The VLDB Journal (2025) 34:41
https://doi.org/10.1007/s00778-025-00920-0

REGULAR PAPER

Model reusability in Reinforcement Learning

Sepideh Nikookar[1,2] · Sohrab Namazi Nia[1] · Senjuti Basu Roy[1] · Sihem Amer-Yahia[3] · Behrooz Omidvar-Tehrani[4]

Received: 5 May 2024 / Revised: 27 January 2025 / Accepted: 4 April 2025

Abstract
The ability to reuse trained models in Reinforcement Learning (RL) holds substantial practical value in particular for complex tasks. While model reusability is widely studied for supervised models in data management, to the best of our knowledge, this is the first ever principled study that is proposed for RL. To capture trained policies, we develop a framework based on an expressive and lossless graph data model that accommodates Temporal Difference Learning and Deep-RL based RL algorithms. Our framework is able to capture arbitrary reward functions that can be composed at inference time. The framework comes with theoretical guarantees and shows that it yields the same result as policies trained from scratch. We design a parameterized algorithm that strikes a balance between efficiency and quality w.r.t cumulative reward. Our experiments with two common RL tasks (query refinement and robot movement) corroborate our theory and show the effectiveness and efficiency of our algorithms.

Keywords Reinforcement Learning · Reusability of ML models · Optimization algorithms

1 Introduction

The field of Reinforcement Learning (RL) has experienced significant advancements, making it a powerful tool for solving complex sequential decision-making problems across various domains [1, 15, 18, 21, 22, 29, 34, 40]. The database community has leveraged Q-learning and Deep-RL frameworks heavily in data exploration (e.g. [5, 28, 49]), course design [25], path planning [26], and in query optimization (e.g., [32]). This work focuses on how to reuse pretrained RL models to solve a new complex task optimally without involving any new training. Optimality is studied from the standpoint of *quality wrt finding policy with the highest cumulative reward*. While reusability of supervised models [20, 24, 52] have been studied in the data management community, to the best of our knowledge, ours is the first that initiates this study for RL.

RL training is costly not only because a single training session explores a large state-action space but also because fine-tuning an RL model is a trial-and-error process. It is common that reward signals need to be incorporated into already-deployed RL models (e.g., adding a penalty in click-through rate maximization [7]), requiring retraining from scratch. Although many efforts have been deployed to reduce training time [27], a medium-size RL model takes a few days to train [30, 38, 45]. This motivates RL reusability [31, 50, 51].

Example 1 Consider a user exploring a large set of online reviews to purchase a camera. The goal is to design an automated algorithm that guides the user to a particular target camera (and a representative review for that camera), while also showing other informative reviews in the process. Such processes are addressed as data exploration tasks [5, 28, 49] using RL. The trained model refines the original query "Cam-

Sepideh Nikookar: work done while at NJIT.

The order of authors is based on their respective contributions.

✉ Senjuti Basu Roy
senjutib@njit.edu

Sepideh Nikookar
sepideh.nikookar@emory.edu

Sohrab Namazi Nia
sn773@njit.edu

Sihem Amer-Yahia
sihem.amer-yahia@univ-grenoble-alpes.fr

Behrooz Omidvar-Tehrani
omidvart@amazon.com

1 New Jersey Institute of Technology, Newark, USA

2 Emory University, Atlanta, USA

3 CNRS, University Grenoble Alpes, Grenoble, France

4 Amazon AWS AI, Ohio, USA

era“ by choosing keywords, a.k.a. actions in RL, from the following list: $X = \{$Camera, Wide Lens, Shutter, Digital, ISO (Sensitivity to Light), Lightweight, Affordable$\}$.

RL is commonly modeled as a Deterministic Markov Decision Process (MDP) [5, 28, 49]. RL offers the benefit to perform decision-making tasks while adapting to the environment autonomously through continuous interaction. An MDP is defined over states, actions, and reward (formally defined in Sect. 2.1) and a deterministic MDP implies that executing the same action at a particular state consistently leads to the same subsequent state. This deterministic nature is commonly assumed in practice as it implies a high degree of predictability.

States and actions. Continuing with the same example, given 7 keywords, there are 2^7 non-empty queries that could be formed using them. Given any of these queries, there is one review that is most relevant to it and is returned. Each state therefore represents the most relevant review given a query. Thus there exists a total of 127 states, each of which has a review (one additional state has no keywords hence no reviews). The action space contains “adding” and “changing” keywords from the aforementioned list.

Pre-trained Models. For simplicity of exposition, let us assume that there exists two previously trained RL models in our environment for two different tasks. The first model was trained for task t_1, to navigate the review set as quickly as possible by selecting actions that enhance the similarity of the current query to a representative review of a target camera. The second trained model was for task t_2 to cover and satisfy as many features from the keyword lists as possible while converging to the representative review (the specific reward functions are defined in Example 2.

New task. We are given a new task t_c with the goal of exploring a sequence of reviews that maximize coverage of desirable features while minimizing steps to reach the representative review. In this case, the reward function of t_c is simply a sum of the reward functions of t_1 and t_2. Linear combination of reward components is quite common in the literature [5, 25, 26, 28, 49].

Our goal and approach. We intend to solve t_c optimally without any new training. We do not propose a new RL algorithm for that. Rather, we would like to use the pretrained models of t_1 and t_2. And more generally, we would like to leverage models that are pretrained with arbitrary reward functions.

Reusability of supervised models [20, 24, 52] has been studied in data management for entity resolution, or deep transfer learning. In the Machine Learning literature, meta-learning and transfer learning [25, 26, 31, 35, 36, 38, 39, 50, 51] study similar challenges for model reusability but without any optimality guarantees. Typical meta-RL algorithms assume the ability to sample from a pre-specified task distribution, and learn to efficiently solve new tasks drawn from this distribution. However, meta-RL is designed to only accelerate training time of a new task, but it cannot fully eliminate the need for any new training. Transfer learning is another widely used technique [25, 26, 35, 36, 38, 39] to reuse previously acquired knowledge. However, it does not enjoy theoretical guarantees [44].

We take a data management lens to understand what type of metadata of pretrained models need to be captured, how to store them effectively, and design efficient algorithms to reuse them for producing optimal policy for a new task.

Challenges. We are interested in designing an optimal policy (one that has the highest cumulative reward) with provable guarantees for a new task without any training. Our approach must be generic, i.e., it is applicable to both Temporal Difference Learning (TDL) such as Q-learning [46] and SARSA [38] as well as to Deep-RL[3] considering any number of arbitrary reward functions. Our first challenge is the design of an expressive and lossless model to represent and store the Q-values of pre-trained policies.

The next obstacle emanates from the update equations of TDL that heavily rely on the discount factor γ to capture the future reward of a $\langle state, action\rangle$ pair. As we shall demonstrate, the presence of a non-zero γ makes it impossible to exactly calculate the Q-values of the new task without training. At the same time, a non-zero γ also makes the future reward component of the Q-value update formulae recursive, meaning, the future reward of a $\langle state, action\rangle$ has recursive dependency on all future $\langle state, action\rangle$ pairs, until the end state is reached. This last challenge makes it hard to provide provable guarantees without training from scratch.

Formalism (**Sect.** 2). Our overarching contribution is to present a computational framework that captures fine-grained metadata of pre-trained RL models to obtain an equivalent policy for a new task. Equivalence is enforced by ensuring that the cumulative reward of the obtained solution is no smaller than the highest cumulative reward of the policy when it is trained from scratch. Our first contribution is to identify the generality and expressiveness of the proposed framework, i.e., under what assumptions the MDP (state, action, deterministic vs probabilistic transition, reward function structure, finite vs infinite horizon, discount factor) and for which RL approaches this is applicable. We design an expressive graph-based model augmented with lean data structures that capture fine-grained training information.

Algorithmic and theoretical contributions (**Sects.** 3 and 4). We first develop an exact linear time algorithm **ExZeroDiscount** for generating an optimal policy for a new task when the discount factor $\gamma = 0$. This algorithm produces a policy whose cumulative reward is no smaller than the cumulative reward of applying a policy trained from scratch. For $\gamma \neq 0$, the Q-value of a $\langle state, action\rangle$ has a component related to future reward, i.e., the next state that $\langle state, action\rangle$ leads to. This dependence is recursive: in a nutshell the future reward component of Q-value of a $\langle state, action\rangle$ depends on the Q-values of all future state-action until the goal state is reached. This reward depends on how many times state-action pairs are visited during training, a critical piece in Q-value calculation, which cannot be computed exactly for the new task without training. We show that it is possible to provide lower and upper bounds of future rewards. We also demonstrate that this limitation poses a serious computational bottleneck. We prove that in the worst case one may have to sift through an exponential number of paths to find the optimal policy for a new task. The recursion involved in the Q-update formulae requires us to design solutions that start from the goal state and backtrack, while producing bounds of Q-values for each $\langle state, action\rangle$ pair. These bounds are used to prune unnecessary computation and design the exact solution **ExNonZeroDiscount**. In Sect. 4.1, we develop principled algorithms that further tighten these bounds. We show that both **ExZeroDiscount** and **ExNonZeroDiscount** admit theoretical guarantees that they produce optimal policy with the highest cumulative reward without any new training. For applications that benefit from balancing quality and efficiency, we develop a parameterized algorithm **Greedy**-k in Sect. 4.2. Its behavior depends on the value of an input parameter k where higher values result in a slowdown but improved cumulative reward for the new task and vice versa. In Sect. 3.3, we analytically study the generality of the proposed framework in handling reward functions of the new tasks which are composed by scaling reward function of pre-trained tasks by arbitrary positive weights. We prove that our solutions remain valid and unaffected by the introduction of arbitrary yet constant weight factors. The intuition towards that is that the adjustment involves multiplying the corresponding part of the reward function by a weight factor only scales the reward component but does not alter the fundamental structure of the optimal policy. As a result, the computed optimal policy remains unchanged. The theoretical guarantee of the framework only applies to TDL approaches (that have Q-tables) and linear functions and works well in practice for the other cases. Adaptation to Deep-RL works well in practice, yet it does not provide theoretical guarantee, because these algorithms usually learn a neural network for estimating Q-values that are opaque, arbitrary, and could not be expressed in a closed form.

Experiments (Sect. 5). Our evaluation tackles four goals: (i) analyze the quality of the designed solutions, (ii) examine the generality of the designed solutions by scaling different components of the reward functions by arbitrary positive weights and study quality and speed up, (iii) analyze speed up, and (iv) investigate memory requirements and processing time of our graph-based data structure. Several essential metrics are considered, including recall, pruning percentage, speed up, memory usage. We vary RL-pertinent parameters (discount factor, state and action space size, number of reward functions), study them for two different use cases: query refinement and robot movement, and investigate Q-learning, SARSA, and Deep RL. We use an Amazon reviews dataset for data exploration and synthetic dataset for robot movement. The former relies on large language models (LLMs), namely gpt-3.5-turbo, in an innovative fashion to generate refined queries. We design appropriate baseline solutions, including transfer learning based ones.

We find that our solutions enable up to a $50x$ speedup compared to training from scratch. Even in cases where training is very fast, our speedup is $3x$. Experimental analyses on the generalized version of the reward function demonstrate speed up between 5 to 25 times compared to training from scratch. We also observe that our proposed exact solutions achieve 100% recall confirming our theoretical guarantees, and **Greedy**-k has the ability to strike a balance between quality and efficiency. Our experimental analyses demonstrate transfer learning techniques do not yield effective results in our settings. These empirical findings are consistent with the theoretical understanding of transfer learning. The primary motivation behind our work was to minimize unnecessary training, and, therefore, the transfer learning approaches implemented here do not involve any additional training phases. Consistently, these methods fail to perform effectively, which aligns with the general consensus that transfer learning typically requires extra training to achieve meaningful improvements. Consistent to our theoretical analysis, we finally demonstrate that our proposed solution always achieves 100% recall when the different components of the reward functions of the new task is scaled with arbitrary positive weights.

Section 6 presents the related work, we outline some open research problems and related discussions in Sect. 7, and we conclude in Sect. 8.

2 Preliminaries, data model, and problem

2.1 Preliminaries

Definition 1 (*Deterministic Markov Decision Process (MDP) with Multiple Reward Functions*) Our problem setting assumes a fully observable Deterministic Discrete Markov Decision Process (MDP) with a finite horizon and a collection of l reward functions, defined by a triplet $\{\mathcal{S}, \mathcal{A}, \mathcal{R}_l\}$: State space $\mathcal{S}$ is a set of states of the environment; Action space $\mathcal{A}$ is a set of actions from which the agent selects an action at each step; A set of l reward functions $\mathcal{R}_l$ where the i-th reward function (represents task t_i) R_i denotes the reward of an action a_i from state s_i to s'_i, $R_i = r(s_i, a_i, s'_i)$.

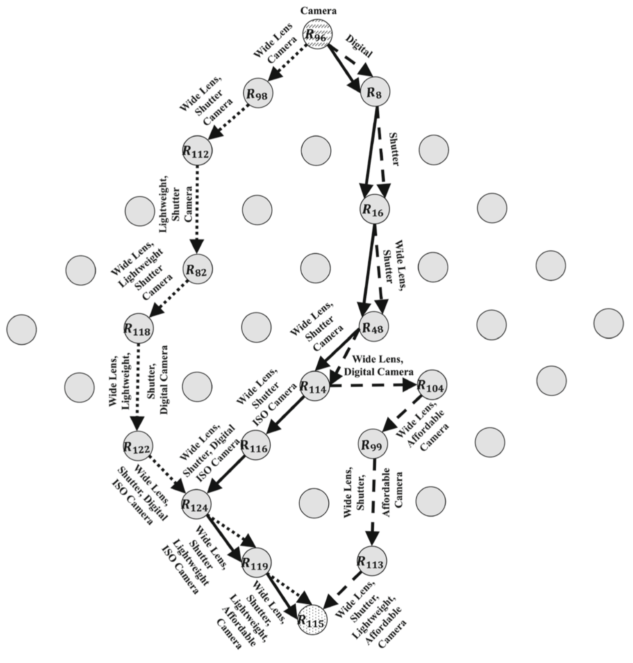

Camera
R_{96}
Wide Lens Camera
Digital
R_{98}
R_8
Wide Lens, Shutter Camera
Shutter
R_{112}
R_{16}
Lightweight, Shutter Camera
Wide Lens, Shutter
Wide Lens, Lightweight Shutter Camera
R_{82}
R_{118}
R_{48}
Wide Lens, Shutter Camera
Wide Lens, Digital Camera
Wide Lens, Lightweight, Shutter, Digital Camera
R_{114}
R_{104}
Wide Lens, Shutter ISO Camera
Wide Lens, Affordable Camera
R_{122}
R_{116}
R_{99}
Wide Lens, Shutter, Digital ISO Camera
Wide Lens, Shutter, Digital ISO Camera
Wide Lens, Shutter, Affordable Camera
R_{124}
Wide Lens, Shutter Lightweight ISO Camera
R_{119}
R_{113}
Wide Lens, Shutter, Lightweight, Affordable Camera
R_{115}
Wide Lens, Shutter, Lightweight, Affordable Camera

Fig. 1 A subset of environment for Example 1, that contains a start state s_{start} (R_{96}) a target state s_{target} (R_{115}). The dotted path shows an optimal policy for task t_1 and the dashed path shows the one for t_2. The optimal policy of the new task t_c is shown in solid

Using Example 1 (refer to Fig. 1), $\mathcal{S}$ has size 2^7, and possible actions are $\mathcal{A}$ = {*Change Word*, *Add Word*}.

Example 2 (**Reward Functions**) There are two tasks t_1 and t_2 in Example 1. t_1 explores reviews to find the one that is most similar to the target review (representative review for a target item) s_{target}.

$R_t(1)(s, a, s') = Cosine(s, s_{target}) - Cosine(s', s_{target})$, where the reward captures how a "chosen review" at s' is more similar to s_{target} than s. t_2 aims to cover as many keywords (features) as possible during review exploration to reach target review s_{target}.

$$\begin{aligned} v(s, a, s', x) &= 1, \text{ if } sim(s'^{x}, x) \geq 0.5 \\ &= 0, \text{ otherwise} \end{aligned}$$

$$R_t(2)(s, a, s') = \sum v(s, a, s', x), x \in X$$

If similarity between features in s' and each $x \in X$ is larger than 0.5 it is assumed that the review associated with s' contains x' and $v(s, a, s', x) = 1, 0$ otherwise. The reward function of the new task t_c is simply $R_t(1) + R_t(2)$.

2.1.1 Reinforcement Learning (RL) approaches

RL [38] is a popular approach for solving MDPs, where an agent acts on its environment and receives a reward for each action it takes, and optimizes the total cumulative reward. There are four main elements of an RL process: a policy, a reward signal, a value function, and, optionally, a model of the environment. Our technical report [2] contains further details on each.

A policy is a mapping from perceived states of the environment to actions to be taken when in those states. In some cases the policy may be a simple function or lookup table, whereas in others it may involve search process. A reward function defines the goal of an RL problem. On each time step, the environment sends the RL agent a reward value incurred from the action performed at that time step. The agent's sole objective is to maximize the total reward it receives over the long run. The reward signal is the primary basis for altering the policy. While the reward signal conveys immediate goodness, a value function delineates long-term desirability. The fourth element of RL is a model of the environment, enabling the system to draw conclusions about its behavior. For instance, when presented with a state and an action, this model has the capability to anticipate the subsequent state and the associated reward. In contrast, our focus are on simpler model-free approaches that rely explicitly on trial-and-error methods. Below, you will find a formal definition of 'policy' and 'optimal policy'.

Definition 2 (*Policy* π) A policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ of an RL agent maps the probability of taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$, that is, $\pi(s, a) = Pr(a_t = a | s_t = s)$.

Definition 3 (*Optimal Policy* π^*) A policy π^* is optimal iff its expected cumulative reward is greater than or equal to the expected cumulative reward of all other policies π. The optimal policy has an associated optimal state-value function, as well as an optimal action-value function, or optimal Q-function, as defined below:

$$Q^*(s, a) \leftarrow max_{\pi} Q_{\pi}(s, a)$$

In other words, Q^* gives the largest expected return achievable by any policy for each possible state-action pair.

We consider the following RL approaches.

Q-Learning [38]. It is a popular RL method that finds the optimal action selection policy in an MDP. It utilizes a Q-table to store the expected cumulative rewards for each $\langle state, action \rangle$ pair. By means of an iterative process that balances exploration and exploitation, Q-Learning continually updates Q-values by considering observed rewards. This enables it to make informed decisions and maximize long-term rewards.

$$Q(s, a, s') \leftarrow Q(s, a) + \alpha[R + \gamma \max_{a' \in A} Q(s', a') - Q(s, a)] \quad (1)$$

where $\alpha \in [0, 1]$ is the learning rate that determines how much the agent adjusts Q-values based on new experiences and $\gamma \in [0, 1]$ is the discount factor which holds significant importance in shaping the agent's learning process and decision-making strategy.

Discount factor γ. This factor represents the relative weight attributed to future rewards compared to immediate rewards. This parameter is incorporated into the Q-value update rule, where it determines the extent to which the agent values long-term consequences when selecting actions. A higher γ value encourages the agent to prioritize distant rewards, promoting strategic decision-making over extended time horizons. Conversely, a lower γ value leads to more myopic decision-making, prioritizing immediate gains over long-term benefits. Properly choosing γ is crucial for achieving optimal learning outcomes, as it influences the trade-off between exploration and exploitation, the convergence and stability of the algorithm, and the agent's risk sensitivity. Thus, γ plays a pivotal role in shaping the behavior, adaptability, and performance of Q-learning agents across various environments.

SARSA [38]. Another RL algorithm, SARSA operates similarly to Q-Learning but follows an on-policy approach. It considers the current policy when determining the next action and updates Q-values based on the subsequent state and action pair as follows:

$$Q(s, a, s') \leftarrow Q(s, a) + \alpha[R + \gamma Q(s', a') - Q(s, a)] \quad (2)$$

Deep RL [19, 38]. Deep RL combines RL with deep neural networks to handle complex tasks and large state or action spaces, enabling learning directly from raw sensory data without manual feature engineering. Various techniques exist to train policies to solve tasks with Deep RL algorithms (i.e., model-based vs model-free), each having their own benefits. To be consistent with approaches based on Q-tables, we study model-free Deep RL, where a policy π is learned using a neural network Q-function $Q(s, a)$ that estimates the future returns taking action a from state s.

2.2 Data model and problem definition

Tasks and Reward Functions. We consider a set of l tasks, each of which is defined over the same Deterministic Markov Decision Process, and each having its own reward function. Reward function of task t_i has a closed form described as $R_t(i)$. In Example 1, there are two tasks part of the same MDP and each with its own reward function given in Example 2 in Sect. 2.1.

Pre-Trained Models $\mathcal{M}_{(t_i)}$ of Task t_i. Each task is trained for ϵ episodes using TDL approaches such as Q-learning and SARSA resulting in a pre-trained models $\mathcal{M}_{(t_i)}$ and a task t_i. $\mathcal{M}_{(t_i)}$ represents the collection of policies obtained in training. As an example, the model of task t_1 in Example 1 determines actions to reach the target review while minimizing the number of explored reviews. The pre-trained model $\mathcal{M}_{(t_1)} = \{\pi^1_{(t_1)}, \pi^2_{(t_1)}, \ldots\}$ for task t_1 encompasses all possible strategies or rules that one could follow to explore reviews and reach the most similar review to the target review.

It is to be noted that TDL approaches such as Q-learning and SARSA train an agent in the environment, and the Q-table (lookup table) captures the maximum expected future rewards for each action at each state. The Q-function then represents the expected return for taking an action in a state following the model.

Q Table Q_{t_i} of Task t_i. The representation of a pre-trained model is in the form of its Q-table. The Q_{t_i} represents the expected future rewards associated with taking specific actions in different states within the environment specific to task t_i. By exploring Q_{t_i} through different traversal methods, such as selecting actions based on the highest Q-value or following a stochastic policy, multiple policies can be derived tailored to task t_i. These policies reflect varied strategies or decision-making approaches that enable the agent to effectively pursue its objectives within the context of task t_i.

Pre-Trained Models $\boldsymbol{\mathcal{M}}$. The set $\boldsymbol{\mathcal{M}} = \{\mathcal{M}_{(t_1)}, \mathcal{M}_{(t_2)}, \ldots, \mathcal{M}_{(t_l)}\}$ represents all pre-trained models for all l tasks. As an illustration, the pre-trained models $\boldsymbol{\mathcal{M}} = \{\mathcal{M}_{(t_1)}, \mathcal{M}_{(t_2)}\}$ in Example 1 represent the learned strategies and actions acquired through training. Specifically, $\mathcal{M}_{(t_1)}$ consists of policies tailored for finding the review that is most similar to the target review. $\mathcal{M}_{(t_2)}$ encompasses policies focused on selecting reviews that cover as many features as possible while exploring the review set.

Definition 4 (*Cumulative Reward $R(\pi_{(t_i)})$ of a policy $\pi_{(t_i)}$*) Given a policy $\pi_{(t_i)}$, its cumulative reward is the sum of rewards the agent receives from the actions it chooses to perform.

In Example 1, the cumulative reward associated with policy π_{t_1} is defined as the total sum of rewards obtained when looking for the reviews that are most similar to the target review. In essence, this cumulative reward quantifies how effectively the agent performs t_1 by evaluating the success of its actions in achieving the goal of reaching the target review.

Optimal Policy $\pi^*_{(t_i)}$ and Cumulative Reward of Task t_i. A policy $\pi^*_{(t_i)}$ for task t_i is optimal, if its expected return is greater than or equal to that of all other policies $\pi_{(t_i)} \in \mathcal{M}_{(t_i)}$ for all states. The optimal policy has an associated optimal state-value function and an optimal action-value function (i.e., Q-function). According to the Bellman equation [38], an optimal policy $\pi^*_{(t_i)}$ maximizes its cumulative reward. In Example 1, $\pi^*_{(t_1)}$ is the ideal strategy for finding reviews similar to the target review. It is selected based on its expected return, ensuring it outperforms all alternative policies $\pi_{(t_1)}$ for all possible states. The cumulative reward of t_1 represents the total expected reward that can be achieved when following the optimal policy $\pi^*_{(t_1)}$. This cumulative reward is essential for evaluating the effectiveness of the chosen strategy and quantifying the agent's performance in successfully completing the task.

Definition 5 (*New Reward Function/Task*) A new reward function $R_t(c)$ associated with a new task t_c, is added to the MDP, and is represented by a linear function of m existing functions $R_t(c) = f(R_t(i) \pm R_t(j) \ldots \pm R_t(m))$. From this point on, we will use new task and combined task interchangeably to refer to t_c.

Linearity Assumption. The reward for the new task must be a linear function of $m \leq l$ existing reward functions. This assumption is key to providing theoretical guarantees. In Example 1, the reward for task t_c is the sum of the rewards of t_1 and t_2, denoted as $R_t(c) = R_t(1) + R_t(2)$, where $R_t(1)$ and $R_t(2)$ represent the rewards for tasks t_1 and t_2, respectively.

Problem 1 (Optimal policy for t_c without training.) *Given t_c with reward function $R_t(c)$ and given all pre-trained models of l tasks, $\boldsymbol{\mathcal{M}} = \{\mathcal{M}_{(t_1)}, \ldots, \mathcal{M}_{(t_l)}\}$, the objective is to find an optimal policy for task t_c that maximizes its cumulative reward without the need for additional training for t_c. The reward function of t_c is $R_t(c) = f(R_t(i) \pm R_t(j) \ldots \pm R_t(m))$*

In Example 1, we have two different sets of pre-trained models, $\mathcal{M}_{(t_1)}$ for reaching the target review, and $\mathcal{M}_{(t_2)}$ for maximizing feature coverage. Our goal is to discover the optimal policy for the new task t_c with reward function $R_t(c) = R_t(1) + R_t(2)$ by leveraging what has already been learned through the pre-trained models $\mathcal{M}_{(t_1)}$ and $\mathcal{M}_{(t_2)}$. To solve our problem, we need to address the following cases:

- As shown and proved in Lemma 1, to compute a path for t_c with the highest cumulative reward without any training, where $R_t(c) = f(R_t(i) \pm R_t(j) \ldots \pm R_t(m))$, it is not sufficient to only consider the optimal policies of those tasks. This yields to the question of capturing $\boldsymbol{\mathcal{M}}$ (Sect. 3.1).
- Solve Problem 1 when $\gamma = 0$ (Sect. 3.2.1)
- Solve Problem 1 when $\gamma \neq 0$ (Sects. 3.2.2 and 4).

3 Proposed solution framework

Similarly to classical TDL algorithms, to solve a new task t_c, we need to quantify the maximum expected future rewards for each (s, a). The challenge however is that since t_c has not been trained yet, the Q-table of t_c is simply not available. We therefore need to reproduce the Q-table of t_c using $\boldsymbol{\mathcal{M}}$.

3.1 Capturing $\boldsymbol{\mathcal{M}}$

We first demonstrate why storing $\boldsymbol{\mathcal{M}}$ is not trivial. We then explain how to store $\boldsymbol{\mathcal{M}}$, such that the Q-table of t_c could be reproduced in a lossless manner.

Lemma 1 *Given t_c and $R_t(c) = R_t(i) + R_t(j)$, the optimal policy $\pi^*_{(t_c)}$ for t_c cannot be obtained only from the paths of $\pi^*_{(t_i)}$ and $\pi^*_{(t_j)}$.*

Proof (sketch.) The lemma states that even when the reward of a new task t_c can be expressed by simply adding up the rewards of two pre-trained tasks t_i and t_j, the optimal policy of t_c cannot always be constructed solely from the paths produced by the optimal policies of t_i and t_j. The proof relies on a counterexample. In Fig. 1 the cumulative reward of $\pi^*_{(t_1)} = 8$ (resp. $\pi^*_{(t_2)} = 20$ and $\pi^*_{(t_c)} = 26$) is denoted by the dotted (resp., dashed and solid) path. It is easy to notice that the the solid path contains a state that is neither present in the dotted nor in the dashed.

We refer to Eqs. 1, 2 and note that the Q-value of every $\langle state, action \rangle$ contains immediate reward and marginal increase in the Q-values between (s', a') and (s, a) (where s' is the next state). The second component has a recursive dependence on the number of times s' is visited during training. Hence, either the Q-table of t_c is obtained by training t_c from scratch or from the Q-tables of other pre-trained tasks. The key observation is that there does not exist a formal relationship between the number of times a state is visited in both cases. This potentially incurs different marginal increases in Q-values between (s', a') and (s, a). In fact, it is easy to construct a counter-example to that end (refer to Table 2 for details).

To estimate the Q-table of t_c in a lossless manner, one has to store the updated Q-value of every $\langle state, action \rangle$ pair of every task t_i. We represent this as a triplet $\langle t_i, a, q_{s,a,s'} \rangle$, where the first element denotes the task id, the second element represents the action, and the third element is the Q-value associated with the $\langle state, action \rangle$ pair (s, a), consistently leading to s' in deterministic MDPs.

Graph Creation to Store $\boldsymbol{\mathcal{M}}$. $\boldsymbol{\mathcal{M}}$ is stored as a directed attributed graph $G(\mathcal{S}, \mathcal{E})$, where edges are labeled. Each state in $\mathcal{S}$ is a node in $\mathcal{G}$. For each (s, a, s'), an edge between (s, s') is created with $\langle t_i, a, q_{s,a,s'} \rangle$. Clearly, multiple edges may exist between (s, s'), because the same (s, a, s') could be obtained with different t_i with different Q-values, and multiple actions are possible. This generates the edge set $\mathcal{E}$ of $G(\mathcal{S}, \mathcal{E})$. Algorithm 1 contains the pseudocode.

Space and Running time. $G(\mathcal{S}, \mathcal{E})$ incurs negligible processing time as it is created during training itself. The space required to store $G(\mathcal{S}, \mathcal{E})$ is $(|\mathcal{E}|+|\mathcal{S}|)$. In Examples 1 and 2, this process produces the graph illustrated in Fig. 2.

Table 1 Table summarizing our key notations

Symbol	Description
ϵ	# Episodes
l	# Pre-trained tasks
$R_t(.)$	Reward function of task t
$\pi^*_{(t)}$	Optimal policy of t
$\mathcal{M}_{(t)}$	All policies of t
Π	All policies of all l tasks
t_c	New task
$R_t(c) = f(R_t(i) \pm ... \pm R_t(m))$	t_c's reward is a linear function of $m \leq l$ pre-trained tasks

Algorithm 1 Creating $G(\mathcal{S}, \mathcal{E})$ to Store $\mathbf{\Pi}$

```
Require: $\mathcal{S}$, Action Set $\mathcal{A}$, $\mathcal{R}_l$, $\epsilon$
  Initialize $G(\mathcal{S}, \mathcal{E}) = \{\mathcal{S}, \varnothing\}$
  for each task $t$ do
    for $i \leftarrow 1$ to $\epsilon$ do
      while $s \neq s_{goal}$ do
        Observe the current state $s$
        $a \leftarrow argmax_{\forall a \in A} R(s, a, s)$, $r \leftarrow R(s, a, s)$
        update $Q(s, a)$ based on Equations 1, 2
        $e \leftarrow < t, a, q_{s,a,s'} >$
        $E \leftarrow \{E \bigcup e\}$
        $s \leftarrow s'$
```

Algorithm 2 Algorithm **ExZeroDiscount**

```
Require: $t_c$, $R_t(c)$, $G(\mathcal{S}, \mathcal{E})$, $s_{start}$ and $s_{goal}$ of $t_c$
  $s \leftarrow s_{start}$; $\pi^*_{(t_c)} = \varnothing$
  for each $(s, a, s')$ do
    $Q_{t_c}(s, a, s') \leftarrow \sum_{R_t(i) \in R_t(c)} Q_{t_i}(s, a, s')$
  $s \leftarrow s_{start}$
  while $s \neq s_{goal}$ do
    $\{s', a\} \leftarrow argmax_{\forall s' \in \{\mathcal{S}-s\}, \forall a \in \mathcal{A}} Q_{t_c}(s, a, s')$
    $\pi^*_{(t_c)} \leftarrow \{\pi^*_{(t_c)} \rightarrow (s, a)\}$
    $s \leftarrow s'$
  return $\pi^*_{(t_c)}$
```

3.2 Algorithms for solving t_c

Given a new task t_c whose reward is a linear combination of m different reward functions for which policies are pre-trained in $\mathcal{M}$ and stored in $G(\mathcal{S}, \mathcal{E})$, we are now ready to describe our algorithms for solving t_c. The primary challenge is to take the original scaffolding of TDL algorithms in an effort to construct the Q-table of t_c from $G(\mathcal{S}, \mathcal{E})$ in a lossless manner.

To that end, we study two distinct cases: Algorithm **ExZeroDiscount** is designed when the discount factor $\gamma = 0$, whereas, Algorithm **ExNonZeroDiscount** handles the more complex scenario of non-zero discount factor γ. Our algorithms exhibit different mathematical properties, requiring significantly different logic with a zero discount factor ($\gamma = 0$), the update rule of Q-values are simple, they only depend on the immediate reward of each (s, a). We note that a zero discount factor scenario is not superficial: when the end goal of the agent is unknown at the beginning (e.g., discovering an unknown destination in a mission-critical application where the agent only knows it reached the end goal when it sees it), the Q function must rely on the immediate reward only, making $\gamma = 0$.

Both algorithms are designed using two primary steps **(A)** and **(B)**: **(A) Q-table reconstruction.** The goal is to reconstruct the Q-table of t_c from $G(\mathcal{S}, \mathcal{E})$. **(B) Optimal policy computation for** t_c. Given a start and end state for t_c, do forward look-up and find the optimal policy with the maximum cumulative reward. We shall demonstrate in the next section that while achieving both of these steps is computationally easy in **ExZeroDiscount**, it becomes highly expensive for **ExNonZeroDiscount**.

3.2.1 Algorithm ExZeroDiscount.

We first provide the insights behind the design of **ExZeroDiscount**. The main intuition is the following - **(A) Q-table reconstruction.** With a zero discount factor, the Q update rules are purely greedy and depend only on the immediate reward of a state-action. **(B) Compute optimal policy for** t_c. Given the re-constructed Q-table, it is computationally easy to exactly compute the aggregated cumulative Q-values of every (s, a) of t_c using the formula expressed in Eq. 6, when $\gamma = 0$. Therefore, the optimal policy of t_c is guaranteed to follow the path that has the highest aggregated Q-value for every state-action pair starting at s_{start} of t_c and ending at s_{goal} of t_c.

Consequently, given $G(\mathcal{S}, \mathcal{E})$, **ExZeroDiscount** computes the aggregated Q-value of each (s, a) considering the Q-values of the m reward functions that t_c's reward is defined with. After that, the algorithm identifies the path with the highest cumulative Q-value greedily. This path then emerges as the optimal policy for t_c. The first loop in Algorithm 2 does Q-table reconstruction and the while loop in the algorithm produces the optimal policy with a single greedy pass.

Table 2 Q-tables of t_1, t_2 and t_c of Examples 1 and 2 after 4 episodes, $\alpha = 0.1$, $\gamma = 0.9$

State/action	t_1		t_2		t_c	
	Change Word	Add Word	Change Word	Add Word	Change Word	Add Word
R_{96}	0.199	0.199	0.788	–	1.18	–
R_8	0.199	–	0.823	–	1.185	–
R_{16}	–	0.199	–	1.15	–	1.23
R_{98}	–	0.199	–	–	–	–
R_{48}	0.1	0.1	–	0.958	–	1.54
R_{97}	–	0.1	–	–	–	–
R_{112}	0.2	–	–	–	–	–
R_{82}	0.109	0.1	–	–	–	–
R_{114}	–	0.199	0.597	0.588	0.4	1.16
R_{104}	0.1	–	0.588	–	0.4	–
R_{99}	–	0.1	–	0.398	–	0.3
R_{118}	–	0.1	–	–	–	–
R_{116}	–	0.199	–	0.398	–	0.891
R_{113}	–	0.1	–	0.38	–	0.3
R_{122}	0.188	–	–	–	–	–
R_{124}	0.296	–	0.398	–	0.889	–
R_{119}	0.271	–	0.38	–	0.813	–
R_{115}	–	–	–	–	–	–

The first (resp., second and last) two columns represent Q-table of t_1 (resp., t_2 and t_c). It is easy to notice that the Q-values of (s, a) of t_c are not easily relatable to those of t_1 and t_2

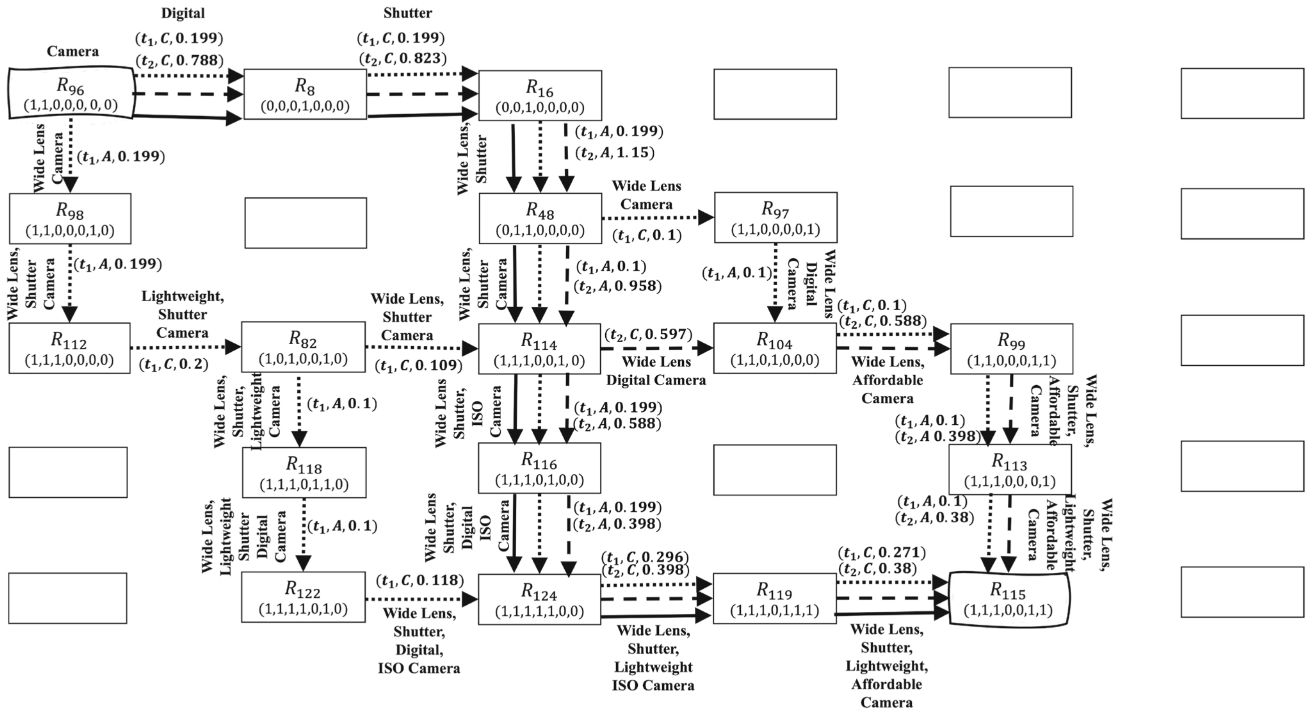


Fig. 2 The sub-graph $G'(\mathcal{S}', \mathcal{E}') \subseteq G(\mathcal{S}, \mathcal{E})$ representing Example 1. The graph stores labeled information on each directed edge of the form $\langle t_i, a, q_{s,a,s} \rangle$ and the updated query. The dotted and dashed paths show stored information obtained during training for t_1 and t_2, respectively. The optimal policy of t_c is shown in solid. The Q-values are obtained using a learning rate $\alpha = 0.1$, a discount factor $\gamma = 0.9$

On Examples 1 and 2, Algorithm **ExZeroDiscount** produces the solid path in Fig. 1 as the optimal policy with reward 26.

Running Time of Algorithm ExZeroDiscount. Algorithm **ExZeroDiscount** requires $\mathcal{O}(m \times |\mathcal{E}|)$ times to process the for loop and $\mathcal{O}(|\mathcal{S}|)$ times to compute the while loop. The running time is thus dominated by $\mathcal{O}(m \times |\mathcal{E}|)$.

Theoretical Claims. When $\gamma = 0$, the update equation (Eqs. 1 and 2) for Q-values for all $\langle state, action \rangle$ pairs simplifies to:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha R \tag{3}$$

Training has to run for ϵ episodes for each task. When, $\gamma = 0$, the Q-value of a state-action (s, a) of a task that has been visited $\eta \le \epsilon$ times, could be captured in a closed form, as Lemma 2 describes.

Lemma 2

$$Q^{\eta}(s,a) \leftarrow (-1)^{\eta-1}R[(\alpha-1)^{\eta} + (-1)^{\eta-1}]; \quad \forall (s,a) \in \mathcal{S} \times \mathcal{A} \tag{4}$$

Proof Let us consider Q-Learning as our chosen approach with the following Q-function (SARSA could be proved analogously):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma\ max_{a'} Q(s',a') - Q(s,a)] \tag{5}$$

When $\gamma = 0$, the Q-function simplifies to $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha R$. We proceed with a proof by induction. When (s, a) is visited for the first time, the Q-value is updated as $Q^1(s,a) \leftarrow \alpha R$, since all Q-values are initially set to 0. Now, with $\eta = 1$ in Equation (4):

$$\begin{aligned} Q^1(s,a) &= (-1)^{1-1}R[(\alpha-1)^1 + (-1)^{1-1}] \\ &= R[\alpha - 1 + 1] = \alpha R \end{aligned}$$

Assuming (s, a) has been visited $(\eta - 1)$ times, its Q-value is:

$$Q^{\eta-1}(s,a) \leftarrow (-1)^{\eta-2}R[(\alpha-1)^{\eta-1} + (-1)^{\eta-2}]$$

Now, we can calculate the Q-value when (s, a) is visited η times (the same steps apply to SARSA):

$$\begin{aligned} Q^{\eta}(s,a) &= (1-\alpha)Q^{\eta-1}(s,a) + \alpha R \\ &= (1-\alpha)[(-1)^{\eta-2}R[(\alpha-1)^{\eta-1} + (-1)^{\eta-2}]] \\ &\quad + \alpha R \\ &= (-1)(\alpha-1)[(-1)^{\eta-2}R(\alpha-1)^{\eta-1} \\ &\quad + (-1)^{2\eta-4}R] + \alpha R \\ &= (-1)^{\eta-1}R(\alpha-1)^{\eta} + (1-\alpha+\alpha)R \\ &= (-1)^{\eta-1}R[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \end{aligned}$$

Theorem 1 *If the discount factor γ is set to zero in both Q-Learning and SARSA algorithms, the Q-value of the new task t_c for all (state, action) (s, a) pairs will be obtained as follows:*

$$Q_{t_c}(s,a) = Q_{t_i}(s,a) \pm Q_{t_j}(s,a) \pm \cdots \pm Q_{t_m}(s,a); \ \forall (s,a) \tag{6}$$

Proof Using Lemma 2, the Q-value for each individual policy after η visits can be expressed as follows:

$$\begin{aligned} Q^{\eta}_{t_i}(s,a) &= (-1)^{\eta-1}R_t(i)[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \\ Q^{\eta}_{t_j}(s,a) &= (-1)^{\eta-1}R_t(j)[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \\ &\vdots \\ Q^{\eta}_{t_m}(s,a) &= (-1)^{\eta-1}R_t(m)[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \end{aligned}$$

For the combined task, the Q-value after seeing (s, a) η times is:

$$\begin{aligned} Q^{\eta}_{t_c}(s,a) &= (-1)^{\eta-1}(R_t(i) \pm R_t(j) \pm \ldots \\ &\quad \pm R_t(m))[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \end{aligned}$$

Now, combining Q-values of all m policies gives the following:

$$\begin{aligned} Q^{\eta}_{t_i}(s,a) &\pm Q^{\eta}_{t_j}(s,a) \pm \cdots \pm Q^{\eta}_{t_m}(s,a) \\ &= (-1)^{\eta-1}R_t(i)[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \\ &\pm (-1)^{\eta-1}R_t(j)[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \\ &\vdots \\ &\pm (-1)^{\eta-1}R_t(m)[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \\ &= (-1)^{\eta-1}(R_t(i) \pm R_t(j) \pm \cdots \\ &\pm R_t(m))[(\alpha-1)^{\eta} + (-1)^{\eta-1}] \\ &= Q^{\eta}_{t_c}(s,a) \end{aligned}$$

Theorem 2 *Algorithm **ExZeroDiscount** returns an optimal policy $\pi^*_{(t_c)}$ with the highest cumulative reward for t_c.*

Proof (sketch.) We omit the details due to lack of space. We note that when $\gamma = 0$, the Q-values of every (s, a) depends on the immediate reward which **ExZeroDiscount** could calculate exactly. Then the forward pass on the Q-table will produce the policy with the highest cumulative reward.

3.2.2 Algorithm ExNonZeroDiscount

When $\gamma \neq 0$, the problem becomes more intricate due to the fact that Q-values of any (s, a) of t_c is no longer solely reliant on immediate rewards. They now depend on the Q-value of the next state s', which in turn depends on the how many times s' has been visited during training and how much of Q-values it calculated. In fact, this dependency propagates recursively and there is no easy way to calculate any (s, a) of t_c. **(A) Q-table reconstruction.** It is impossible to derive the exact Q-value of (s, a) for t_c. What remains still possible is to provide two boundary values for each Q-value of (s, a) (the lower- and the upper-bounds), within which the actual Q-value of (s, a) resides. Subroutine **`BackTrackDAG`** (Line 1 of Algorithm 3) is designed to enable that. **(B) Compute optimal policy for t_c.** Subroutine **`ForwardLookUp`** first computes all candidate policies that could potentially be an optimal policy for t_c. Unfortunately, this step is also computationally intensive for **ExNonZeroDiscount**, since in the worst case there could an exponential number of paths to evaluate to determine the optimal policy of t_c with the highest cumulative reward. **ExNonZeroDiscount** (Line 2 of Algorithm 3) then computes cumulative rewards for each of those policies and selects the one with the highest cumulative reward.

Algorithm 3 Algorithm **ExNonZeroDiscount**
```
Require: $t_c$, $R_t(c)$, $G(\mathcal{S}, \mathcal{E})$, $s_{start}$ and $s_{goal}$ of $t_c$
  $\{LB, UB\} \leftarrow \texttt{BackTrackDAG}(G, t_c, R_t(c))$
  $\pi^*_{(t_c)} \leftarrow argmax_{\forall \pi \in Cand}\texttt{ForwardLookUp}(s_{start}, s_{goal}, LB, UB)$
```

Algorithm 4 Subroutine `BackTrackDAG`
```
Require: $t_c$, $R_t(c)$, $G(\mathcal{S}, \mathcal{E})$
  $s' \leftarrow$ end; $s \leftarrow backtrack(s', a)$
  while $s \neq s_{start}$ do
    for each $(s, a)$ do        ▷ Iterate over each $(s, a)$ pair leading to $s'$
      $LB_Q(s, a) \leftarrow \texttt{ComputeLB}(G, t_c, R_t(c))$
      $UB_Q(s, a) \leftarrow \texttt{ComputeUB}(G, t_c, R_t(c))$
      $LB \leftarrow LB \bigcup LB_Q(s, a)$
      $UB \leftarrow LB \bigcup UB_Q(s, a)$
    $s' \leftarrow s$
  return $LB, UB$
```

Subroutine `BackTrackDAG`. Algorithm 4 admits t_c, $R_t(c)$, $G(\mathcal{S}, \mathcal{E})$ and produces the Q-values of every (s, a) for t_c. That, however, is not immediately possible. We refer to the Q-value update (Eqs. 1 and 2) of SARSA and Q-learning [38], and note that, except for the end state, the Q-value of every $\langle state, action\rangle$ has a recursive dependency on the Q-values of future ones. *This indeed is a key observation as it allows us to reason on the values of the Q-table of t_c from $G(\mathcal{S}, \mathcal{E})$.*

Algorithm 5 Subroutine **`ForwardLookUp`**
```
Require: $t_c$, $LB$, $UB$, $G(\mathcal{S}, \mathcal{E})$, $s_{start}$ and $s_{goal}$ of $t_c$
  $s \leftarrow s_{start}$; $Cand \leftarrow \varnothing$; Policy $\pi \leftarrow \varnothing$
  for each (s,a) do
    while $s \neq s_{goal}$ do
      if $UB_Q(s, a) < LB_Q(s, a')$, $a' \in \{\mathcal{A} - a\}$ then
        prune edge $(s, a)$
      else
        $\pi \leftarrow \{\pi \rightarrow (s, a)\}$
      $s \leftarrow s'$
    $Cand \leftarrow \{Cand \bigcup \pi\}$
    Policy $\pi \leftarrow \varnothing$
    $s \leftarrow s_{start}$
  return $Cand$
```

Thus, **`BackTrackDAG`** involves a backtracking mechanism that starts at the node of G that does not have any out degree (meaning this node is the end state during training), calculates its Q-value, and then uses that to estimate the lower- and upper-bounds of Q-values of every (s, a) that immediately precede the goal state (that means, these latter states are reached before the goal state during training). The backtracking continues, until the start state is reached. However, to estimate the bounds of $Q(s, a)$ accurately, it is important that (s, a) and any successor of (s, a) do not have any cyclic dependency. That is, the subgraph of $G(\mathcal{S}, \mathcal{E})$ that is used to solve t_c is a DAG and does not contain cycles.

We refer to Fig. 2 and note that for the running example, this process starts at the end node R_{115}, and calculates the lower and upper bounds of Q-value of $\langle state, action\rangle$ $[R_{113}, \textit{Add Word}]$, then it calculates Q-value of $\langle state, action\rangle$ $[R_{99}, \textit{Add Word}]$, and so on. If an (s, a) has a successor state s' with multiple out-degrees (such as $[R_{48}, \textit{Add Word}]$ leads to R_{114} which has two successor nodes R_{104} and R_{116}), the Q-values of all the successors need to be computed before.

Lemma 3 ***BackTrackDAG*** *produces correct lower and upper bound of Q-values for every (s, a) only when the subgraph of $G(\mathcal{S}, \mathcal{E})$ that is involved in computing Q-values of t_c is a DAG and does not contain any cycle.*

Proof Here we discuss why the subgraph of $G(\mathcal{S}, \mathcal{E})$ that is involved in computing Q-values of t_c must be a DAG (directed acyclic graph) to produce correct lower and upper bound of Q-values. Refer to Fig. 3 below and notice that because of an extra action *Drop Word* allowed in the environment, now there exist a cycle between the states R_{48} and R_{114}.

The subgraph of $G(\mathcal{S}, \mathcal{E})$ that is involved in computing Q-values of t_c must be a DAG (Directed Acyclic Graph) to produce correct lower and upper bound of Q-values. Without that being satisfied, the framework still extends but the solution does not withstand any optimality guarantee anymore. Refer to Fig. 3 below and notice that because of an extra action *Drop Word* allowed in the environment,

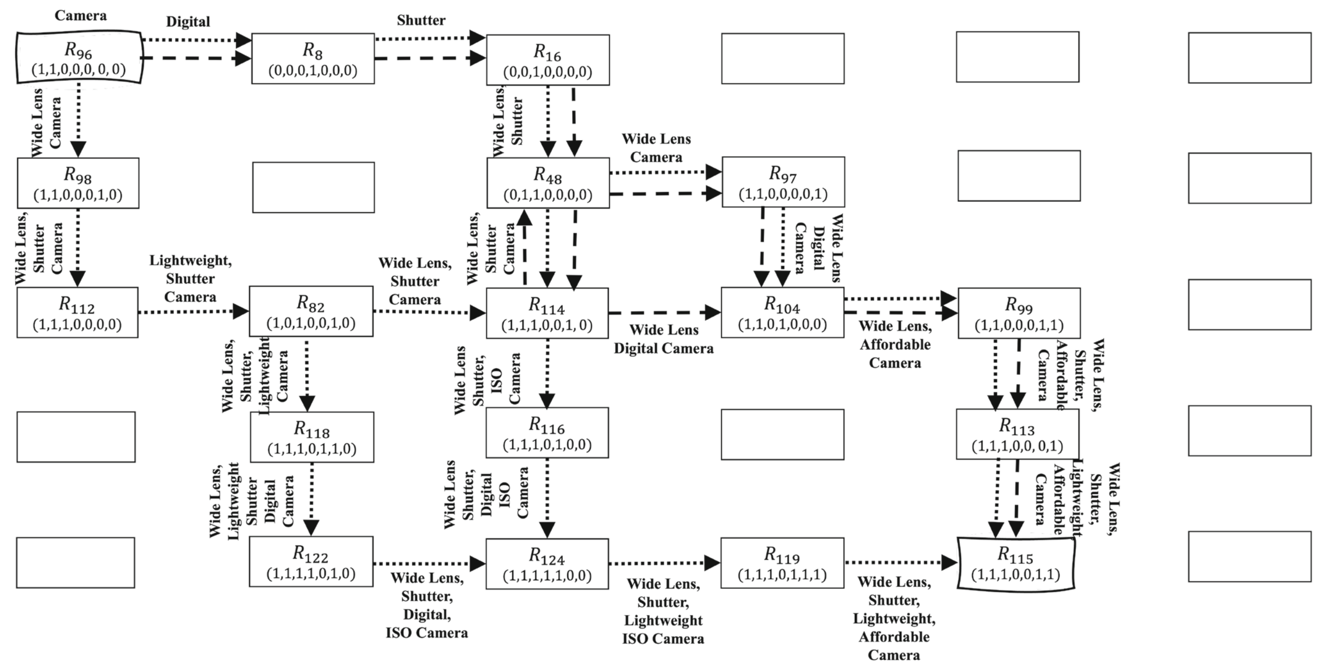

Camera
R_{96} (1,1,0,0,0, 0, 0)
Digital
R_8 (0,0,0,1,0,0,0)
Shutter
R_{16} (0,0,1,0,0,0,0)
Wide Lens Camera
R_{98} (1,1,0,0,0,1,0)
Wide Lens, Shutter
R_{48} (0,1,1,0,0,0,0)
Wide Lens Camera
R_{97} (1,1,0,0,0,0,1)
Wide Lens, Shutter Camera
Wide Lens, Shutter Camera
Wide Lens Digital Camera
Lightweight, Shutter Camera
Wide Lens, Shutter Camera
R_{112} (1,1,1,0,0,0,0)
R_{82} (1,0,1,0,0,1,0)
R_{114} (1,1,1,0,0,1, 0)
Wide Lens Digital Camera
R_{104} (1,1,0,1,0,0,0)
Wide Lens, Affordable Camera
R_{99} (1,1,0,0,0,1,1)
Wide Lens, Shutter, Lightweight Camera
Wide Lens, Shutter, ISO Camera
Wide Lens, Shutter, Affordable Camera
R_{118} (1,1,1,0,1,1,0)
R_{116} (1,1,1,0,1,0,0)
R_{113} (1,1,1,0,0, 0,1)
Wide Lens, Lightweight Shutter Digital Camera
Wide Lens Shutter, Digital ISO Camera
Wide Lens, Shutter, Lightweight Affordable Camera
R_{122} (1,1,1,1,0,1,0)
R_{124} (1,1,1,1,1,0,0)
R_{119} (1,1,1,0,1,1,1)
R_{115} (1,1,1,0,0,1,1)
Wide Lens, Shutter, Digital, ISO Camera
Wide Lens, Shutter, Lightweight ISO Camera
Wide Lens, Shutter, Lightweight, Affordable Camera

Fig. 3 The graph $G(\mathcal{S}, \mathcal{E})$ representing Example 1 in the paper with one additional change. Just for the purpose of this example, let us assume that the agent is allowed to take 3 different actions: $\mathcal{A} = \{Change\ Word, Add\ Word, Drop\ Word\}$, and two tasks with their own reward functions are defined and described in Example 2. The graph stores labeled information on each directed edge of the form $\langle t_i, a, q_{s,a,s} \rangle$. The dotted and dashed paths show stored information obtained during training for t_1 and t_2, respectively, which are removed from this figure for the ease of exposition

now there exist a cycle between the states R_{48} and R_{114}. Refer to Fig. 3 again and note based on the procedure **BackTrackDAG**, the algorithm starts at the end node R_{115}, and use that to calculate the lower and upper bound of Q-value of $\langle state, action \rangle$ $[R_{113}, Add\ Word]$, use that to calculate Q-value of $\langle state, action \rangle$ $[R_{99}, Add\ Word]$, and so on. If an (s, a) has a successor state s' with multiple out-degrees (such as $[R_{48}, Add\ Word]$ leads to R_{114} which has two successor nodes R_{104} and R_{116}, in order to calculate the Q-value of that (s, a) the Q-values of all the successors need to be computed before. Following this process above, $[R_{16}, Add\ Word]$ leads to R_{48} which has two successor nodes R_{97} and R_{114} whose Q-values are to be used to calculate the Q-value of R_{48}. Now R_{114} depends on R_{48}, R_{104} and R_{116} - however, R_{48} (which is $[R_{16}, Add\ Word]$) depends on R_{114} and R_{97}, leading to a cyclic dependency. It is to be noted that until the order of resolution is determined between R_{48} and R_{114}, **BackTrackDAG** can not proceed accurately. It is also easy to notice that the subgraph of $G(\mathcal{S}, \mathcal{E})$ that is involved in computing Q-values of t_c must be a DAG in order to have an unique order of resolution for every $\langle state, action \rangle$ in the environment.

Lower and Upper bounds of Q-values. There are non-trivial challenges involved in **BackTrackDAG**. As mentioned before, the exact Q-value of (s, a) cannot be calculated for t_c. What is possible is to calculate a lower and upper bound of Q-values for each (s, a) and use that to produce t_c efficiently. There are trivial ways of computing these bounds for every (s, a). For example, $LB_Q(s, a)$ could be calculated considering the worst case, i.e., the next state s' is visited only once - whereas, $UB_Q(s, a)$ could be estimated by considering the best case, that is, the next state s' has been visited all ϵ times. In Sect. 4, we describe further opportunities on how to estimate these bounds in a tighter manner. Subroutine **BackTrackDAG** returns these LB, UB values for each (s, a) for t_c.

Subroutine ForwardLookUp. This subroutine (refer to Algorithm 5) takes the lower and upper bounds (LB, UB) of Q-values returned by **BackTrackDAG** and given s_{start} and s_{goal} of t_c, it returns all possible candidate policies $Cand$ based on Q-values (refer to Algorithm 5). It starts at s_{start} and makes a forward traversal to find every path that starts at s_{start} and ends at s_{end} on $G(\mathcal{S}, \mathcal{E})$. Each path corresponds to a policy π (refer to Algorithm 5). To make the process further efficient, it prunes out some unnecessary (s, a). If the $UB_Q(s, a)$ of a (state, action) is smaller than the lower bound of Q-values of all other actions a' that could be made at state s, it deems (state, action) to be unnecessary and prunes that (s, a) (basically it decides there are alternative better ways to transit from state s than taking action a). The algorithm

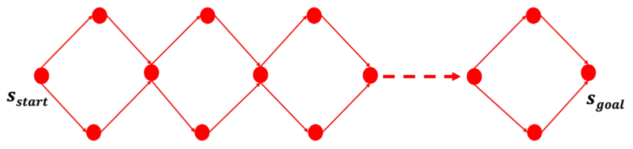


Fig. 4 # possible policies **ForwardLookUp** returns could be exponential to the number of diamonds $\mathcal{O}(2^{|\mathcal{S}|})$ at the worst case even when there are only two actions

ends when all possible policies are computed between s_{start} and s_{end}.

Lemma 4 ***ForwardLookUp*** *can return an exponential number of policies in the worst case, even when only two actions are involved.*

Proof (sketch). Refer to Fig. 4 where policies in $Cand$ may form a diamond graph and the number of policies could be $\mathcal{O}(2^{|\mathcal{S}|})$, precisely $2^{\frac{|\mathcal{S}|-1}{3}}$.

Theorem 3 *Algorithm* ***ExNonZeroDiscount*** *returns an optimal policy $\pi^*_{(t_c)}$ with the highest cumulative reward for t_c.*

Proof (sketch). The proof could be done by construction. It can be shown that **BackTrackDAG** indeed returns the bounds of Q-values of every (s, a) of t_c, where the Q-value of (s, a) of t_c is within those bounds (if t_c is trained from scratch). It can also be shown that **ForwardLookUp** returns all possible policies in the Q-table of t_c when training from scratch. It is hence easy to infer that one of those policies that **ForwardLookUp** returns will have the highest cumulative reward and hence be the optimal.

Running time of ExNonZeroDiscount. As proved in Lemma 4, **ForwardLookUp** may return an exponential number of policies in the worst case, even when there are only two actions. Therefore, when there are $|\mathcal{A}|$ possible actions and $|\mathcal{S}|$ possible states, in the worst case it could take $\mathcal{O}(|\mathcal{A}|^{|\mathcal{S}|})$ time to run.

3.3 More general case of combining linear reward functions

In this subsection, we consider a more general case of the reward function defined earlier (Definition 5).

Definition 6 (*New Reward function: Weighted linear function of existing functions*) A new reward function $R_t(c)$ associated with a new task t_c, is added to the MDP, and is represented by a weighted linear function of m existing functions $R_t(c) = f(w_i \times R_t(i) \pm w_j \times R_t(j) \cdots \pm w_m \times R_t(m))$, where $\{w_1, w_2, w_3, \ldots, w_m\}$ are positive constant weights.

Please note that Sect. 3 studies a special case of the problem where $w_1 = w_2 = w_3, = \ldots = w_m = 1$.

In the general case, the solutions presented in Sect. 3 for both $\gamma = 0$ and $\gamma \neq 0$ remain valid and unaffected by the introduction of arbitrary yet constant weight factors. Specifically, the adjustment involves multiplying the corresponding part of the reward function, $R_t(i)$, by this weight factor w_i. This modification scales the reward component but does not alter the fundamental structure of the optimal policy. As a result, the computed optimal policy remains unchanged, irrespective of the value of γ, as long as the reward terms are consistently weighted.

To elaborate, the inclusion of the weight factor w_i influences the magnitude of the rewards but does not impact the underlying decision-making process that determines the optimal policy. In reinforcement learning and decision theory, policies are derived based on the relative comparison of different actions and states, typically maximizing the cumulative expected reward. Since the weight factor merely scales the reward terms by a constant factor, the ranking of actions and the resulting optimal policy remain the same.

Thus, whether $\gamma = 0$ (which corresponds to a myopic policy focusing on immediate rewards) or $\gamma \neq 0$ (which reflects a long-term perspective by considering future rewards), the optimal policy remains unchanged when the reward function is weighted appropriately. This robustness demonstrates that the optimal policy is primarily determined by the structure of the reward function and the decision-making model, rather than the absolute scale of the rewards themselves.

Theorem 4 *Algorithms* ***ExZeroDiscount*** *and* ***ExNonZeroDiscount*** *return an optimal policy $\pi^*_{(t_c)}$ with the highest cumulative reward for t_c, even when $R_t(c) = f(w_i \times R_t(i) \pm w_j \times R_t(j) \cdots \pm w_m \times R_t(m))$, where $\{w_1, w_2, w_3, \ldots, w_m\}$ are positive constant weights.*

Proof (sketch): The graph structure $G(\mathcal{S}, \mathcal{E})$ (described in Sect. 3.1) still stores the updated Q-value of every $\langle state, action \rangle$ pair of every task t_i as a triplet $\langle t_i, a, q_{s,a,s'} \rangle$, where the first element denotes the task id, the second element represents the action, and the third element is the Q-value associated with the $\langle state, action \rangle$ pair (s, a), consistently leading to s' in deterministic MDP

With a zero discount factor ($\gamma = 0$), **ExZeroDiscount** takes $G(\mathcal{S}, \mathcal{E})$ as is - since Q-values of every (s, a) depends on the immediate reward, **ExZeroDiscount** simply now scales the i-th component of the reward function by weight w_i. After that, to compute optimal policy $\pi^*_{(t_c)}$, **ExZeroDiscount** performs a forward look-up and finds the optimal policy with the maximum cumulative reward. The rest of the proof is the same as that of Theorem 2.

With a non-zero discount factor ($\gamma \neq 0$), given $G(\mathcal{S}, \mathcal{E})$ **ExNonZeroDiscount** scales Q-values of task $t(i)$ by its corresponding weight w_i. The rest of the proof is same as that of Theorem 3. It can be shown that **BackTrackDAG** in **ExNonZeroDiscount** returns the correct bounds of the Q

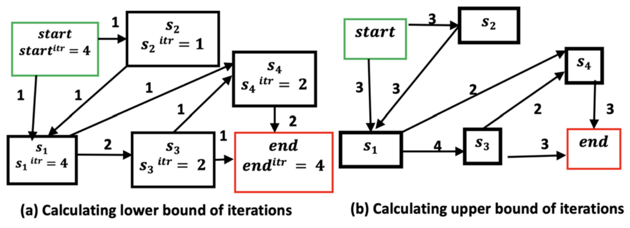


Fig. 5 Producing tighter $LB_{(s,a)}$ and $UB_{(s,a)}$ for $\epsilon = 4$

values of every (s, a) of t_c, where the Q value of (s, a) of t_c is within those bounds (if t_c is trained from scratch). Since **ForwardLookUp** returns all possible policies in the Q-table of t_c when training from scratch. It is therefore easy to infer that one of those policies that **ForwardLookUp** returns will have the highest cumulative reward and hence be the optimal.

4 Efficiency opportunities

In Sect. 4.1, we explore efficiency opportunities in Algorithm **ExNonZeroDiscount**. We refer to our discussion in Sect. 3.2.2 and describe how to produce tighter bounds of Q-values of (s, a) for t_c (as exact Q-values cannot be obtained without training). In Sect. 4.2, we present a parameterized algorithm **Greedy**-k that, depending on k, strikes a trade-off between running time and quality of the produced solution.

4.1 Tighter Q-value bounds estimation

With a non-zero discount factor $\gamma \neq 0$, we have presented **ExNonZeroDiscount** in Sect. 3.2.2 that leverages lower and upper bounds of Q-values of (s, a) for t_c in its design. By investigating the Q-function update formula in Eqs. 1 and 2, the following could be proved: lower bound and upper bound of Q-value only varies in one aspect, which is the number of times a state is visited during training (we call that itr). A pessimistic estimate gives a lower bound of Q-values of each (s, a), whereas, an optimistic estimate gives an upper bound. Section 3.2.2 presents a preliminary discussion on this, where the lower bound (upper bound) is designed considering the minimum (maximum ϵ) number of iterations $MinItr$ ($MaxItr$). In this section, we leverage the structure of $G(\mathcal{S}, \mathcal{E})$ to further make these numbers tighter, i.e., $MinItr$ ($MaxItr$) of each (s, a) bigger (smaller). Tighter bounds enable pruning more edges during **ForwardLookUp**.

4.1.1 Calculating tighter *MinItr*

Algorithm 6 calculates $MinItr$ of each edge and requires backtracking, just like **BackTrackDAG**. It admits $G(\mathcal{S}, \mathcal{E})$, t_c and its reward function, and then it assigns number of iterations (itr) to each s through backtracking. The reason is that to find $MinItr$ of (s, a) one has to first rely on the itr of s. Once each state has an iteration number assigned, it produces $MinItr$ of each edge (s, a) through backtracking, starting at the end state and finishing at the start state. During this process, it keeps track of the number of times s and its next state s' are visited by considering in-degrees and out-degrees of nodes in $G(\mathcal{S}, \mathcal{E})$.

Algorithm 6 Algorithms **MinItr**

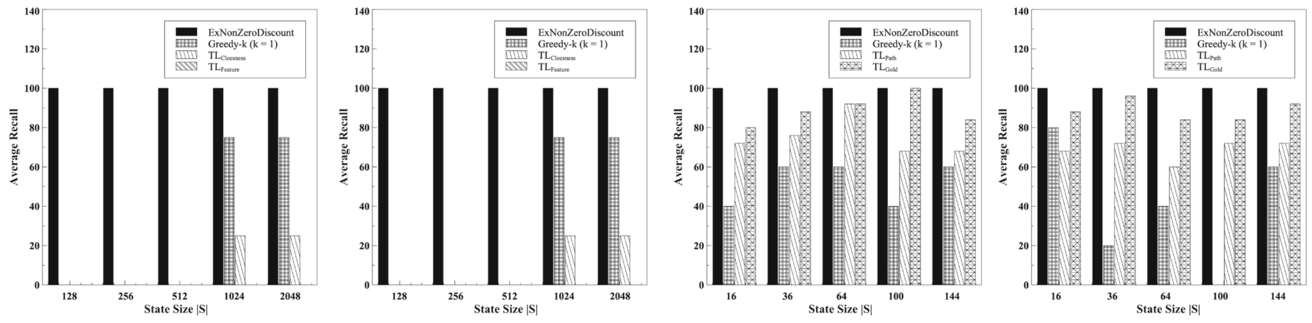
```
Require: $t_c$, $R_t(c)$, $G(\mathcal{S}, \mathcal{E})$, $\epsilon$
  for $s \in \mathcal{S}$ do
    if $s = s_{goal}$ or $s = s_{start}$ then
      $s^{itr} \leftarrow \epsilon$
    else if $s_{goal}$ cannot be reached by removing $s$ then
      $s^{itr} \leftarrow \epsilon$
    else
      $s^{itr} \leftarrow max(InDeg(s), OutDeg(s))$
  $s' \leftarrow s_{goal}$
  while $s' \neq s_{start}$ do
    $s \leftarrow backtrack(s', a)$
    for each $(s, a, s')$ do
      if $OutDeg(s) = 1$ & $InDeg(s) > 1$ then
        $MinItr(s, a) \leftarrow s^{itr}$
      else if $InDeg(s') = 1$ & $OutDeg(s') > 1$ then
        $MinItr(s, a) \leftarrow s'^{itr}$
      else
        $MinItr(s, a) \leftarrow 1$
    $s' \leftarrow s$
  return $MinItr(s, a)$
```

Using the example in Fig. 5a, this process assigns $s_4^{itr} = 2$, because that is the maximum of in-degree and out-degree of s_4, whereas, $s_1^{itr} = \epsilon = 4$, as removing s_1 will make the process not reach end state. Similarly, edge $s_4 \rightarrow end$ is assigned value 2, as s_4 has two incoming edges, but only one outgoing edge, therefore the aforementioned edge gets the same $MinItr$ as s_4^{itr}.

Running time. $MinItr$ is computed in **BackTrackDAG** without incurring any additional time.

4.1.2 Calculating tighter *MaxItr*

The default of adding $MaxItr$ is to add ϵ to every (s, a). Like $MinItr$, $MaxItr$ also backtracks from the end state and follows the same traversal over $G(\mathcal{S}, \mathcal{E})$. However, unlike $MinItr$, it does not need to compute the iteration number for states. This is because we assume that the maximum number of iterations per state is ϵ. If a (s, a) leads to s_{goal}, then $MaxItr$ of that (s, a) becomes $\epsilon - [InDeg(s_{goal}) - 1]$. If that is not the case, algorithm finds the sum of $MaxItr$ of each of the outgoing edges from s' (next state of what (s, a) leads to), it then differentiates between two cases. If the sum is smaller than ϵ, it assigns $MaxItr$ of that (s, a)



(a) Q-Learning on `Query` (b) SARSA on `Query` (c) Q-Learning on `Grid` (d) SARSA on `Grid`

Fig. 6 Recall results demonstrate that our exact solutions generalize to different applications with 100% recall, where transfer learning baselines perform poorly (in many cases, recall stays at 0), with **Greedy**-k ($k = 1$) having lower recall as expected. **Greedy**-k ($k = 2$) has the same recall as **ExNonZeroDiscount**, and omitted for lack of space

to be $\sum MaxItr - [InDeg(s') - 1]$. If the sum is greater than ϵ, then the algorithm intends to differentiate between two more cases: if in-degree of s' is 1, then it sets $MaxItr$ of (s, a) to ϵ. However, if in-degree of s' is > 1, it sets $MaxItr$ of (s, a) to $\epsilon - [InDeg(s') - 1]$. The rationale lies in how the number of visitations of a state-action pair (s, a) is tied to the maximum number of iterations, ϵ, and to the in-degree of the next state s' ($InDeg(s')$). However, when a state s' has multiple incoming edges, it implies that s' has been reached at least once through those alternative edges. This is the reason we subtract $InDeg(s')$ from ϵ.

Using the example in Fig. 5b, this process sets $MaxItr$ of $s_4 \rightarrow end$ and $s_3 \rightarrow end$ both to 3, whereas, the sum of $MaxItr$ of s_3 and s_1 is found to be 5 and 6, respectively. However, in-degree of $s_3 = 1$, whereas, that of $s_1 = 2$. Therefore,it sets $MaxItr$ of $s_1 \rightarrow s_3$ and $s_3 \rightarrow s_4$ to be 4 and 2 respectively.

Running time. $MaxItr$ could be calculated inside **`BackTrackDAG`** without incurring any additional time.

4.2 Parameterized algorithm Greedy-*k*

In scenarios where the state and action space are exponentially large, getting an exact solution becomes impractical. To address that, we introduce a parameterized algorithm **Greedy**-k that is designed to strike a balance between efficiency and quality. The algorithm is designed with a parameter k, where a small value of k does a greedy (shallow) look up over G allowing faster running time, and vice versa. When $k = |\mathcal{A}|$, the algorithm yields an exact solution. It takes G that stores Q-values of different reward functions used to express $R(c)$, and performs the classical inference steps of the RL algorithms for a given input k. It starts at s_{start} and retains the top-k actions (based on the Q-values of each (s, a)) as k transitions from s at each state s. The process ends when it leads to s_{goal}. Once this process is over, it computes the cumulative rewards for each of the unique paths and outputs that as the optimal policy which has the highest cumulative reward.

Using the Q-tables of the example in Table 2, when $k = 1$, **Greedy**-k will leverage the Q-tables of t_1 and t_2 independently and greedily select the top-1 transition from each state to reach the goal. As an example, if it starts at R_8, it will change one of the word in the current query based on the highest Q value of t_2 and go to R_{16}. From there, it will follow Q-value of t_1 and add a new word to the current query to reach R_{48}. This process will continue until the goal state is reached.

Running time. **Greedy**-k could take, in the worst case, $\mathcal{O}(k^{|\mathcal{S}|})$ time to run. When $k = 1$, it takes linear time in the size of the state space.

5 Experimental evaluation

We formulate four to empirically validate the effectiveness of our approach: **(RQ1)** Quality of our solution compared to the ground truth (ie, training policies from scratch) in terms of cumulative reward and recall. We implement several baselines, including transfer learning based solutions and compare them against our proposed solutions.**(RQ2)** Generality of our proposed solution, where we change the original reward functions by scaling them by arbitrary and constant positive weights. We also implement DQN, a deep RL based solution to demonstrate that our proposed framework remains effective. **(RQ3)** Scalability considering speed up and pruning effectiveness. **(RQ4)** Time taken to build the graph data model and the memory usage of our graph data model compared to training from scratch. We focus on two applications (interactive query refinement and robot movement on a grid) to demonstrate generality of our framework. We keep the default number of reward functions to 2, and vary the num-

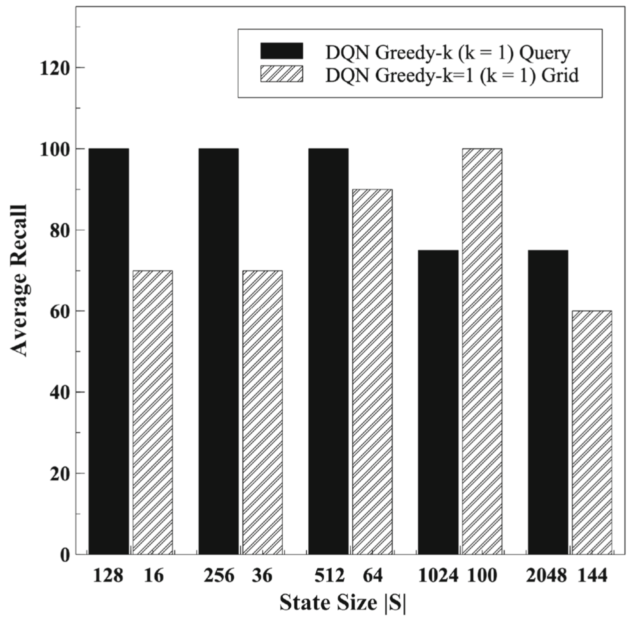


Fig. 7 Recall of `DQN Greedy` ($k = 1$). It does not achieve 100% recall as expected

ber of reward functions whenever appropriate. We dive deep on the above **RQs** in the subsequent subsections. We also refer the reader to our technical report [2] for more details.

5.1 Experimental setup

Experiment Settings. All algorithms are implemented in Python 3.11.1 on a MacOS Monterey 12.16.3 with 2.7 GHz Quad-Core Intel Core i5 Processor and 8 GB RAM. Our code and data are publicly available.[1] All numbers are presented as an average over 10 runs.

5.1.1 Evaluated applications.

We design two applications: Data exploration through Query Refinement (`Query`) and Grid World (`Grid`).

Query Refinement. The process takes a keyword query, a large set of Amazon Reviews (Electronics category),[2] and a set of X categorical features (e.g., battery life, fitness capabilities) as inputs. The goal is to guide the user in the review exploration process, while interactively refining the query, such that additional "similar" reviews could also be explored. During training, a representative review for a target item is selected to measure the quality of explored reviews. In order to obtain the refined query or to produce similarity between a current review and the "target" review, a Large Language Model (LLM) is prompted.[3] Both query and all reviews in the dataset are embedded into a vector space using the Hugging Face Transformers Library [48].

Process of Query Refinement. At each step, it first calculates the review (denoted as the "chosen review") that is most similar to the current query based on the cosine similarity between the query vector and the review vectors. Then, given the feature space, the LLM is prompted to produce a similarity score between each feature and the "chosen review". If the produced similarity score exceeds 0.5, the feature is considered to be present in the "chosen review"; else it is considered absent. Now the LLM is prompted to either add a new word to the current query, generalize the existing query word, or change the current word order to come up with a new query q', and the aforementioned process repeats. The process ends when the "chosen review" is similar enough to the "target review".

State Space of Query Refinement. The state space is designed by considering all possible 2^X feature combinations. A review belongs to one of these 2^X states, given the feature set. Possible interactions are the set of actions (add word, generalize, change word order) to refine an initial query.

Tasks in Query Refinement. Two tasks are pre-trained: t_1 to find closeness between the "chosen review" at s' and the "target": $R_t(1)(s,a,s') = Cosine(s, s_{target}) - Cosine(s', s_{target}))$; t_2 to satisfy as many features as possible: $R_t(2)(s,a,s') = \sum\{\mathbb{1}_{s'x} | sim(s'^X, x) \geq 0.5, x \in X$ }. The LLM tries to get to the next state s' represented by a "chosen review" which satisfies as many of given X features as possible. The reward function of the new task t_c is $R_t(1) + R_t(2)$.

Grid World. It is a synthetic environment where the states are grid cells and the actions are moving directions. Two tasks are pre-trained: t_1: find the shortest path to a grid cell marked as "target", t_2: collect as many gold as possible in the grid. The reward function of t_1 is $R_t(1)(s,a,s') = dis(s, s_{target}) - dis(s', s_{target})$ which captures the marginal gain the move (s,a) leads to in terms of being close to s_{target}. We use the Manhattan distance. The reward function of t_2 is $R_t(2)(s,a,s') = \sum\{\mathbb{1}_{s_{gold}} | dis(s', s_{gold}) \leq 2 \ \& \ s_{gold} \notin g\}$ where g is a set of already visited golds. $R_t(2)$ aims to finds the number of new golds the transition could give rise to considering the immediate neighborhood of s' up to 2 hops. The reward function of the new task t_c is simply $R_t(1) + R_t(2)$.

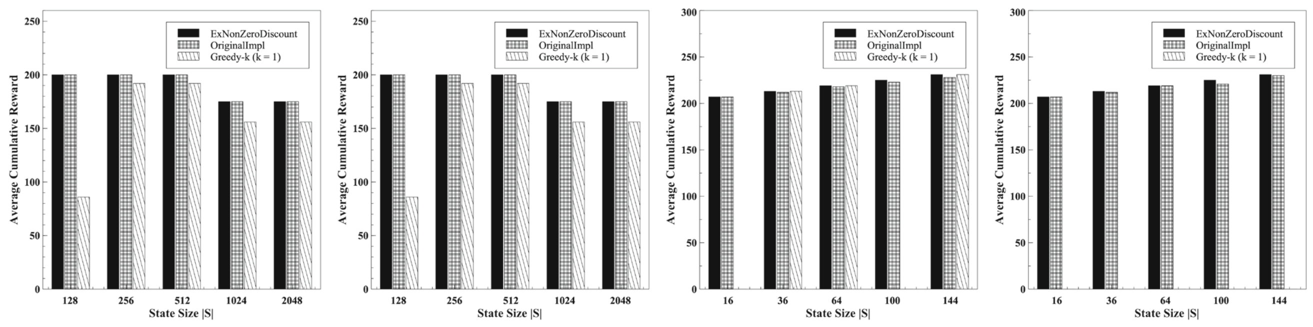
5.1.2 Evaluated algorithms

We evaluate the following algorithms.

[1] https://github.com/PolicyReusablityInRL/Policy-Reusablityhttps://github.com/PolicyReusablityInRL/Policy-Reusablity.

[2] http://jmcauley.ucsd.edu/data/amazon/.

[3] *We leverage* `gpt-3.5-turbo` *in this experiment.*



(a) Q-Learning on `Query` (b) SARSA on `Query` (c) Q-Learning on `Grid` (d) SARSA on `Grid`

Fig. 8 Cumulative rewards increase with increasing environment size and **ExNonZeroDiscount** 's reward is never lower than implementation from scratch. **Greedy**-k ($k = 1$) works well at times, but gives 0 reward when it gets stuck and can not reach the goal

- **Transfer Learning** [25, 26]. The reward function of the new task is applied on the optimal policy obtained from a pre-trained task, integrating it as part of the new task's reward structure. This allows the algorithm to utilize knowledge from the pre-trained task while adapting it to the new environment. As an example, TL_{Gold} refers to the optimal policy obtained for the gold collection task in the `Grid`. While this policy maximizes rewards in the original task, its cumulative reward is recalculated based on the new task's objectives, ensuring that transferred knowledge remains relevant and contributes effectively to the learning process.
- **Training from Scratch** (**OriginalImpl**). Using a particular RL approach, perform training then inferencing.
- **ExZeroDiscount**, **ExNonZeroDiscount**, and **Greedy**-k.

The aforementioned algorithms are implemented with Q-learning [46], SARSA [38], and DQN [3], a Deep-RL algorithm that approximates the Q-table using a neural network. Therefore, training from scratch for DQN involves obtaining the optimal policy given the reward function. Contrarily, our approach `DQN Greedy` for DQN, outputs one optimal policy based on each reward function, then enumerates all possible paths based on combining them, and outputs the one with the highest cumulative reward. Consequently, `DQN Greedy` is a heuristic that works reasonably well in practice.

Varied parameters and defaults. We vary the size of state space $|\mathcal{S}|$ (default=128 for `Query` and default = 64 for `Grid`), size of action space $|\mathcal{A}|$ (default=2), discount factor γ (default=0.99), number of reward functions l (default=2, $R_t(c) = R_t(1) + R_t(2)$), k in **Greedy**-k(default=1). When m is varied, the different reward functions are created through linear transformation of $R_t(1)$ and $R_t(2)$ and $R_t(c) = R_t(1) \pm R_t(2) \pm R_t(m)$ (chosen randomly). Learning rate is set to $\alpha = 0.1$ and number of episodes is set to $\epsilon = 1000$.

5.1.3 Measures

We report several measures. **Cumulative reward** is the sum of reward the RL agent accumulates over a specific trajectory. **Recall** of an algorithm A is computed by calculating the cumulative reward of the optimal policy produced by A and comparing it with the optimal policy when trained from scratch. If the cumulative reward of the policy obtained from A is not smaller than that of the latter, a recall of 1 is assigned, a 0 is assigned otherwise. **Speed up** of an algorithm A is calculated as follows: (training + inference time) if done from scratch, divided by the running time of A. **Pruning percentage** is studied only for our designed exact solutions **ExZeroDiscount** and **ExNonZeroDiscount**, and is:

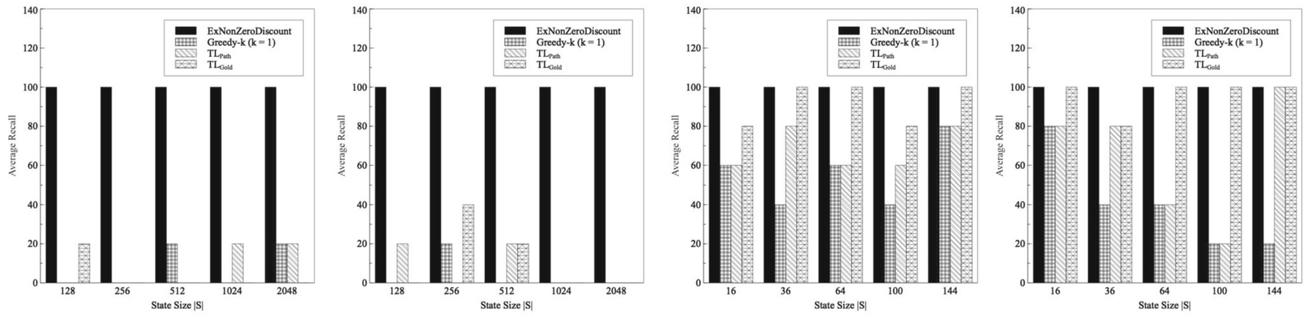
$$\left(1 - \frac{\#\text{paths processed by the algorithm}}{\#\text{paths processed by the baseline}}\right) \times 100$$

Memory usage is calculated in MB. **Preprocessing time** is presented in seconds.

5.2 Experimental results

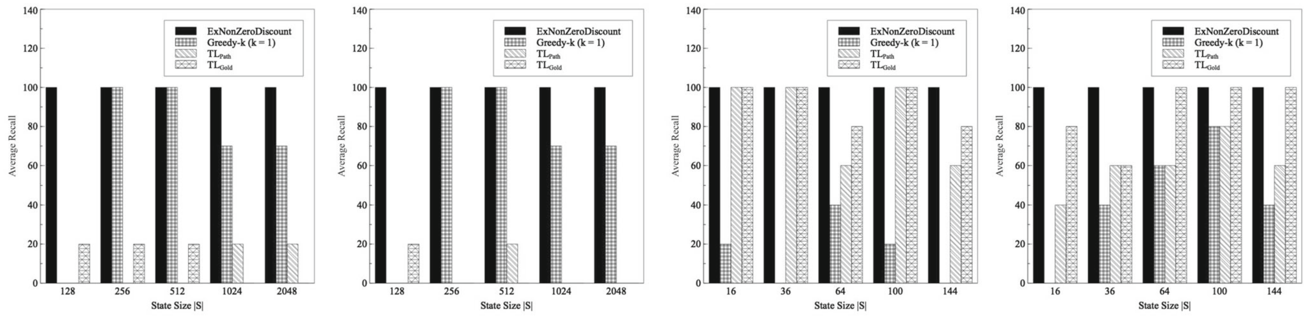
5.2.1 Results on RQ1—Quality

We explore the quality of our solution compared with the ground-truth, i.e., learning RL policies from scratch. Figure 6 shows the recall results for the `Grid` and `Query` applications across the following algorithms: **ExNonZeroDiscount**, `TL`, **Greedy**-k. Each algorithm is reported for Q-Learning, SARSA. Figure 7 displays the recall results for Deep-RL for **Greedy**-k algorithm. Our first and foremost observation is, consistent with our theoretical analysis, our designed exact solutions **ExZeroDiscount**, **ExNonZeroDiscount** always achieve 100% recall. The baselines perform poorly in general except in a few cases, and **Greedy**-k achieves lower recall consistent to our theoretical analysis. Regarding the average cumulative reward, Fig. 8 presents a



(a) Q-Learning on `Query` (b) SARSA on `Query` (c) Q-Learning on `Grid` (d) SARSA on `Grid`

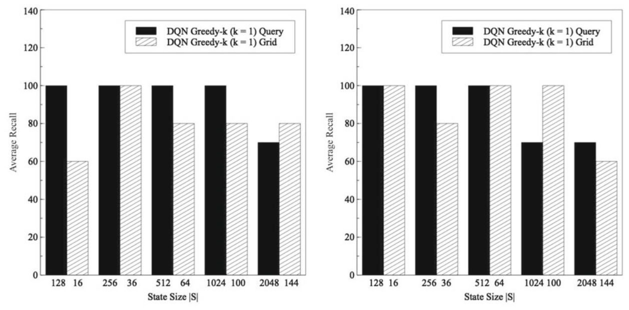
Fig. 9 Recall results demonstrate that **ExNonZeroDiscount** generalize for linear reward combinations of individual reward functions with any weights $(\alpha, \beta = (2, 3))$ with 100% recall, where transfer learning baselines perform poorly, with **Greedy**-k $(k = 1)$ having lower recall as expected



(a) Q-Learning on `Query` (b) SARSA on `Query` (c) Q-Learning on `Grid` (d) SARSA on `Grid`

Fig. 10 Recall results demonstrate that **ExNonZeroDiscount** generalize for linear reward combinations of individual reward functions with any weights $(\alpha, \beta = (0.5, 1))$ with 100% recall, where transfer learning baselines perform poorly, with **Greedy**-k $(k = 1)$ having lower recall as expected

subset of results that are representative. We generally observe an increase in cumulative reward when we increase the size of the environment, which is intuitive. We also observe that **ExNonZeroDiscount** never has a smaller cumulative reward than **OriginalImpl**, i.e., training from scratch. For **Greedy**-k, the cumulative reward is generally lower for $k = 1$ and becomes identical to **ExNonZeroDiscount** when $k = 2$. This last observation is again consistent with our theoretical analysis, when all 2 actions are considered in **Greedy**-k, it becomes identical to the optimal solution.



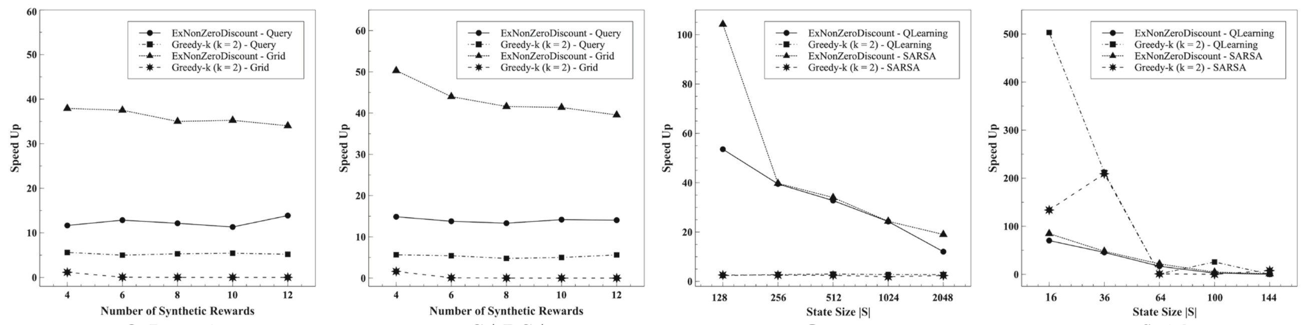
(a) $\alpha, \beta = (2, 3)$ (b) $\alpha, \beta = (0.5, 1)$

Fig. 11 Recall for DQN **Greedy**-k (k = 1) under different reward weights. As expected, it does not achieve 100% recall

5.2.2 Results on RQ2—Generality

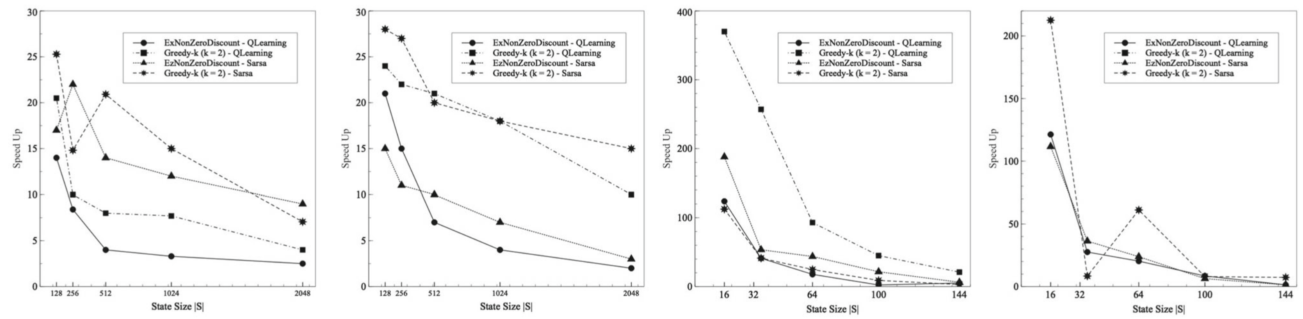
In this section, we present the generality of the proposed solution framework. As two key representatives, we consider the Grid and Query refinement applications and change the reward function by considering arbitrary positive weights. We also have demonstrated how the proposed solution behaves when the underlying RL framework is DQN based RL. The proposed framework is designed to generalize to any arbitrary reward function or reinforcement learning (RL) architecture. However, the optimality guarantee is specifically examined for Q-table-based RL methods, focusing on new tasks with reward functions that are linear functions of the reward functions from tasks for which training data is available.

For demonstrating generalization, we extend the combined reward function from $R_c = R_1 + R_2$ to $R_c = \alpha \times R_1 + \beta \times R_2$. Figures 9 and 10 shows the results for $\alpha = 2, \beta = 3$ and $\alpha = 0.5, \beta = 1$ respectively. As it is shown, ExNonZeroDiscount remains as the only algorithm achieving 100% recall in all the cases, showing the fact that



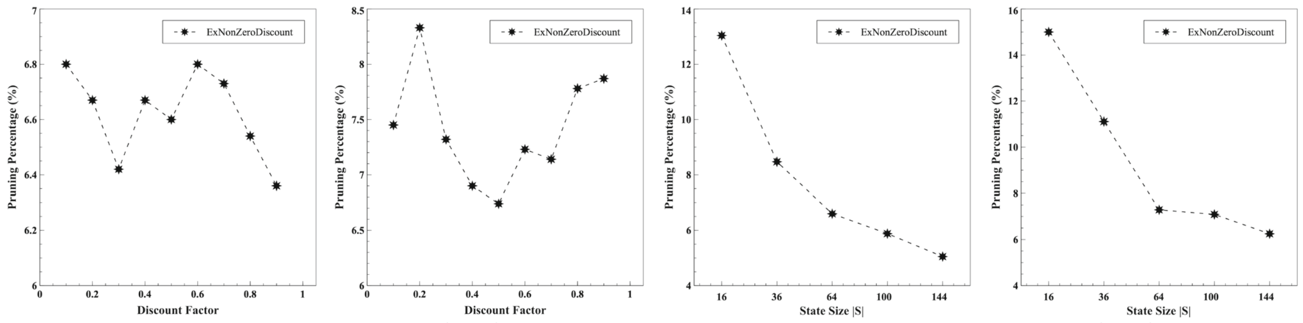
(a) Q-Learning (Varying #Rewards) (b) SARSA (Varying #Rewards) (c) `Query` (Varying #State) (d) `Grid` (Varying #State)

Fig. 12 Speed up. Proposed solutions are typically 6 to 50 times faster than training from scratch



(a) Query ($\alpha, \beta = (2, 3)$) (b) Query ($\alpha, \beta = (0.5, 1)$) (c) `Grid` ($\alpha, \beta = (2, 3)$) (d) `Grid` ($\alpha, \beta = (0.5, 1)$))

Fig. 13 Speed up when the reward components are scaled up by arbitrary positive weights for both `Query` and `Grid` applications. The results demonstrate that our proposed solutions are 5 to 25 times faster compared to training from scratch



(a) Q-Learning on `Grid` (Discount Factor) (b) SARSA on `Grid` (Discount Factor) (c) Q-Learning on `Grid` (State Size) (d) SARSA on `Grid` (State Size)

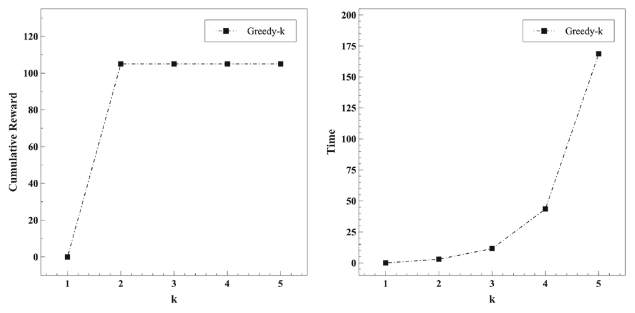
Fig. 14 Pruning Percentage (%). We observe a pruning percentage around 8%

our proposed algorithm is generalizable for any arbitrary positive weights of α and β.

Figure 11 shows that Greedy-k algorithm using Deep neural network (DQN architecture) is still unable to achieve 100% recall as we extend the reward function to the parameterized form with positive weights of α and β.

5.2.3 Results on RQ3—Speed up

We explore speed up of the proposed solutions. We vary the number of reward functions, # states, and report speed-up. Figure 12 shows the results of speed up in `Query` and `Grid` applications. We observe that while the speed-up is variant, our solutions are in general 6 to 50 times faster in `Grid` than training from scratch. Because of the sparsity of the Q-table, the speed up is less in `Query` but is always more than $3X$. Figure 13 shows the results of speedup when the



(a) Cumulative Reward (b) Time

Fig. 15 Quality scalability trade off of **Greedy**-k in `Grid` application

weights are scaled up by different positive weights for both `Query` and `Grid` applications. The results demonstrate that our proposed solutions are 5 to 25 times faster on an average compared to training from scratch.

Pruning Effectiveness. We explore the effectiveness of our pruning strategy in **ExNonZeroDiscount**. We vary the environment size and discount factor γ and report the pruning percentage (Fig. 14). We observe a percentage around 8%. In case $\gamma = 0$, it is 100%, which is consistent with our theoretical analysis.

Quality scalability trade-off of Greedy-k. In this section, we demonstrate how **Greedy**-k behaves with increasing k in terms of running time and the average cumulative reward.

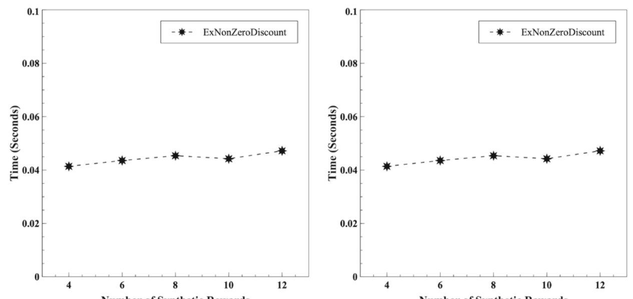
Figure 15 illustrates the relationship between time and cumulative reward for the `Grid` application when employing the **Greedy**-k approach with varying values of k. It is evident that as k increases, the running time also increases somewhat non-linearly, which is consistent with our theoretical analysis. What is interesting from the Fig. 15a is that the cumulative reward does not increase beyond $k = 2$. This demonstrates that a small k value may still be quite suitable to achieve the best trade-off between quality and running time in **Greedy**-k.

5.2.4 Results on RQ4—Preprocessing overhead

In these set of experiments we evaluate the pre-processing overhead of the proposed framework—we measure the time taken to produce the graph structure described in Sect. 3.1, and the memory it consumes.

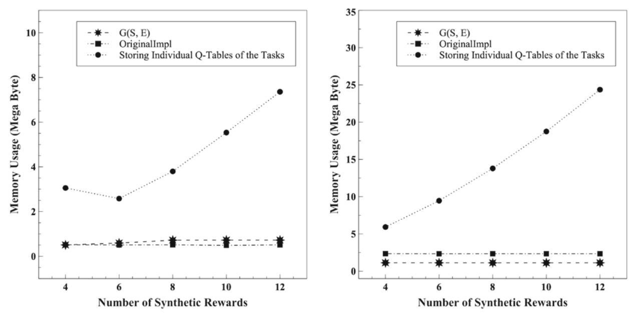
Preprocessing Time. We vary the number of tasks (# of reward functions) and present the time that our graph model (G in Sect. 3.1) requires. Figure 16 demonstrates that the preprocessing or indexing time of **ExNonZeroDiscount** minimally changes with increasing number of tasks and scales very well.

Memory. We vary the number of reward functions and present memory usage that our graph model (G in Sect. 3.1) requires, and compare that with two alternatives, (i) where



(a) Preprocessing time on `Grid` (b) Preprocessing time on `Query`

Fig. 16 The preprocessing time of the graph data structure is linear with increasing number of tasks (#rewards) and scales very well



(a) Memory Usage on `Grid` (b) Memory Usage on `Query`

Fig. 17 Proposed graph model is lean in storage

all l individual Q-tables of the pre-trained tasks are stored, (ii) when training a policy from scratch. Figure 17 shows that consistent with our theoretical analysis, it is evident that our graph data structure is lean and does not require more storage than training from scratch.

6 Related work

Policy Reusability in RL. RL is used in a range of scenarios within dynamic and uncertain environments, as highlighted in various studies [29] [16] [12]. These applications span from controlling and monitoring traffic lights [47], automating exploratory data analysis [5], discovering user groups [33], to tackling complex task planning [25]. To make RL policies reusable, several training-based approaches are proposed, including Multi-Task RL (MTRL) [43], Option-Critic Architecture [4], and Meta RL [14]. MTRL aims to simultaneously learn policies for multiple related tasks, so that knowledge gained in one task can inform and accelerate learning in others. The option-critic architecture learns both policies over primitive actions as well as policies over temporally-extended actions called "options" to enable more efficient transfer between related tasks. Meta RL learns policies that can quickly adapt to new tasks by leveraging prior experience on a distribution of related tasks, enabling faster learning on new tasks compared to learning each task independently from scratch. *Contrarily, our approach is an inference-only approach that reuses pre-trained models without the need for any re-training and comes with guarantees.*

Machine Learning Reusability. The concept of ML reusability has garnered significant attention from researchers. This focus encompasses a wide range of areas, including the reuse of pre-trained models like BERT for various NL understanding tasks [9], the development of federated learning to maintain data privacy while reusing models [23, 28], the automation of ML pipelines with AutoML to enhance model reusability [42], the integration of ML models with databases for efficient data retrieval [17], and the challenge of reusing data for different ML tasks in DatomIQ [37] and MAPLE [51]. Additionally, researchers [44, 50] have addressed the sample efficiency problem in RL and explored knowledge transfer through Case-Based Reasoning, particularly through the abstraction of reusable cases from RL. *In contrast to these generic ML approaches, ours specifically delves into MDP structures, and enables the reusability of pre-trained models.*

Transfer Learning. Transfer learning (TL) is a vital concept in ML and RL, allowing models to apply knowledge acquired from one task or domain to related ones [6, 25, 26, 35, 36, 38, 39, 41]. While there does not exist a single definition of (TL), in [39], a comprehensive framework for classifying and analyzing transfer learning methods in the context of RL is presented, emphasizing the improvement of learning performance in related tasks. [36] introduces the "replacing trace" concept within the eligibility traces framework, demonstrating its superiority in facilitating faster and more reliable learning compared to conventional traces. In [35], the authors focus on enabling RL agents to handle multiple tasks, specifically composite sequential decision tasks, by introducing a novel learning algorithm and modular architecture that facilitate the transfer of knowledge from elemental tasks to efficiently solve complex, systematically composed tasks. We implement TL in Sect. 5 motivated by prior works [25, 26]. *Contrarily, ours is the first policy reusability approach with theoretical guarantees for any TDL based RL techniques with an adaptation to Deep-RL.*

7 Discussion and open problems

7.1 Removing cycles in $G(\mathcal{S}, \mathcal{E})$ inside `BackTrackDAG`

We have demonstrated that for **`BackTrackDAG`** to work accurately, $G(\mathcal{S}, \mathcal{E})$ needs to be cycle free. However, when the action space allows the agent move freely in the environment, the resultant graph is more likely to have cycles than not. If the problem of making $G(\mathcal{S}, \mathcal{E})$ free of cycles were related to removing the minimum number of edges ($\langle state, action\rangle$) from $G(\mathcal{S}, \mathcal{E})$ - that would relate to the minimum feedback arc set problem [8] or its weighted version [11]. Minimum feedback arc set problem is a classical combinatorial optimization problem and is known to be NP-hard [13]. There does exist efficient heuristics [10] as well that solve the problem in polynomial time. Contrary to removing minimum number of edges in $G(\mathcal{S}, \mathcal{E})$, our goal however is to remove those edges that have the least contribution in Q-values. An open problem is thus to investigate how Q-values of $\langle state, action\rangle$ accumulates recursively to decide an appropriate order of elimination of edges ($\langle state, action\rangle$) to make $G(\mathcal{S}, \mathcal{E})$ free of cycles.

7.2 Theoretical guarantees for linear approximation of Q-function

In the paper, we proved that the proposed framework returns optimal policies for Q-learning and SARSA. We noted that the framework extends to Deep-RL but it does not provide any optimality guarantee. Deep-RL uses neural network to approximate the Q-values of $\langle state, action\rangle$. Neural networks are highly complex non-linear models that do not exhibit many mathematical properties (e.g., convexity). However, if m Q-tables of the pre-trained tasks $(t_i, t_j, \ldots t_m)$ were approximated using m linear functions (e.g., linear regression) over a fixed set of features, if the same feature set is also assumed for t_c, an interesting open problem is to theoretically investigate the relationship between the feature weights of $t_i, t_j, \ldots t_m$ and that of t_c. With such relationship being established, our framework would be extended to solve the problem under Q-function approximation using linear models.

7.3 Theoretical guarantees for deep RL

We begin by extending our analysis to single-layer neural networks, where the weights approximate the state-action Q-values, effectively serving as a compact representation of a Q-table. When the network is sufficiently expressive, i.e., it has at least as many weights as there are state-action pairs in the Q-table, the approximation error can be negligible, and the total error is bounded by the training error and generalization error. If the network is under-parameterized, an additional approximation error arises due to its limited capacity to represent the full Q-table. The total error in such cases is the sum of the training error, generalization error, and approximation error.

For multi-layer neural networks, the analysis extends by leveraging their enhanced representational capacity through non-linear transformations. These networks maintain the foundational principles of TD learning, where the training process minimizes a loss function (e.g., temporal difference loss) to approximate the Q-value function. Recursive dependencies introduced by the Bellman equation are handled through backpropagation, propagating updates across layers to refine Q-value approximations. The theoretical guarantees established for single-layer networks extend to multi-layer networks under three conditions: (i) the network architecture must have sufficient capacity to approximate the Q-value function, (ii) training must converge to a fixed point satisfying the Bellman equation, and (iii) the approximation error, including generalization error, must remain bounded. These conditions ensure that the theoretical results for single-layer networks generalize directly to multi-layer networks, allowing for more complex Q-value functions to be approximated.

However, these guarantees apply specifically to RL methods based on TD learning and assume finite state-action spaces and deterministic environments. Extending this framework to continuous spaces, non-deterministic MDPs, or other DRL methods, such as policy gradient approaches, remains an open challenge for future work.

7.4 Non-linear functions and non-deterministic MDPs

An interesting open problem is to investigate how to extend the proposed approach to reward functions that non-linearly combine m different reward functions coming from the pre-trained tasks. To that end, we are investigating both necessary and sufficient conditions of the individual reward functions, as well as that of the operators hat combine them. In the case, where the m reward functions are combined using a linear function, We are still interested to extend our approach to non-deterministic MDPs where a ($\langle state, action \rangle$) leads to multiple possible transitions, each with its associated probability. We believe that perhaps a randomized algorithmic approach would be more suitable to tackle this latter situation.

8 Conclusion

We explore the reusability of pre-trained models in RL and propose solutions that generalize to both TDL, Deep-RL and to any number of arbitrary reward functions. We introduce an expressive and lossless graph data model and develop efficient policy combination algorithms with rigorous theoretical guarantees. Our experimental results validate the generality of the framework in two high fidelity applications and corroborate all theoretical results. These findings underline the potential to yield results on par with policies trained from scratch, highlighting its significance in advancing the applicability of pre-trained RL models in practice. This work paves the way for more resource-efficient and effective RL applications in real-world contexts.

Acknowledgements The work of Sepideh Nikookar, Sohrab Namazi Nia, and Senjuti Basu Roy are supported by the following funding agencies: (1) National Science Foundation award number(s): 1942913, 2007935, 1814595 (2) Office of Naval Research award number(s): N000141812838, N000142112966, N000142412466.

References

1. Abbeel, P., Coates, A., Quigley, M., Ng, A.: An application of reinforcement learning to aerobatic helicopter flight. In: Advances in Neural Information Processing Systems 19 (2006)
2. Anonymous Technical report. https://www.dropbox.com/scl/fo/ikd6w88mv2jsa11bdlj7v/h?rlkey=lrpsox76rhyvvmosu4m7ckruf&dl=0 (2024)
3. Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: Deep reinforcement learning: a brief survey. IEEE Signal Process. Mag. **34**(6), 26–38 (2017)
4. Bacon, P.L., Harb, J., Precup, D.: The option-critic architecture. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol 31 (2017)
5. Bar El, O., Milo, T., Somech, A.: Automatically generating data exploration sessions using deep reinforcement learning. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (2020)
6. Barrett, S., Taylor, M.E., Stone, P.: Transfer learning for reinforcement learning on a physical robot. In: Ninth International Conference on Autonomous Agents and Multiagent Systems-Adaptive Learning Agents Workshop (AAMAS-ALA), vol. 1 (2010)
7. Cai, H., Ren, K., Zhang, W., Malialis, K., Wang, J., Yu, Y., Guo, D.: Real-time bidding by reinforcement learning in display advertising. WSDM '17, pp. 661–670 (2017)
8. Charbit, P., Thomassé, S., Yeo, A.: The minimum feedback arc set problem is np-hard for tournaments. Comb. Probab. Comput. **16**(1), 1–4 (2007)
9. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
10. Eades, P., Lin, X., Smyth, W.F.: A fast and effective heuristic for the feedback arc set problem. Inf. Process. Lett. **47**(6), 319–323 (1993)
11. Fomin, F., Lokshtanov, D., Raman, V., Saurabh, S.: Fast local search algorithm for weighted feedback arc set in tournaments. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 24, pp. 65–70 (2010)
12. Garcıa, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**(1), 1437–1480 (2015)
13. Garey, M.R., Johnson, D.S.: Computers and intractability. A Guide to the (1979)

14. Gupta, A., Mendonca, R., Liu, Y., Abbeel, P., Levine, S.: Meta-reinforcement learning of structured exploration strategies. In: Advances in Neural Information Processing Systems 31 (2018)
15. Jiang, C., Li, X., Lin, J., Liu, M., Ma, Z.: Adaptive control of resource flow to optimize construction work and cash flow via online deep reinforcement learning. Autom. Constr. **150**, 104817 (2023)
16. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996)
17. Kraska, T., Beutel, A., Chi, EH., Dean, J., Polyzotis, N.: The case for learned index structures. In: Proceedings of the 2018 International Conference on Management of Data, pp. 489–504 (2018)
18. Lezzar, F., Zidani, A., Atef, C.: A collaborative web-based application for health care tasks planning. In: Proceedings of the 4th International Conference on Web and Information Technologies, Citeseer, pp. 30–39 (2012)
19. Li, Y.: Deep reinforcement learning: an overview. arXiv preprint arXiv:1701.07274 (2017)
20. Lu, Y., Kandula, S., König, A.C., Chaudhuri, S.: Pre-training summarization models of structured datasets for cardinality estimation. Proc. VLDB Endow. **15**(3), 414–426 (2021)
21. Mahadevan, S., Theocharous, G.: Optimizing production manufacturing using reinforcement learning. In: Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference, Citeseer, vol. 372, p. 377 (1998)
22. Mao, H., Alizadeh, M., Menache, I., Kandula, S.: Resource management with deep reinforcement learning. In: Proceedings of the 15th ACM Workshop on Hot Topics in Networks, pp. 50–56 (2016)
23. McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, BA.: Communication-efficient learning of deep networks from decentralized data. In: Artificial intelligence and statistics, PMLR, pp 1273–1282 (2017)
24. Nakandala, S., Kumar, A.: Nautilus: An optimized system for deep transfer learning over evolving training datasets. In: Proceedings of the 2022 International Conference on Management of Data, pp. 506–520 (2022)
25. Nikookar, S., Sakharkar, P., Smagh, B., Amer-Yahia, S., Roy, S.B.: Guided task planning under complex constraints. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 833–845 , IEEE (2022)
26. Nikookar, S., Sakharkar, P., Somasunder, S., Basu Roy, S., Bienkowski, A., Macesker, M, Pattipati, KR., Sidoti, D.: Cooperative route planning framework for multiple distributed assets in maritime applications. In: Proceedings of the 2022 International Conference on Management of Data, pp. 1518–1527 (2022b)
27. Obando-Ceron, J.S., Castro, P.S.: Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research. CoRR arXiv:2011.14826 (2020)
28. Omidvar-Tehrani, B., Personnaz, A., Amer-Yahia, S.: Guided text-based item exploration. In: Hasan M.A., Xiong, L. (eds.) Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17–21, 2022, pp. 3410–3420. ACM (2022)
29. Pednault, E., Abe, N., Zadrozny, B.: Sequential cost-sensitive decision making with reinforcement learning. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 259–268 (2002)
30. Pertsch, K., Lee, Y., Lim, J.: Accelerating reinforcement learning with learned skill priors. In: Conference on Robot Learning, PMLR, pp. 188–204 (2021)
31. Pineau, J.: Reproducible, reusable, and robust reinforcement learning. In: Advances in Neural Information Processing Systems (2018)
32. Rosset, C., Jose, D., Ghosh, G., Mitra, B., Tiwary, S.: Optimizing query evaluations using reinforcement learning for web search. In: The 41st International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 1193–1196 (2018)
33. Seleznova, M., Omidvar-Tehrani, B., Amer-Yahia, S., Simon, E.: Guided exploration of user groups. Proc. VLDB Endow. (PVLDB) **13**(9), 1469–1482 (2020)
34. Shortreed, S.M., Laber, E., Lizotte, D.J., Stroup, T.S., Pineau, J., Murphy, S.A.: Informing sequential clinical decision-making through reinforcement learning: an empirical study. Mach. Learn. **84**, 109–136 (2011)
35. Singh, S.P.: Transfer of learning by composing solutions of elemental sequential tasks. Mach. Learn. **8**, 323–339 (1992)
36. Singh, S.P., Sutton, R.S.: Reinforcement learning with replacing eligibility traces. Mach. Learn. **22**, 123–158 (1996)
37. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. ACM SIGMOD Rec. **34**(4), 42–47 (2005)
38. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
39. Taylor, M.E., Stone, P.: Transfer learning for reinforcement learning domains: a survey. J. Mach. Learn. Res. **10**(7), 1633–1685 (2009)
40. Tlili, A., Chikhi, S.: Risks analyzing and management in software project management using fuzzy cognitive maps with reinforcement learning. Informatica **45**(1), 1–24 (2021)
41. Torrey, L., Shavlik, J.: Transfer learning. In: Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques, IGI global, pp. 242–264 (2010)
42. Vanschoren, J., Van Rijn, J.N., Bischl, B., Torgo, L.: Openml: networked science in machine learning. ACM SIGKDD Explor. Newsl. **15**(2), 49–60 (2014)
43. Vithayathil, Varghese N., Mahmoud, Q.H.: A survey of multi-task deep reinforcement learning. Electronics **9**(9), 1363 (2020)
44. Von Hessling, A., Goel, A.K.: Abstracting reusable cases from reinforcement learning. In: ICCBR Workshops, pp. 227–236 (2005)
45. Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., Freitas, N.: Dueling network architectures for deep reinforcement learning. In: International Conference on Machine Learning, PMLR, pp. 1995–2003 (2016)
46. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**, 279–292 (1992)
47. Wei, H., et al.: Intellilight: a reinforcement learning approach for intelligent traffic light control. In: SIGKDD (2018)
48. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al.: Huggingface's transformers: state-of-the-art natural language processing. arXiv preprint arXiv:1910.03771 (2019)
49. Youngmann, B., Amer-Yahia, S., Personnaz, A.: Guided exploration of data summaries. Proc. VLDB Endow. **15**(9), 1798–1807 (2022)
50. Yu, Y.: Towards sample efficient reinforcement learning. In: IJCAI, pp. 5739–5743 (2018)
51. Yu, Y., Chen, S.Y., Da, Q., Zhou, Z.H.: Reusable reinforcement learning via shallow trails. IEEE Trans. Neural Netw. Learn. Syst. **29**(6), 2204–2215 (2018)
52. Zhao, C., He, Y.: Auto-em: End-to-end fuzzy entity-matching using pre-trained deep models and transfer learning. In: The World Wide Web Conference, pp. 2413–2424 (2019)