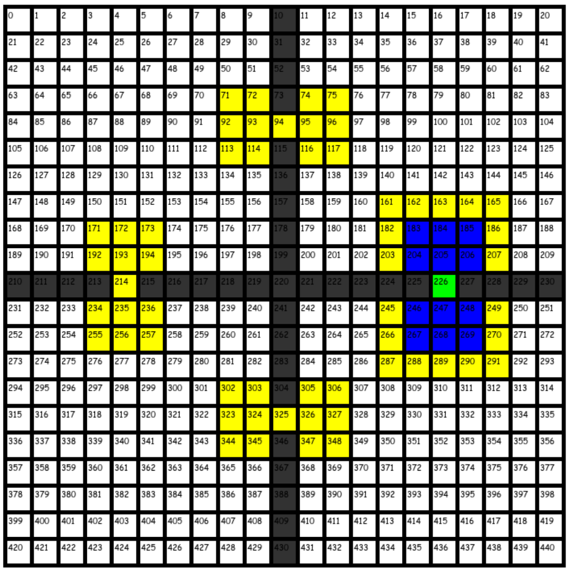
All the RL tasks that we have employed in our experiments are episodic discrete RL tasks. In two-rooms [16] and two-rooms5× environments, an agent begins a new episode in any state of the left room and aims to reach the goal state at the bottom right by passing through a gateway separating the rooms (Figures 3(a) and 3(c)). Similarly, in four-rooms [20], an agent starts in any state in the left-hand side rooms and attempts to reach the goal state between the right-hand side rooms (Figure 3(b)).

4.1 EMDD-RL and SDD Comparison Experiments

In this section, we evaluate EMDD-RL and SDD for the subgoal identification problem with two groups of experiments. The experiments in the first group (referred to as speed evaluation

[1]All datasets and the source code can be found at https://github.com/SaimSUNEL/EMDDRL


(a) Two-rooms Environment


(b) Four-rooms Environment

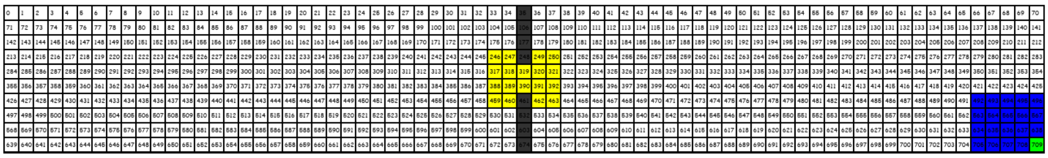

(c) Two-rooms5× Environment

Fig. 3. Two-rooms, two-rooms5×, and four-rooms environments with labeled states. The yellow color indicates the expected subgoal states in the environments. Green-coloured states are goal states. When an algorithm reports a state among the yellow-colored states, it is considered successful. In all environments, blue-colored states around a goal state are discarded in DD calculations for EMDD-RL and SDD to avoid DD value bias toward these states. For the four-rooms, the states surrounding the eliminated states are also strong candidates for subgoal, since positive trajectories have to pass over these states.

experiments) aim to explore which algorithm has a better running time performance. The experiment in the second group (referred to as the accuracy experiment) analyzes how accurate these methods are when the amount of data available for them changes.

EMDD-RL has three hyperparameters. The first hyperparameter (k) is the skipping factor. When k is bigger, the method eliminates more instances from a seed set. If too many instances are eliminated, then the method might miss a region in a state space where a DD score peak occurs during maximization, so this hyperparameter should be tweaked to avoid skipping peaks. The second hyperparameter (*model*) is the selection of the distribution of label probabilities. The third and final hyperparameter (S) is the set of instances checked in the M step. For all experiments, we have tested the following hyperparameter values: $k = \{1,2,3,4\}$, $S = \{\text{S1, S2}\}$, *model* = {linear, exponential}.

Both algorithms require a scaling factor (*sd*) while performing their distance calculations (Equation (4)). Since both work with one-dimensional instances (states), we have set the scaling factor *sd* as 1.0 in the experiments. We eliminate duplicate states in episode histories to reduce computation overhead for both methods while creating positive and negative bags. Depending on whether an agent reaches a goal state or not in a task, each bag gets labeled as positive or negative. For all experiments in this part, we have utilized a transitioning graph extracted directly from the environments (the graph has not been constructed from trajectories) for both algorithms to prevent possible problems that may occur because of incomplete node connections changing from episode to episode, which may affect the actual performances of the algorithms.

To determine the accuracy of a method in identifying subgoals for a task, we count the number of times a method has accurately identified an expected subgoal state in a task and divide it by the total number of subgoal predictions made by the method. The expected subgoal states for two-rooms and four-rooms environments are highlighted with yellow in Figures 3(a) and 3(b).

4.1.1 Speed Evaluation Experiments. Here, we present the experiments conducted to measure the speed performance of EMDD-RL and SDD. These experiments are divided into two parts: speed experiment and state-space effect experiment. In the former, we compare the speed performances of both algorithms, whereas, in the latter, we investigate how the state-space size of an RL task affects the running time (speed) performances of the algorithms.

We have gathered state trajectories from two-rooms and four-rooms environments with two different instance collection strategies for the speed experiment. With the first one, we have not restricted an agent with a step limit in an episode, but with the second one, 200 and 400 step limits are imposed for two-rooms and four-rooms, respectively. In a step-limited episode, if the agent cannot reach a goal state without exceeding the step limit, then the episode is terminated, and a new episode begins. Both step-limitless and step-limited trajectories are gathered from 100 trials. In a trial, the agent starts with zero knowledge about the problem and interacts with it for some episodes to solve it.

For collecting data from both environments, we have made use of a Q-learning [26] agent with the same hyperparameter values as Reference [16] (epsilon (ϵ) = 0.1 learning rate (α) = 0.05, discount factor (γ) = 0.9) and the agent is given four different actions to apply (left, right, up, down) with an action noise (moving 90% probability in an intended direction and 10% probability in any direction). The agent has received a reward of 0 in non-goal states and 1 in a goal state. The eliminated states for static filtering are depicted with blue in Figures 3(a) and 3(b).

We have obtained three datasets for both trajectory collections. From step-limited trajectories, Step-balanced and Step-positive datasets are formed. The Step-balanced dataset consists of a trial's first 20 unsuccessful episodes and the first 20 successful episodes. The Step-positive dataset is constructed with the first 20 successful episodes, and no unsuccessful episode is included. Similarly, the Positive dataset is created from step-limitless trajectories, which contains the first 20 episodes of a trial. EMDD-RL and SDD have been evaluated with these datasets extracted from each trial.

In the state-space effect experiment, we have inspected both algorithms' speed performance as the state space of the two-rooms environment changes. We have designed five two-rooms environment versions with different state-space sizes (rooms are expanded horizontally, and the wall is kept in the middle), and datasets are formed in the same way as the speed experiment.

Table 1. Speed Experiment Results of SDD for Each Dataset of Two-rooms

Dataset	Accuracy (%)	Time (s)
Step-balanced	100	0.6480
Step-positive	100	0.3169
Positive	78	0.6650

Table 2. Speed Experiment Results of SDD for Each Dataset of Four-rooms

Dataset	Accuracy (%)	Time (s)
Step-balanced	89	1.9020
Step-positive	89	0.9129
Positive	88	2.1361

4.1.2 Accuracy Experiment. This experiment aims to explore the accuracy of the algorithms in producing subgoals when the amount of available experience data changes. To this end, the EMDD-RL and SDD algorithms have been run with varying bag sizes. We have utilized the Step-balanced dataset formed in the speed experiment as it is the only dataset that contains negative bags. We have tested positive bag sizes of 5, 10, 15, and 20, while negative bag sizes have been selected from the set of 0, 5, 10, 15, and 20. For every combination of positive and negative bag sizes, we have conducted a hyperparameter search with EMDD-RL.

4.2 Results of EMDD-RL and SDD Comparison Experiments

4.2.1 Speed Evaluation Experiments. This section presents the results of the speed evaluation experiments for each environment separately. For each trial in an environment, we have created three datasets as described, and on every trial, EMDD-RL and SDD algorithms have been run (a total of 100 runs with each dataset for each method). The speed experiment has been repeated ten times for each algorithm to obtain reliable execution time results cleansed from execution oscillations from run to run. The running time results of the state-space effect and accuracy experiment are calculated using 100 runs (1,000 runs in the speed experiment).

In Tables 1 and 3, we present the average execution time results of SDD and EMDD-RL algorithms in the two-rooms environment. For the EMDD-RL algorithm, we present the results of the fastest configuration that attains the same or better accuracy compared to SDD and the configuration that attains the highest accuracy (shaded rows).

The hyperparameter configuration ($model$ = exponential, S = S1) results in the fastest speed performance for the EMDD-RL algorithm across all datasets and k values tested. We have also provided the average loop count in the EM step. Using the exponential model results in the lowest number of loops required for the EM step. The exponential model is superior to the linear model in speed, and S = S1 always attains the shortest execution time against S = S2. Depending on the k value, the algorithm accelerates significantly. It should be neither too big nor too small for fast execution time and accuracy. A value of 2 or 3 is reasonable for this hyperparameter.

When we take the fastest EMDD-RL hyperparameter configurations that attain the same or better accuracy into account, EMMDRL runs 6.30 times faster on the Step-balanced dataset, 5.30 times faster on the Step-positive dataset, 3.37 times faster on the Positive dataset of two-rooms compared to SDD. As for the accuracy performance, the EMDD-RL algorithm performs as well as SDD on Step-balanced and Step-positive datasets; however, on the Positive dataset, EMDD-RL achieves better accuracy.

Tables 2 and 4 present the average running time results of SDD and EMDD-RL algorithms in the four-rooms environment. The observations made for the EMDD-RL hyperparameters with two-rooms are also valid for the results in the four-rooms environment (speed superiority of exponential model and S1, k value effect). When we consider the fastest hyperparameter configurations attaining the same or better accuracy (shaded hyperparameter configurations in the tables), EMDD-RL runs significantly faster on the Step-balanced, Step-positive, and Positive datasets of four-rooms, achieving a speedup of 10.85, 8.69, and 4.89 times, respectively. Additionally, it yields

Table 3. Speed Experiment Results of EMDD-RL for Each Dataset of Two-rooms

Dataset	S	*model*	k	Loop	Accuracy (%)	Time (s)
Step-balanced	S1	EXP	3	11.96	100	0.1028
Step-balanced	S1	EXP	3	11.96	100	0.1028
Step-positive	S1	EXP	3	13.08	100	0.0598
Step-positive	S1	EXP	3	13.08	100	0.0598
Positive	S2	EXP	4	19.06	78	0.1970
Positive	S2	EXP	3	25.09	79	0.2624

The shaded rows present the results of the fastest hyperparameter configurations that achieve the same or higher accuracy compared to SDD. Non-shaded rows show the results of the hyperparameter configurations with the highest accuracy.

Table 4. Speed Experiment Results of EMDD-RL for Each Dataset of Four-rooms

Dataset	S	*model*	k	Loop	Accuracy (%)	Time (s)
Step-balanced	S1	EXP	3	14.73	90	0.1753
Step-balanced	S1	EXP	2	21.42	92	0.2546
Step-positive	S1	EXP	3	46.40	91	0.1051
Step-positive	S2	EXP	1	46.40	94	0.5580
Positive	S1	EXP	3	41.89	88	0.4370
Positive	S1	EXP	3	41.89	88	0.4370

The shaded rows present the results of the fastest hyperparameter configurations that achieve the same or higher accuracy compared to SDD. Non-shaded rows show the results of the hyperparameter configurations with the highest accuracy.

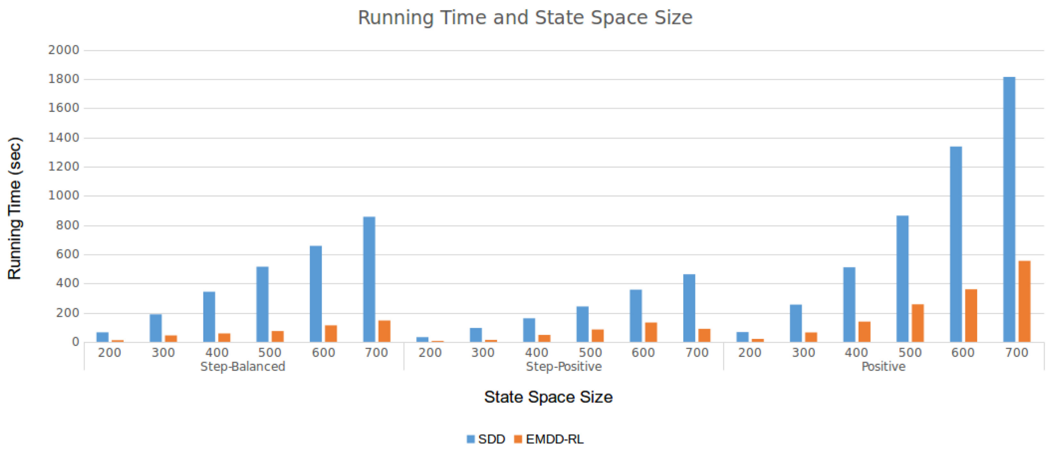


Fig. 4. Barchart graph of execution times of the SDD and EMDD-RL algorithms in two-rooms environments with different state-space sizes. The charts are partitioned with respect to datasets. The running time axis shows the total duration of 100 runs of EMDD-RL and SDD on a particular dataset formed with a particular state-space size. The EMDD-RL hyperparameter configurations whose running time results are shown here achieve the same or better accuracy performance compared to SDD.

more accurate results on Step-balanced and Step-positive datasets, whereas both algorithms perform equally well on the Positive dataset.

For the state-space effect experiment, both algorithms have been evaluated using datasets formed from two-rooms environment versions with 210, 310, 410, 510, 610, and 710 states. The step limits imposed are proportional to the state-space sizes and are as follows: 200, 300, 400, 500, 600, and 700. Figure 4 presents the speed performance results of the algorithms. As the state space of the environment expands, the speed performance gap between the algorithms enlarges. The impact of state-space growth on EMDD-RL is notably less than on the SDD algorithm, which makes EMDD-RL a promising choice for environments with large state space.

4.2.2 Accuracy Experiment. Tables 5 and 6 show the performance results of the algorithms in the accuracy experiment. For each bag size pair in the tables, we have considered the result of the EMDD-RL hyperparameter configuration that achieves the highest accuracy score and fastest execution time. The Time columns depict the average execution time of the algorithms for 100 runs.

The table rows are colored based on the performance results of EMDD-RL (green: EMDD-RL outperforms; no color: both perform equally well; red: SDD outperforms). In the two-rooms

Table 5. Results of the Accuracy Experiment in Two-rooms

Alg	Pstv	Ngtv	S	*model*	k	Accuracy (%)	Time
SDD	5	0				92	0.0724
EMDD-RL			S2	EXP	2	86	0.0740
SDD	5	5				70	0.1365
EMDD-RL			S2	EXP	4	81	0.0688
SDD	5	10				69	0.1971
EMDD-RL			S1	LIN	4	78	0.0854
SDD	5	15				60	0.2618
EMDD-RL			S2	EXP	4	76	0.1263
SDD	5	20				57	0.3271
EMDD-RL			S2	EXP	3	66	0.1658
SDD	10	0				98	0.1591
EMDD-RL			S1	EXP	2	99	0.0687
SDD	10	5				97	0.2268
EMDD-RL			S2	EXP	3	97	0.0984
SDD	10	10				96	0.3039
EMDD-RL			S2	EXP	3	96	0.1245
SDD	10	15				93	0.3939
EMDD-RL			S1	EXP	3	95	0.0849
SDD	10	20				93	0.4649
EMDD-RL			S1	EXP	1	93	0.3054
SDD	15	0				100	0.2386
EMDD-RL			S1	EXP	2	100	0.0792
SDD	15	5				99	0.3123
EMDD-RL			S1	EXP	4	99	0.0575
SDD	15	10				98	0.3895
EMDD-RL			S1	EXP	1	99	0.2194
SDD	15	15				98	0.4711
EMDD-RL			S1	EXP	1	99	0.2492
SDD	15	20				98	0.5564
EMDD-RL			S1	EXP	2	98	0.1553
SDD	20	0				100	0.3198
EMDD-RL			S1	EXP	3	100	0.0588
SDD	20	5				100	0.3906
EMDD-RL			S1	EXP	4	100	0.0639
SDD	20	10				100	0.4823
EMDD-RL			S1	EXP	3	100	0.0790
SDD	20	15				100	0.5792
EMDD-RL			S1	EXP	3	100	0.0913
SDD	20	20				100	0.6416
EMDD-RL			S1	EXP	3	100	0.1040

The green color specifies the bag size combinations that result in better accuracy for EMDD-RL, while no color is used if EMDD-RL and SDD [16] have identical accuracy performance. However, if SDD performs better than EMDD-RL, then it is indicated with red color. The unit of the Time column is second. For the two-rooms, in most bag size combinations, the EMDD-RL algorithm excels in accuracy performance or performs as well as SDD.

Table 6. Results of the Accuracy Experiment in Four-rooms

Alg	Pstv	Ngtv	S	*model*	k	Accuracy (%)	Time
SDD	5	0				57	0.1865
EMDD-RL			S1	LIN	4	63	0.0529
SDD	5	5				56	0.3678
EMDD-RL			S1	EXP	4	72	0.0836
SDD	5	10				53	0.5455
EMDD-RL			S2	LIN	4	70	0.4752
SDD	5	15				52	0.6933
EMDD-RL			S2	EXP	3	65	0.2796
SDD	5	20				55	0.8511
EMDD-RL			S2	EXP	3	66	0.3285
SDD	10	0				83	0.4402
EMDD-RL			S2	EXP	3	86	0.1626
SDD	10	5				83	0.6784
EMDD-RL			S1	EXP	3	87	0.1124
SDD	10	10				80	0.9291
EMDD-RL			S1	EXP	4	84	0.1215
SDD	10	15				84	1.1165
EMDD-RL			S2	EXP	4	87	0.2976
SDD	10	20				85	1.3538
EMDD-RL			S2	EXP	4	87	0.3416
SDD	15	0				85	0.7088
EMDD-RL			S1	LIN	4	87	0.0971
SDD	15	5				85	0.9252
EMDD-RL			S1	EXP	2	90	0.1683
SDD	15	10				85	1.2187
EMDD-RL			S1	EXP	3	91	0.1404
SDD	15	15				84	1.4033
EMDD-RL			S1	EXP	2	91	0.2291
SDD	15	20				82	1.6698
EMDD-RL			S2	EXP	1	91	1.1150
SDD	20	0				89	0.9411
EMDD-RL			S2	EXP	1	94	0.5626
SDD	20	5				90	1.1590
EMDD-RL			S1	EXP	1	92	0.3197
SDD	20	10				90	1.4012
EMDD-RL			S1	EXP	1	94	0.3754
SDD	20	15				91	1.6837
EMDD-RL			S2	EXP	3	93	0.3392
SDD	20	20				89	1.9210
EMDD-RL			S1	EXP	2	92	0.2575

The green color specifies the bag size combinations that result in better accuracy for EMDD-RL, while no color is used if EMDD-RL and SDD [16] have identical accuracy performance. However, if SDD performs better than EMDD-RL, then it is indicated with red color. The unit of the Time column is second. For the four-rooms environment, in all bag size combinations, the EMDD-RL algorithm excels in accuracy performance.

environment results, for most of the bag size combinations, EMDD-RL achieves better or similar accuracy performance (for only one case, 5 positive bags and 0 negative bags, it fails). However, it outperforms SDD in the four-rooms environment results. For some size pairs (e.g., 5 positive bags and 0 negative bags), the SDD algorithm runs faster than EMDD-RL. When dealing with small datasets, calculating individual DD scores of instances can become more efficient than going through the EM loop, which can result in increased computation overhead for EMDD-RL.

The results of the EMDD-RL and SDD comparison experiments indicate that EMDD-RL outperforms SDD in terms of speed and accuracy for the subgoal identification problem.

4.3 Hyperparameter Search Experiments

In this part, we present two experiments that empirically assess how strongly the accuracy performances of the graph-based, statistics-based, and MIL-based algorithms depend on their

Table 7. Methods and Their Hyperparameter Values Tested in Two-rooms, Four-rooms, and Two-rooms5× Environments for the First Hyperparameter Search Experiment

Method	Hyperparameters and Values					
Betweenness [23]	t_s		t_o			t_p
	100,200		5,10,15			0.15,0.20,0.25,0.30
L-Cut [4]	h	t_c		t_o		t_p
	100,140,180	0.10,0.15		10		0.10,0.15,0.20
Q-Cut [17]	t_s				t_c	
	50,100,150,200				500,1000,2000,3000,4000	
RN [3]	t_{RN}	q	p	$\lambda_{fa}/\lambda_{miss}$	$p(N)/P(T)$	l_n
	1.50,1.70,1.90	0.0056	0.0712,0.0700	50.0,100.0,150.0	100.0,150.0	7,10,15
Segmented Q-Cut [17]	t_s		t_c			t_d
	100,150		80,100,120,140,160			10,15
SCC [12]	Environments	t_t				
	two-rooms	2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26				
	four-rooms	2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,36,37,39				
	two-rooms5×	2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,35				
SDD [16]	—					
EMDD-RL	S		$model$			k
	S1,S2		linear, exponential			1,2,3,4

For each algorithm, every combination of their hyperparameter values has been tested. For the SCC algorithm, hyperparameter values are determined by inspecting trajectories for each environment.

hyperparameter values. With the first experiment, we investigate whether a predetermined set of hyperparameter values could be employed for different tasks to obtain optimal accuracy performance. To this end, we evaluate a predetermined set of hyperparameter values for each algorithm in three RL tasks (two-rooms, four-rooms, two-rooms5×). In the second experiment, we inspect the impact of minor changes in hyperparameter values on accuracy performance using only the two-rooms environment. When minor changes do not result in significant performance fluctuations, a hyperparameter search procedure can be guided by focusing on the hyperparameter values in the vicinity of values that yield promising results. Thus, search effort can be reduced. In the case where performance fluctuations occur, checking many hyperparameter values may become necessary to identify a configuration that yields satisfactory results, which engenders increased effort.

In both experiments, we have utilized the Step-balanced dataset of the speed experiment for all algorithms (datasets are created from two-rooms5× in a similar manner). The hyperparameter values used for the algorithms in each experiment are presented in Tables 7 and 8. For EMDD-RL and SDD, transitioning graphs are constructed with trajectories.

The SCC algorithm stands out among others concerning its hyperparameter value selection strategy. The method determines its hyperparameter value dynamically by inspecting its transitioning graph. When the agent follows a different set of trajectories, the candidate hyperparameter values will likely change. So, for fairness, in the experiments, we have considered hyperparameter values extracted after inspecting trajectories for each environment. Different from other methods, in this respect, the SCC algorithm does not consider the same set of hyperparameter values for two-rooms, four-rooms, and two-rooms5× environments.

4.4 Results of Hyperparameter Search Experiments

Tables 9 and 10 show results for the hyperparameter search experiments. Table 9 presents several statistical measure scores calculated with all hyperparameter configuration results for each method. Figure 5 summarizes the table as a bar graph (EMDD-RL and SDD are omitted). The mean accuracy values are illustrated with bars, while the standard deviation of accuracy values is shown with lines on the bars. Although the same hyperparameter values are used for each method (except the SCC method, but tested value sets are very close to each other) in all environments, the

Table 8. Methods and Their Hyperparameter Values Tested Solely in the Two-rooms Environment for the Second Hyperparameter Search Experiment

Method	Hyperparameters and Values					
Betweenness [23]	t_s: 100,120,140,200	t_o: 5,10,15	t_p: 0.10,0.15,0.20,0.25,0.30			
L-Cut [4]	h: 100,120,140,180	t_c: 0.10,0.15	t_o: 5,10,15	t_p: 0.10,0.15,0.20		
Q-Cut [17]	t_s: 20,50,80,100,150,200	t_c: 100,500,1000,2000,3000,4000				
RN [3]	t_{RN}: 0.5,0.7,1.0,1.50,1.70,1.90	q: 0.0056	p: 0.0712,0.0700	$\lambda_{fa}/\lambda_{miss}$: 50.0,100.0,150.0	$p(N)/P(T)$: 100.0,150.0	l_n: 5,7,10,12,15
Segmented Q-Cut [17]	t_s: 20,30,50,80,100,150	t_c: 80,100,120,140,160	t_d: 4,6,8			
SCC [12]	t_t: 2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26					
SDD [16]	—					
EMDD-RL	S: S1,S2	*model*: linear, exponential	k: 1,2,3,4			

For each algorithm, every combination of their hyperparameter values has been tested.

attained results are significantly different. Standard deviations are high for every method except Segmented Q-Cut and EMDD-RL (Segment Q-Cut has failed to achieve satisfactory results in both experiments). Yielding significantly different results in different environments with the same hyperparameter values shows that the graph-based and statistics-based methods strongly depend on their hyperparameter values, and different sets of hyperparameter values need to be tested for every new environment to achieve reliable subgoal identification accuracy performance. EMDD-RL yields more stable results with varying hyperparameter configurations compared to the other methods.

Table 10 presents the accuracy results of several hyperparameter configurations in the second experiment. The "Number of predictions" column provides the total number of subgoals reported by a particular hyperparameter configuration of a method. While the hyperparameter configurations are similar, the accuracy and number of subgoal predictions can vary greatly. In other words, small perturbations in hyperparameter values can culminate in significant performance changes. EMDD-RL is less affected by this phenomenon than the graph-based and statistics-based techniques.

4.5 Key-room Environment Experiment

In this section, we introduce a new discrete (episodic) problem called key-room (Figure 6) and the experiment conducted with this environment employing all the methods, referred to as the key-room environment experiment. In the RL tasks of the previous sections (two-rooms, four-rooms, two-rooms5×), it has been straightforward to identify subgoals as they correspond to gateways between rooms. The key-room environment introduces a more complex problem where an agent has to visit a particular state (state 4, referred to as the key state) before reaching the goal state to solve the problem. Naturally, this single state becomes a subgoal for the agent. Identifying such a subgoal is more challenging for the graph-based and statistics-based methods, because it violates assumptions of the methods (e.g., it does not reside between strongly connected state regions or a new state-space region is not explored after visiting it). The algorithms must be capable of inferring that the key state is vital for the agent's success.

In the problem, an agent is given four different actions to apply (left, right, up, down) with an action noise (moving 90% probability in an intended direction and 10% probability in any direction). The agent receives a reward value of −0.001 in non-goal states and 5 or −5 in the goal

Table 9. Accuracy Performance Results of the First Hyperparameter Search Experiment

Method	Environment	Accuracy (%) (Mean)	Accuracy (%) (Standard Deviation)	Accuracy (%) (95% Confidence Interval)
L-Cut [4]	two-rooms	26.42	15.39	[18.54, 34.29]
	four-rooms	21.06	4.57	[18.73, 23.40]
	two-rooms5×	17.26	3.79	[15.32, 19.20]
Q-Cut [17]	two-rooms	28.78	42.51	[8.37, 49.19]
	four-rooms	93.08	6.65	[89.88, 96.27]
	two-rooms5×	72.66	14.65	[65.63, 79.70]
Segmented Q-Cut [17]	two-rooms	29.79	8.38	[25.76, 33.81]
	four-rooms	33.17	5.68	[30.44, 35.90]
	two-rooms5×	6.19	1.49	[5.47, 6.90]
RN [3]	two-rooms	37.20	32.16	[31.04, 43.37]
	four-rooms	57.46	20.16	[53.60, 61.32]
	two-rooms5×	30.35	12.73	[27.91, 32.79]
Betweenness [23]	two-rooms	11.52	20.82	[2.54, 20.50]
	four-rooms	20.85	25.74	[9.75, 31.95]
	two-rooms5×	4.86	7.55	[1.60, 8.11]
SCC [12]	two-rooms	53.27	33.16	[39.30, 67.24]
	four-rooms	31.75	30.12	[21.41, 42.08]
	two-rooms5×	19.07	17.54	[12.65, 25.50]
SDD [16]	two-rooms	100.00	0.00	[–, –]
	four-rooms	89.00	0.00	[–, –]
	two-rooms5×	98.00	0.00	[–, –]
EMDD-RL	two-rooms	99.56	0.007	[99.17, 99.95]
	four-rooms	85.87	0.06	[82.17, 89.57]
	two-rooms5×	97.87	0.006	[97.54, 98.20]

Several statistical measures are calculated from the accuracy scores of all hyperparameter configurations of a particular method. For the graph-based and statistics-based methods, results can significantly vary when the same hyperparameter values are utilized in different problems.

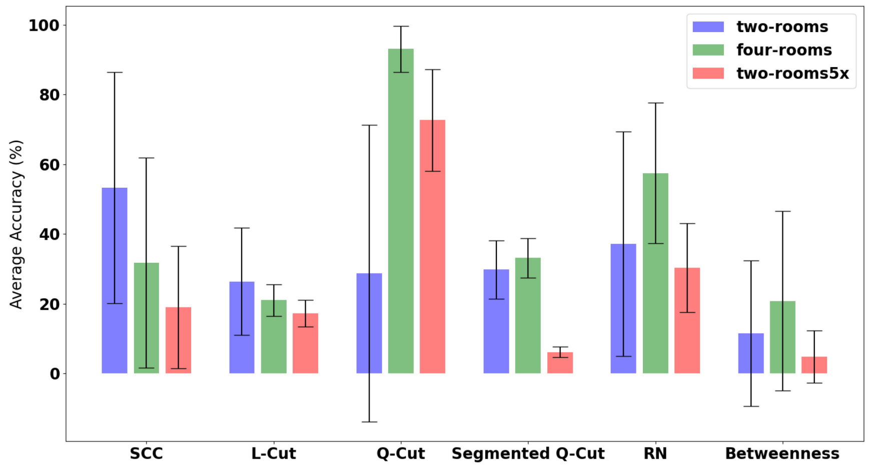


Fig. 5. Barchart graph of the results in Table 9. Average accuracy scores are illustrated with bars. The lines on the bars indicate the standard deviation of the results for a particular method.

Table 10. Accuracy Performance Results of Several Hyperparameter Configurations in the Second Experiment

Algorithm			t_c	t_o	t_p	h	Number of predictions	Accuracy (%)
L-Cut [4]			0.10	15	0.10	140	22	31.82
			0.10	15	0.10	180	7	0
			0.10	15	0.15	100	8	62.5
			0.10	15	0.15	120	11	54.55
			0.10	15	0.15	140	2	50
			0.10	15	0.15	180	2	0
			0.10	15	0.20	100	3	66.67
			0.10	15	0.20	120	4	25
			0.10	15	0.20	140	1	100
			0.10	15	0.20	180	0	0
					t_s	t_c	Number of predictions	Accuracy (%)
Q-Cut [17]					20	100	16818	0.05
					20	500	298	2.68
					20	1000	17	35.29
					20	2000	4	100
					20	3000	4	100
					50	100	2843	0.04
					50	500	327	0.31
						t_t	Number of predictions	Accuracy (%)
SCC [12]						15	25	28.0
						16	14	50.0
						17	10	30.0
						18	12	58.33
						19	15	86.67
						20	12	100.0
				t_s	t_c	t_d	Number of predictions	Accuracy (%)
Segmented Q-Cut [17]				20	80	4	116	31.03
				20	80	6	117	21.37
				20	80	8	211	18.48
				20	80	10	153	17.65
				20	80	15	147	27.89
				20	100	4	78	28.21
				20	100	6	86	24.42
				20	100	8	164	22.56
				20	100	10	138	18.84
	t_{RN}	q	p	$\lambda_{fa}/\lambda_{miss}$	$p(N)/p(T)$	l_n	Number of predictions	Accuracy (%)
RN [3]	0.5	56	0.0712	50.0	100.0	5	486,209	14.37
	0.5	56	0.0712	50.0	100.0	7	470,401	14.53
	0.5	56	0.0712	50.0	100.0	10	446,698	14.7
	1.70	56	0.0712	100.0	100.0	12	49	97.96
	1.70	56	0.0712	100.0	100.0	15	2	50
	1.70	56	0.0712	100.0	150.0	5	61	72.13
	1.70	56	0.0712	100.0	150.0	7	48	47.92
	1.70	56	0.0712	100.0	150.0	10	37	86.49
	1.70	56	0.0712	100.0	150.0	12	40	100
	1.70	56	0.0712	100.0	150.0	15	0	0
	1.70	56	0.0712	150.0	100.0	5	61	72.13
				t_o	t_p	t_s	Number of predictions	Accuracy (%)
Betweenness [23]				5	0.10	100	786	20.87
				5	0.10	120	579	23.14
				5	0.10	140	412	23.3
				5	0.10	200	53	0
				5	0.15	100	271	21.4
				10	0.15	120	21	33.33
				10	0.15	140	13	38.46
				10	0.15	200	0	0
				10	0.20	100	3	66.67
				10	0.20	120	2	100
				10	0.20	140	2	100
				S	$model$	k	Number of predictions	Accuracy (%)
EMDD-RL				S1	LIN	1	100	100
				S1	LIN	2	100	100
				S1	LIN	3	100	99
				S1	LIN	4	100	99
				S1	EXP	1	100	100
				S1	EXP	2	100	100
				S2	EXP	2	100	100
				S2	EXP	4	100	98

Even though the values are close to each other in hyperparameter space for the graph-based and statistics-based methods, their results can differ significantly.

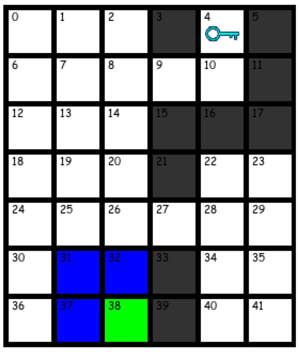


Fig. 6. Key-room environment. A key is located at state 4. The goal state is state 38. To get a positive reward, the agent must visit state 4 (take the key) before reaching the goal state. The blue-colored states are removed for static filtering.

state, depending on whether it has visited the key state. For each new episode, the agent starts in one of the following states randomly: {0,1,2,6,7,8,12,13,14}.

With these characteristics, the key-room environment differs from the other tasks introduced previously in that the environment hides the information about whether the key state is visited or not from the agent. It only provides state information to the agent. Thus, it embodies hidden-state information, which renders it a **partially observable MDP (POMDP)** [29] problem. The POMDP framework extends MDP with two additional components: Ω and O, where Ω is the set of observations and O observation probability function. In a POMDP problem, an agent is provided with an observation instead of actual state information. POMDP generalizes MDP, and it is more capable of representing real-world problems.

We have collected experiment data and performed a hyperparameter search for each method to compare the algorithms in this environment, similar to the experiments with two-rooms and four-rooms environments. The same data gathering and data set creation steps in Section 4.1.1 have been followed (100 trials, Step-balanced dataset, 20 positive bags, 20 negative bags). We have not imposed a step limit, and trajectories have been labeled depending on whether the agent has visited state 4 before reaching the goal state. Transitioning graphs for EMDD-RL and SDD are constructed from trajectories. We have tested all the hyperparameter values in Table 8, but we have also considered additional hyperparameter values for several algorithms, because the environment features fewer states compared to two-rooms (Betweenness: $t_s = \{10, 20, 40, 50, 100, 120, 140, 180, 200\}$, L-Cut: $h = \{10, 20, 40, 50, 100, 120, 140, 180, 200\}$, Segmented Q-Cut: $t_s = \{10, 20, 30, 40, 50, 80, 100, 150\}$, $t_d = \{4, 6, 8, 10, 15\}$, Q-Cut: $t_s = \{5, 10, 20, 50, 80, 100, 150, 200\}$,$t_c = \{20, 40, 60, 80, 100, 500, 1,000, 2,000, 3,000, 4,000\}$, SCC: $\{2–105\}$).

4.6 Results of Key-room Environment Experiment

Table 11 summarizes the accuracy results of the methods in the key-room environment. The MIL-based methods significantly outperform the graph-based and statistics-based methods in identifying the key state as a subgoal. The RN and Betweenness methods can identify the key state better than the other graph-based methods, but they have picked almost every state as a subgoal.

5 DISCUSSION

When dealing with the subgoal identification problem, the following aspects stand out for methods: accuracy and speed. The main goal is to identify subgoals as accurately as possible, but

Table 11. Accuracy Performance Results of the Key-room Environment Experiment

Method	Accuracy (%)	Number of predictions
L-Cut [4]	0	832.96
Q-Cut [17]	0	1,340.12
SCC [12]	0	108
Segmented Q-Cut [17]	0	18.00
RN [3]	1.42	349,819
Betweenness [23]	0.24	1,248
SDD [16]	46	100
EMDD-RL	52	100

For the methods attaining an accuracy score of 0, the "Number of predictions" column averages the number of subgoals reported by all hyperparameter configurations.

computation demand must be taken into account for practicality. Computation demand can emerge due to an algorithm itself or the hyperparameter search it requires (in the worst case, due to both).

In the previous section, we have compared the methods of the three categories by basing our experiment setups on the following questions: (1) As EMDD-RL is a new MIL-based method, how does it compare to SDD regarding accuracy and speed? (2) How strongly do the methods depend on their hyperparameter values concerning accuracy performance? (3) How do the graph-based and statistics-based methods perform in a problem where their subgoal assumptions are violated?

For the first question, we have conducted three experiments (speed, state-space effect, and accuracy) using three environments. When all the results of these experiments are considered, EMDD-RL outperforms SDD in terms of speed and accuracy. In addition, its computation demand scales better with the state-space size of a task than SDD. Unlike SDD, EMDD-RL requires three hyperparameters ($model$, S, k). Concerning accuracy performance, EMDD-RL does not possess a dominant hyperparameter configuration. However, as for speed performance, the configuration ($model$ = exponential, S = S1) has yielded the shortest execution time in the experiments. This setting can be employed if an agent demands a subgoal in a shorter time. Assigning S to S2 has degraded the speed performance compared to S = S1; however, it has delivered more accurate results on several datasets. So, a search over the hyperparameters must be performed if the agent aims to identify subgoals more accurately. As default, the configuration ($model$ = exponential,S = S1, k = 2) could be considered for EMDD-RL.

Regarding the second question, two experiments (hyperparameter search experiments) have been designed. With the first one, predefined hyperparameter configurations for algorithms have been tested against their accuracy performance in three tasks. In the second, results of relatively close hyperparameter configurations have been inspected. The results indicate that the graph-based and statistics-based approaches strongly depend on their hyperparameters, and a new hyperparameter search with possibly different hyperparameter values should be carried out in a new task. However, EMDD-RL attains more stable results with varying hyperparameter configurations.

The number of hyperparameters and their value sets describe the hyperparameter configuration space of an algorithm. When there is no guideline for hyperparameter value selection, an exhaustive search is needed over the configuration space of the algorithm. Performing a search over a finite value set is relatively simple, since the values to be considered are known in advance. However, a serious problem arises with hyperparameters whose values are real-valued (continuous). Since there are infinitely many possible values, some groups of values or ranges should be preferred to others. Unfortunately, without actually trying and inspecting the outcomes, it is not easy to know whether selected values yield the best results or whether there is a better configuration that might have been excluded during selection. With this regard, EMMD-RL and SCC facilitate hyperparameter tuning as they require several hyperparameters whose values can be picked from a finite set. In contrast, the other techniques require either real-valued hyperparameters or hyperparameters with many discrete values (e.g., integers). Furthermore, in most studies, the authors leave a well-defined procedure for hyperparameter value selection undescribed.

As for the third question, we have examined the accuracy performances of the methods in the key-room environment. The environment challenges the algorithms by introducing an unconventional subgoal, unlike the other environments in the experiments. Its subgoal (the key state) violates the assumptions made by the methods. It is neither located near a gateway state nor facilitates an agent to transition to a new state-space region, so the graph-based and statistics-based methods have failed to detect it as a subgoal. Different from these methods, SDD and EMDD-RL have achieved much more accurate results. It is thanks to the fact that SDD and EMDD-RL have their origins in the MIL paradigm. Since trajectories are classified as positive or negative depending on a success criterion, both algorithms extract the most helpful information from a dataset to differentiate positive and negative bags. When employed for an RL task, the algorithms intrinsically form a notion of what is useful or not for an agent in the task, an important property that the other subgoal identification methods lack. Such a property is necessary for identifying the subgoal of the key-room environment.

When all these results are considered, for the subgoal identification problem, the MIL-based methods could be preferred to the other techniques as they require fewer hyperparameters and intrinsically have a notion of what might be helpful for an agent to achieve its objective.

Despite their merits, the MIL-based subgoal identification algorithms can be further improved. A prominent disadvantage is that (although it is not due to the nature of algorithms) they require particular states to be eliminated from trajectory histories when applied to discrete RL tasks. In this study, we have manually fed these states to the algorithms. However, these states can be chosen automatically by inspecting trajectory histories. For instance, a certain percentage of a trajectory history could be considered for static filtering. This ratio can be determined by examining various discrete RL tasks.

It is apparent that the amount of experience data directly affects the subgoal identification performance of all the methods. Reliable subgoal identification and policy learning entail gathering sufficient data. In this study for the experiments, we have considered trajectory histories of 40 episodes (20 positive bags, 20 negative bags) for all methods. We have inspired the number of positive bags (20) from the SDD study [16], where the authors use the experience data of 20 to 30 episodes for subgoal identification. With the considered experience data amount, the methods have identified reliable subgoals with a suitable hyperparameter configuration, though the accuracy of some methods is low. However, more than 40 trajectory histories may be required in tasks with large state spaces. In general, it can be challenging to ascertain how many episodes should be considered for data gathering beforehand. As a solution, it can be regarded as a hyperparameter for all subgoal identification methods.

6 CONCLUSION AND FUTURE WORK

In the reinforcement learning literature, subgoal identification refers to discovering critical states for decomposing a task. In this work, first, we classify prominent subgoal identification methods for discrete RL tasks into the following three categories: graph-based, statistics-based, and MIL-based. Second, we present a novel MIL-based method called EMDD-RL; third, we thoroughly compare the methods of these categories with three main experiment setups. EMDD-RL adapts a MIL method called EMDD for subgoal identification in discrete RL tasks. It has achieved better accuracy and speed performance results than another MIL-based approach called SDD in the experiments. Our experiment results suggest that MIL-based approaches could be preferable to the methods in the other categories for subgoal identification in practice. They feature fewer hyperparameters, which facilitate hyperparameter tuning, and have a better knowledge extraction capability toward what may be helpful for an agent to achieve its goal.

Our method can be improved and employed as a new method in other research areas. Among all the algorithmic steps of EMDD-RL, a natural improvement could take place in the algorithm's seed set selection part. EMDD-RL populates the initial seed set with the instances of the shortest positive trajectory and eliminates some of them with a simple strategy. A better mechanism that shrinks the set further without missing possible subgoal candidates can speed up EMDD-RL more. Different from this direction, EMDD-RL could be introduced to the partially observable MDP (POMDP) research area as a new subgoal identification method for POMDP problems. Although in this study we have considered only discrete RL problems, after making some modifications, both SDD and EMDD-RL methods could be applied to RL problems with continuous state space (where the application of the other methods mentioned in this study is impractical) for subgoal identification, since both originate from the MIL domain, which is another important research direction that can be considered.

REFERENCES

[1] Hüseyin Aydın, Erkin Çilden, and Faruk Polat. 2022. Using chains of bottleneck transitions to decompose and solve reinforcement learning tasks with hidden states. *Future Gen. Comput. Syst.* 133 (2022), 153–168. https://doi.org/10.1016/j.future.2022.03.016

[2] Akhil Bagaria and George Konidaris. 2019. Option discovery using deep skill chaining. In *Proceedings of the International Conference on Learning Representations.*

[3] Özgür Şimşek and Andrew G. Barto. 2004. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning (ICML'04).* Association for Computing Machinery, New York, NY, 95. https://doi.org/10.1145/1015330.1015353

[4] Özgür Şimşek, Alicia P. Wolfe, and Andrew G. Barto. 2005. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd International Conference on Machine Learning (ICML'05).* Association for Computing Machinery, New York, NY, 816–823. https://doi.org/10.1145/1102351.1102454

[5] Michael Dann, Fabio Zambetta, and John Thangarajah. 2019. Deriving subgoals autonomously to accelerate learning in sparse reward domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 881–889.

[6] Thomas G. Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *J. Artif. Int. Res.* 13, 1 (Nov. 2000), 227–303.

[7] Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. 1997. Solving the multiple instance problem with axis-parallel rectangles. *Artific. Intell.* 89, 1 (1997), 31–71. https://doi.org/10.1016/S0004-3702(96)00034-3

[8] E. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.

[9] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34, 3 (July 1987), 596–615. https://doi.org/10.1145/28869.28874

[10] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2017. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence.* Retrieved from https://api.semanticscholar.org/CorpusID:4674781

[11] Glenn A. Iba. 1989. A heuristic approach to the discovery of macro-operators. *Mach. Learn.* 3, 4 (1989), 285–317.

[12] Seyed Jalal Kazemitabar and Hamid Beigy. 2008. Automatic discovery of subgoals in reinforcement learning using strongly connected components. In *Advances in Neuro-Information Processing*, Mario Köppen, Nikola Kasabov, and George Coghill (Eds.). Vol. 5506. 829–834. https://doi.org/10.1007/978-3-642-02490-0_101

[13] R. Kretchmar, Todd Feil, and Rohit Bansal. 2003. Improved automatic discovery of subgoals for options in hierarchical reinforcement learning. *Journal of Computer Science & Technology* 3, 2 (2003), 9–14.

[14] Oded Maron and Tomás Lozano-Pérez. 1998. A framework for multiple-instance learning. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS'97).* MIT Press, Cambridge, MA, 570–576.

[15] Oded Maron and Tomas Lozano-Perez. 1998. *Learning from Ambiguity.* Ph.D. Dissertation. AAI0599603.

[16] Amy McGovern and Andrew G. Barto. 2001. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the 18th International Conference on Machine Learning (ICML'01).* Morgan Kaufmann Publishers, San Francisco, CA, 361–368.

[17] Ishai Menache, Shie Mannor, and Nahum Shimkin. 2002. Q-cut—Dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning (ECML'02)*. Springer-Verlag, Berlin, 295–306.

[18] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. 2021. Hierarchical reinforcement learning: A comprehensive survey. *ACM Comput. Surv.* 54, 5, Article 109 (June 2021), 35 pages. https://doi.org/10.1145/3453160

[19] Sujoy Paul, Jeroen Vanbaar, and Amit Roy-Chowdhury. 2019. Learning from trajectories via subgoal discovery. *Adv. Neural Info. Process. Syst.* 32 (2019).

[20] Doina Precup. 2000. *Temporal abstraction in reinforcement learning*. Ph.D. Dissertation.

[21] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. 2019. Tunability: Importance of hyperparameters of machine learning algorithms. *J. Mach. Learn. Res.* 20, 53 (2019), 1–32. Retrieved from http://jmlr.org/papers/v20/18-444.html

[22] G. Rummery and Mahesan Niranjan. 1994. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166.

[23] O. Simsek and A. G. Barto. 2009. Skill characterization based on betweenness. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 1497–1504.

[24] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction.* A Bradford Book, Cambridge, MA.

[25] Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.* 112, 1–2 (Aug. 1999), 181–211. https://doi.org/10.1016/S0004-3702(99)00052-1

[26] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Mach. Learn.* 8, 3 (May 1992), 279–292. https://doi.org/10.1007/BF00992698

[27] Marco Wiering and Jürgen Schmidhuber. 1997. HQ-learning. *Adaptive Behavior* 6, 2 (1997), 219–246. https://doi.org/10.1177/105971239700600202 arXiv:https://doi.org/10.1177/105971239700600202

[28] Qi Zhang and Sally A. Goldman. 2002. EM-DD: An improved multiple-instance learning technique. In *Advances in Neural Information Processing Systems 14*, T. G. Dietterich, S. Becker, and Z. Ghahramani (Eds.). MIT Press, 1073–1080.

[29] K. J. Åström. 1965. Optimal control of Markov processes with incomplete state information. *J. Math. Anal. Appl.* 10, 1 (1965), 174–205. https://doi.org/10.1016/0022-247X(65)90154-X

Received 7 February 2023; revised 31 August 2023; accepted 20 January 2024