REWARD SHAPING FOR EFFICIENT EXPLORATION AND ACCELERATION OF LEARNING IN REINFORCEMENT LEARNING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MELİS İLAYDA BAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
OPERATIONAL RESEARCH

JULY 2022

Approval of the thesis:

REWARD SHAPING FOR EFFICIENT EXPLORATION AND ACCELERATION OF LEARNING IN REINFORCEMENT LEARNING

submitted by **MELİS İLAYDA BAL** in partial fulfillment of the requirements for the degree of **Master of Science in Operational Research Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Sinan Gürel
Head of Department, **Operational Research** _____

Prof. Dr. Cem İyigün
Supervisor, **Industrial Engineering, METU** _____

Prof. Dr. Faruk Polat
Co-supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Yaşar Yasemin Serin
Industrial Engineering, METU _____

Prof. Dr. Cem İyigün
Industrial Engineering, METU _____

Prof. Dr. Faruk Polat
Computer Engineering, METU _____

Assoc. Prof. Dr. Seçil Savaşaneril
Industrial Engineering, METU _____

Assoc. Prof. Dr. Mehmet Tan
Computer Engineering, TOBB ETU _____

Date: 21.07.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Melis İlayda Bal

Signature :

ABSTRACT

REWARD SHAPING FOR EFFICIENT EXPLORATION AND ACCELERATION OF LEARNING IN REINFORCEMENT LEARNING

Bal, Melis İlayda
M.S., Department of Operational Research
Supervisor: Prof. Dr. Cem İyigün
Co-Supervisor: Prof. Dr. Faruk Polat

July 2022, 121 pages

In a Reinforcement Learning task, a learning agent needs to extract useful information about its uncertain environment in an efficient way during the interaction process to successfully complete the task. Through strategic exploration, the agent acquires sufficient information to adjust its behavior to act intelligently as it interacts with the environment. Therefore, efficient exploration plays a key role in the learning efficiency of Reinforcement Learning tasks.

Due to the delayed-feedback nature of Reinforcement Learning settings with sparse explicit reward structure, the required time for learning becomes the main cause of learning inefficiency. This problem is exacerbated particularly in complex tasks with large state and action spaces. Decomposing the task or modifying the reward structure to provide frequent feedback to the agent has been shown to accelerate learning.

This thesis proposes two methods with a reward shaping mechanism to address the aforementioned problems. To attack the efficient exploration problem, a framework called *population-based repulsive reward shaping mechanism using eligibility traces*

is proposed under the scope of tabular RL representation. The computational study on benchmark problem domains showed that efficient exploration is achieved with a significant improvement in learning and state-space coverage with the proposed framework. Furthermore, to accelerate learning, the thesis also proposes an approach called *potential-based reward shaping using state-space segmentation with the extended segmented Q-Cut algorithm*. Experimental results on sparse-reward benchmark domains showed that the proposed method indeed speeds up learning of the agent without sacrificing computation time.

Keywords: reinforcement learning, coordinated exploration, eligibility traces, potential-based reward shaping, state-space segmentation

ÖZ

PEKİŞTİRMELİ ÖĞRENMEDE VERİMLİ KEŞİF VE HIZLI ÖĞRENME İÇİN ÖDÜL ŞEKİLLENDİRME

Bal, Melis İlayda
Yüksek Lisans, Yöneylem Araştırması Bölümü
Tez Yöneticisi: Prof. Dr. Cem İyigün
Ortak Tez Yöneticisi: Prof. Dr. Faruk Polat

Temmuz 2022 , 121 sayfa

Bir Pekiştirmeli Öğrenme görevinde, öğrenen etmenin, görevi başarıyla tamamlamak için etkileşim süreci sırasında belirsiz çevresi hakkında yararlı bilgileri verimli bir şekilde çıkarması gerekir. Etmen, stratejik keşif sayesinde çevresi hakkında yeterli bilgiyi elde eder ve bu bilgiyi çevresi ile iletişime girerken akıllıca hareket etmek için davranışlarını ayarlamada kullanır. Bu nedenle, verimli keşif, Pekiştirmeli Öğrenme görevlerinin öğrenme verimliliğinde kilit bir rol oynar.

Seyrek ödül yapısına sahip Pekiştirmeli Öğrenme ortamlarının gecikmeli geri bildirim doğasına sahip olması nedeniyle öğrenme için gereken zaman, öğrenme verimsizliğinin ana nedeni haline gelir. Bu sorun, özellikle büyük durum ve eylem uzaylarına sahip karmaşık görevlerde daha da şiddetlenir. Görevi ayrıştırmanın veya etmene sık geri bildirim sağlamak için ödül yapısını değiştirmenin öğrenmeyi hızlandırdığı gösterilmiştir.

Bu tez, yukarıda bahsedilen sorunları ele almak için ödül şekillendirme mekanizma-

sına sahip iki yöntem önermektedir. Verimli keşif problemini ele almak için, tablosal Pekiştirmeli Öğrenme gösterimi kapsamında *niteliklilik izlerini kullanan popülasyona dayalı itici ödül şekillendirme mekanizması* adlı bir yapı önerilmiştir. Deney sonuçları, önerilen yapı kullanıldığında öğrenme ve durum uzayı keşfindeki iyileşmelerle birlikte verimli keşif elde edildiğini göstermiştir. Ayrıca, bu tez, öğrenmeyi hızlandırmak için *bölümlenmiş Q-Cut algoritmasının genişletilmiş versiyonu ile durum-uzayı segmentasyonu kullanarak potansiyele dayalı ödül şekillendirme* adlı bir yaklaşım önermektedir. Seyrek ödül yapısına sahip problemlerdeki deneysel sonuçları, önerilen yöntemin, hesaplama zamanından ödün vermeden etmenin öğrenmesini hızlandırdığını göstermiştir.

Anahtar Kelimeler: pekiştirmeli öğrenme, koordineli keşif, niteliklilik izleri, potansiyele dayalı ödül şekillendirme, durum uzayı segmentasyonu

To my family, for their infinite love, encouragement, and faith in me

ACKNOWLEDGMENTS

I would like to express my endless gratitude to my venerable advisor Prof. Cem İyigün. It was a great privilege for me to work with such a great teammate. His creative insight, open-mindedness, enthusiasm for learning out-of-comfort-zone topics, unlimited support and guidance always kept me motivated throughout this study.

I also would like to give my sincere gratitude to my dear co-supervisor Prof. Faruk Polat. Without him, I would not be able to think of working on reinforcement learning. He introduced me my greatest interest, and guided me in the field with his wisdom and valuable experiences.

I would like the thank the examining committee members of my thesis, Prof. Yasemin Serin, Assoc. Prof. Dr. Seçil Savaşaneril, and Assoc. Prof. Dr. Mehmet Tan for their helpful suggestions on my thesis study.

I would like to express my special thanks to soon-to-be Dr. Hüseyin Aydın for helping me patiently whenever I need. Our useful discussions and his encouraging feedbacks were so valuable for my study.

I would like to thank the members of the RL research group, for listening my progress, especially Assist. Prof. Dr. Alper Demir for his spot-on comments about my work.

Finally, I cannot thank enough my dearest family, Birnur, Muzaffer, Başak, and our little member Zeytin for their infinite love and encouragement. There was never a moment when I did not feel their support and understanding. They are the joy, energy, and foundation of my life.

This study is supported by 2210-A program of the Scientific and Technological Research Council of Turkey under Grant No. 1649B022001290.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ALGORITHMS

ALGORITHMS

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
DM	Decision Maker
ESegQ-Cut	Extended Segmented Q-Cut
ML	Machine Learning
MC	Monte Carlo
MDP	Markov Decision Process
PBRS	Potential-based Reward Shaping
RL	Reinforcement Learning
RRS	Repulsive-Reward-Shaper
RS	Reward Shaping
SegQ-Cut	Segmented Q-Cut
TD	Temporal Difference
ToH	Tower of Hanoi

CHAPTER 1

INTRODUCTION

Reinforcement learning (RL) [39] is a feedback-based learning paradigm under the core concept of Artificial Intelligence (AI) research, called Machine Learning (ML). Reinforcement learning builds upon learning through interaction with the environment when no prior knowledge regarding the task is provided. During the interaction process, the RL agent must act in the uncertain environment by taking actions and observing the consequences to deduce informative data that might be beneficial for accomplishing the learning task effectively. To this end, exploration of the environment plays a key role since acquiring sufficient information on the environment diminishes the stochasticity involved so that the goal-directed agent can master the task readily. On the other hand, the agent needs to balance the exploration with the exploitation to maximize its total reward in the long run. Through exploration the agent sacrifices from short-run gains to discover actions that may yield higher rewards and worth to exploit. Therefore, exploring the environment efficiently results in a significant improvement in accelerating the learning performance and sample efficiency of RL tasks. In particular, tasks with sparse rewards, which is one of the fundamental challenges of RL, are demonstrated to be solved more successfully with mechanisms that encourage efficient exploration [4].

Existing studies consider this major RL problem by introducing random action selection techniques [8, 11, 41] that mostly emerge from ε-greedy exploration rule [43], or assigning action selection probabilities using the learned policies, estimated value functions or rewards [43, 7, 3, 44]. Various approaches provide exploration bonuses [37, 12] to the agent which are computed based on state-action visitation counts [38, 4, 21], initial value estimates [39, 6] or a prediction of the environment model

[34, 33, 2]. There are also intrinsic-motivation based exploration strategies [4, 30] which encourage visitation to novel states and very few studies [19, 20, 42, 23] which focus on exploration with coordination.

Aside from efficient exploration problem, RL methods also suffer from slow learning due to delayed feedback in sparse-reward environments and large state and action spaces in real-world tasks. To accelerate the learning efficiency in this type of learning, common approaches in the RL literature center around task decomposition or reward-based methods to make reward function denser. Studies that aim to speed up the learning with task decomposition, divide the complex RL problem into simpler sub-problems by identifying relatively important states, labeling them as *bottlenecks* or *subgoals* [26] and then learning macro-policies generated with the options framework [40] to reach the identified states. On the other hand, reward-based methods mostly focus on reward shaping by giving additional rewards to the agent to alleviate the delayed-feedback nature of the problem and improve the learning speed [29, 13].

In this thesis, we attack the efficient exploration problem by proposing a framework that serves as a *population-based repulsive reward shaping mechanism using eligibility traces* to enhance the exploration of the state-space under the scope of tabular RL representation. The framework contains a hierarchical structure of RL agents, where a higher level Repulsive-Reward-Shaper (*RRS*) *agent* coordinates the exploration of its population of sub-agents through repulsion when necessary conditions on their eligibility traces are met. The framework not only encourages the exploration of diverse regions of the environment via coordination but also implicitly unifies the eligibility trace information collected from a population of agents. Our experiments on benchmark problem domains showed that the framework indeed accelerates the learning performance of the agent and improves the coverage of the state-space. The positive impact of the proposed framework is especially observed in the domains with sparse explicit reward structure.

Furthermore, the thesis also proposes *potential-based reward shaping using state-space segmentation with extended segmented Q-cut algorithm* approach to address the problem of learning efficiency. The method is inspired by the idea of Segmented Q-Cut algorithm introduced by [26] that handles sub-goal identification problem in

RL. However, our perspective is significantly different compared to the study [26], because we aim to accelerate the learning by providing feedback to the agent with the potential-based reward shaping mechanism that depends on a state-space segmentation rather than generating macro-policies based on sub-goal identification. The experimental results in benchmark sparse-reward environments showed that our proposed method accelerates the learning speed notably without having a need to sacrifice from computation time.

1.1 Contribution of the Thesis

This thesis contributes to the RL literature with two novel reward shaping-based approaches called *population-based repulsive reward shaping mechanism using eligibility traces* and *potential-based reward shaping using state-space segmentation with the extended segmented Q-cut algorithm*.

- The first proposed reward shaping framework achieves coordinated exploration using eligibility trace information of a population of agents. Compared to the existing methods in the RL literature that perform coordinated exploration, our method is counted as a novel one since eligibility trace information has not been employed in reward shaping to achieve coordination. Beyond coordination mechanism, the method provides an informative initialization for state-action values in single-agent RL settings. Thus, it significantly enhances the learning speed of the agent. Furthermore, the proposed approach unifies the eligibility trace information of multiple agents. Through this, it provides a unique way to benefit from such valuable information collected from multiple agents. Lastly, it notably improves the learning performance and state-space exploration compared to the famous benchmark RL methods in well-known problem domains.

- The second proposed method presents a way to segment the state-space using the transition history of the RL agent. Furthermore, it also provides a way to benefit from extracted segment information in the agent's learning process through a potential-based reward shaping mechanism. The method introduces

an extension of the segmented q-cut algorithm [26], however, it does not require to generate additional macro-policies via options framework and learn options to enhance the agent's learning. Because the approach solely depends on the key ingredient of the learning, rewards. The method basically modifies the reward function using segment information on the state-space rather than identified sub-goals. Hence, it eliminates the need of generating options. Finally, our proposed method accelerates the learning performance compared to Q-Learning algorithm in domains with sparse explicit reward structure. The method also outperforms Q-Learning regarding computation time.

The proposed methods in this thesis unite under the reward shaping perspective. The first proposed method employs reward shaping for efficient exploration problem whilst the second one is for acceleration in learning. Therefore, the structure of the thesis includes two main chapters corresponding to each proposed method. The outline of this thesis is as follows. Chapter 1 introduces the problems that the thesis focuses on and summarizes the existing methods. Chapter 2 provides the necessary background information. Chapter 3 discusses related works in the RL literature. Chapter 4 introduces the first proposed method called *population-based repulsive reward shaping mechanism using eligibility traces*. Moreover, the experimental results and discussions are also given in this chapter. Then, the thesis addresses the problem of learning efficiency and presents the second proposed approach, *potential-based reward shaping using state-space segmentation with extended segmented Q-cut algorithm* along with the computational outcomes and discussions in the Chapter 5. Finally, Chapter 6 concludes the work provided in this thesis and suggests possible future research directions.

CHAPTER 2

BACKGROUND

This chapter provides the required background for the addressed problems and the proposed approaches in this thesis. It gives the information on mathematical background for RL through Markov Decision Processes and summarizes the relevant value-based methods. The chapter then presents the eligibility trace and reward shaping mechanisms. Furthermore, it gives the necessary background that covers graph theory basics and maximum-flow/minimum-cut problem within the context of RL.

2.1 Markov Decision Processes

Sequential decision making problems where earlier decisions influence the later situations and actions are classically formalized by *Markov Decision Processes (MDPs)* [31]. Reinforcement Learning tasks that are represented as a sequential decision making processes under uncertainty where earlier decisions have undetermined consequences are modelled with MDPs.

A Markov Decision Process (MDP) is a stochastic control process defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$ consisting of

- a finite set of *states* (state-space) denoted as $\mathcal{S}$,
- a finite set of *actions* (action-space) denoted as $\mathcal{A}$,
- a *transition function* $\mathcal{T}$ which maps the state-action pairs to a probability distribution over states, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ where $\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') = 1, \forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}$. $\mathcal{T}(s, a, s')$ shows the probability of the transition to state s' after taking action a in state s.

- a *reward function* $\mathcal{R}$ which provides immediate reward after an action choice in some state $\in \mathcal{S}$, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. $\mathcal{R}(s, a)$ shows the expected immediate reward signal given by the environment after taking action a in state s.
- a discount factor $\gamma \in [0, 1]$ that determines the importance of future rewards to the current state.

A state $s \in \mathcal{S}$ shows the whole description of the state of the world which the decision maker is interacting with. An *observation* o, on the other hand, may capture only the partial description of the state. Hence, some of the information about the world might be missing in the observation. Depending on the decision maker's ability to observe the complete state of the world, an MDP can be fully observed or partially observed. If the DM can only observe the partial information on the state of its world, then the MDP is regarded as partially observed.

The action-space $\mathcal{A}$ describes all admissible actions in the world that the decision maker can choose. The action-space can be *discrete* if the number of available moves to the decision maker is finite, or *continuous* if the moves are defined in terms of real-values.

As shown with the transition function $\mathcal{T}(s, a, s')$, the probability of moving to the new state depends only on the current state of the process and action choice of the decision maker. That is, given current state $s_t = s$ and action $a_t = a$ at decision epoch t, the probability of transitioning to next state $s_{t+1} = s'$ is conditionally independent of all previous state and action choices as indicated below.

$$P\left(s_{t+1} = s' \mid s_t, \ldots, s_0, a_t, \ldots, a_0\right) = P\left(s_{t+1} = s' \mid s_t, a_t\right) = P(s, a, s') \tag{2.1}$$

The property (2.1) is called as the *Markov property* and satisfied by the MDPs. Based on this, the reward distribution also depends only on the current state of the process and action choice of the decision maker. Therefore, expected reward at decision epoch t satisfies the following.

$$\mathbb{E}\left[r_{t+1} \mid s_t, \ldots, s_0, a_t, \ldots, a_0\right) = \mathbb{E}\left[r_{t+1} \mid s_t = s, a_t = a\right] = \mathcal{R}(s, a) \tag{2.2}$$

At each decision epoch $t \in T$ where T can be finite or infinite, the decision maker observes its state $s \in \mathcal{S}$ and chooses an admissible action $a \in \mathcal{A}$ in state s. The process

transitions to a new state $s' \in \mathcal{S}$ according to the transition function $\mathcal{T}(s, a, s')$. Depending on the transition function structure, an MDP can be deterministic or stochastic (non-deterministic). In the deterministic case, the process always moves to the same state when the same action is chosen. Otherwise, different outcomes can be observed with the same action selections. After the transition, the decision maker receives an immediate reward $r_t(s, a) \in \mathbb{R}$ based on the reward function $\mathcal{R}(s, a)$.

A *policy* $\pi, \pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is defined as a probability distribution over the state-action pairs. $\pi_t(s, a) = P(a_t = a \mid s_t = s)$ denotes the probability of choosing the action a in state s at decision epoch t. In this definition, π_t shows a *stochastic/randomized* policy. However, a policy π can also be *deterministic*, if π is defined as $\pi : \mathcal{S} \to \mathcal{A}$. In this case, $\pi_t(s) = a$ denotes the unique action choice a in state s at decision epoch t. If $\pi_t = \pi, \forall t \in T$, then the policy is called *stationary*.

The goal of the decision maker is to extract the *optimal policy* π^* that yields the optimal action selections $a_t^* \sim \pi^*(. \mid s_t)$ for each decision epoch t to maximize its expected discounted *return* or utility. Return (G_t) shows the sum of the discounted future rewards discounted by a factor $\gamma \in (0, 1)$ and defined as

$$G_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k = r_{t+1} + \gamma G_{t+1}, \tag{2.3}$$

for each decision epoch $t \in T$. As shown in (2.3), G_t can be written recursively using the return of the subsequent decision epoch.

To maximize the expected return, it is conventional to search and evaluate the policies in the policy space. For this, *value functions* are helpful representatives of the value of policies in terms of expected returns for every possible state. The state-value function of a policy π is a mapping from states to real-values $V^\pi : \mathcal{S} \to \mathbb{R}$. The value $V^\pi(s)$ of a state $s \in \mathcal{S}$ under a policy π defines the expected return starting from state s and following policy π,

$$V^\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid s_t = s \right]. \tag{2.4}$$

The value of a state can be written in a recursive form using the values of possible successive states. The state-value Bellman equation shows this recursive relationship

by

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma G_{t+1} \mid s_t = s\right] \\ &= \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} P\left(s,a,s'\right)\left[\mathcal{R}(s,a) + \gamma V^{\pi}\left(s'\right)\right]. \end{aligned} \tag{2.5}$$

The underlying optimization problem that the decision maker tries to solve is

$$\max_{\pi} V^{\pi}(s). \tag{2.6}$$

Hence, the optimal value function V^* gives the maximum value that is achieved in any state,

$$V^*(s) = \max_{\pi} V^{\pi}(s). \tag{2.7}$$

The optimal value function V^* satisfies the Bellman optimality equation for state values

$$V^*(s) = \max_{a \in A} \mathbb{E}_{s'}[\mathcal{R}\left(s,a\right) + \gamma V^*(s')]. \tag{2.8}$$

[5] showed that there exists a unique optimal value function in a finite MDP. Moreover, for the infinite horizon discounted MDP settings, Theorem 1 by [31] guarantees the optimal policy under the assumption of finite state-space, finite action-space and bounded rewards.

Theorem 1 *For any infinite horizon discounted MDP, there always exists a deterministic stationary policy π that is optimal [31].*

The action-value $Q^{\pi}(s,a)$ of state-action pair s, a under policy π, $Q^{\pi} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, defines the expected return starting from state s, taking an arbitrary action a and then following policy π,

$$Q^{\pi}(s,a) \doteq \mathbb{E}_{\pi}\left[G_t \mid s_t = s, a_t = a\right]. \tag{2.9}$$

The action-value function can also be written in a recursive form as shown in the state-action Bellman equation by

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma G_{t+1} \mid s_t = s, a_t = a\right] \\ &= \sum_{s' \in \mathcal{S}} P\left(s,a,s'\right)\left[\mathcal{R}(s,a) + \gamma \sum_{a' \in \mathcal{A}} \pi(s',a') Q^{\pi}(s',a')\right]. \end{aligned} \tag{2.10}$$

The optimal action-value function Q^* gives the maximum expected value that is achieved in any state-action pair,

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a). \tag{2.11}$$

Q^* satisfies the Bellman optimality equation for state-action values

$$Q^*(s,a) = \mathbb{E}_{s'}\left[\mathcal{R}\left(s,a\right) + \gamma \max_{a' \in A} Q^*\left(s',a'\right)\right]. \tag{2.12}$$

If Q^* is known for each state-action pair, then the optimal policy can be extracted from optimal actions $a^*(s)$ for any state $s \in \mathcal{S}$ as

$$a^*(s) = \arg\max_{a} Q^*(s,a). \tag{2.13}$$

In well-known RL strategies, estimating the value functions or action-value functions is a common step towards discovering the optimal policy.

2.2 Reinforcement Learning

Reinforcement learning builds upon learning through interaction with the environment to achieve a goal when no prior knowledge regarding the task is provided [39]. The key ingredients of RL are the *agent* i.e. the learner or the decision maker as described in the Section 2.1, the uncertain *environment* (world) which the agent interacts with, the *reward signal* which is the feedback provided to the agent to evaluate its movements and a *value function* that helps to represent the goal of the agent. During the interaction process, the RL agent must act in the uncertain environment by taking actions in each time *step* (decision epoch) and observing the consequences in the form of successive states and rewards to deduce informative data that might be beneficial for accomplishing the learning task effectively.

The interaction process illustrated in the Figure 2.1, consists of the RL agent observing the state of the environment and selecting an admissible action, followed by receiving a feedback from the environment as a reward or punishment and environment changing to a new state and so on. The feedback of the environment (*reward*

signal) shows the agent how good or bad its action choice was depending on the environment state. If this interaction process continues until the agent reaches its goal or terminal condition, then it is called as an *episodic* task and an *episode* is defined as the complete sequence of agent-environment interactions from initial to ending state. On the other hand, if the interaction process does not include a terminal state or condition, then the task never ends and regarded as *continuous* or *non-episodic*.

The learning is based on collected experiences during the training instead of an explicit supervision or using a static data set which include labels indicating the best actions to take. Trajectories are regarded as the main source of data for the agent's learning. A trajectory τ shows the complete sequence of states and actions in an episode, $\tau = (s_0, a_0, s_1, a_1, \dots)$. Moreover, the feedback provided to the agent during the interaction is often delayed in most of the RL tasks. That means, the impact of an action choice can only be observed in the reward signal many steps later instead of instantaneous reaction. This repeated trial-and-error fashioned learning combined with a goal-directed learner in a dynamic environment distinguishes RL from other machine learning paradigms.

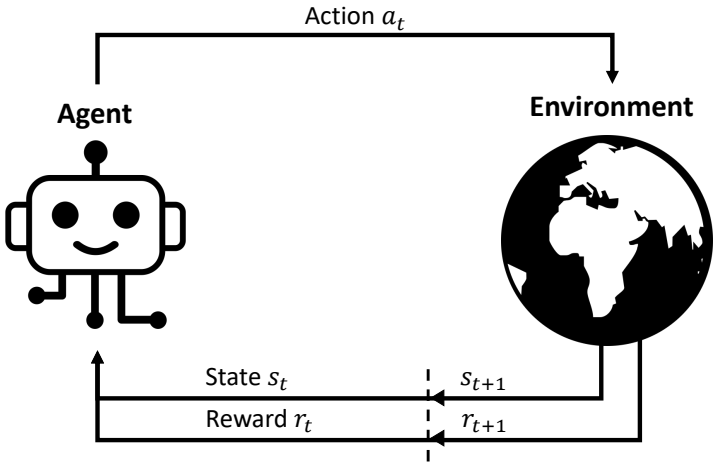


Figure 2.1: The agent-environment interaction loop.

The *goal* of the RL agent is to extract the optimal policy which gives the optimal sequential decisions that maximizes its long term total rewards by utilizing the feedback it receives in the absence of the environment *model*. That means, the agent has no former information on the transition and reward functions $\langle \mathcal{T}, \mathcal{R} \rangle$ of the underlying MDP. The long term total rewards of the agent is called *return* and described as

the future discounted cumulative reward that the agent receives.

A majority of model-free RL algorithms in which the agent does not extract the model of the environment, estimating value functions or Q-functions has significant role to learn the optimal policies. Originated from this idea, *value-based* methods constitute an essential portion of the RL literature. The following section summarizes the tabular versions of the value-based approaches which mainly work for small scale MDPs.

2.3 Value-based Methods

The basic steps of value-based RL methods are to first estimate the value functions and then derive optimal policies using the estimated values. A trivial way to learn these value functions is with Monte Carlo (MC) methods. MC methods provide an unbiased estimation of state-values or Q-values and optimal policies through averaging the complete returns of sampled experiences.

In an episodic task, for a sampled episode $(s_t, a_t, r_t, s_{t+1}), t \in T$, MC updates the state-action values with the rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [G_t - Q(s_t, a_t)], \tag{2.14}$$

where the actual return following time step t, G_t is defined by (2.3) for each t in the episode and $\alpha \in (0, 1)$ denotes the learning rate. Contrary to their simplicity, MC methods suffer from high variance as the variance of the sampled returns can be high, and slow convergence since the value update is performed only after the completion of an episode, which makes the learning process *offline*. On the other hand, learning in an *online* (step-by-step) sense introduced with Temporal Difference methods speeds up the convergence and is explained in detail in the following section.

2.3.1 Temporal Difference Learning

Temporal-difference (TD) learning is the central idea of the most RL algorithms that stands on incremental computation of value estimations. TD approaches differ from MC methods as they bootstrap i.e. they update the value estimates based on the

learned value estimates of successor state-action pairs with the update rule

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha\delta_t. \tag{2.15}$$

where δ_t is the one-step TD error for state-action value after transition to next state s_{t+1} and receiving r_{t+1} which is computed as

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t). \tag{2.16}$$

(2.15) is applied in Sarsa [32] algorithm which is a well-known on-policy TD control approach extends from standard one-step TD learning, or *TD(0)*. The off-policy version of the error computation introduced in (2.17) is used in the Q-Learning [43] algorithm.

$$\delta_t = r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t). \tag{2.17}$$

An extension of one-step TD that employs multi-step bootstrapping is n-step TD learning, *TD(n)*, in which the bootstrapping considers n-step successor state-action pairs as

$$Q_{t+n}(s_t, a_t) \leftarrow Q_{t+n-1}(s_t, a_t) + \alpha\delta_{t:t+n} \tag{2.18}$$

where $\delta_{t:t+n}$ is the n-step TD error computed by

$$\delta_{t:t+n} = G_{t:t+n} - Q_{t+n-1}(s_t, a_t) + \alpha\delta_{t:t+n} \tag{2.19}$$

where $G_{t:t+n}$ denotes the n-step return defined as

$$G_{t:t+n} \doteq r_{t+1} + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n Q_{t+n-1}(s_{t+n}, a_{t+n}). \tag{2.20}$$

TD methods are faster due to online learning, but less stable because of their sensitivity to initial estimates.

2.3.2 Q-Learning

Q-Learning is the prominent off-policy model-free TD control algorithm that finds the optimal action-value function by learning the Q-values in an online manner [43, 39]. Q-Learning does not require the model of the environment and is independent of the policy being followed. Therefore, most of the RL methods inspired by the Q-Learning due to its characteristics and simplicity. The pseudocode of the Q-Learning is given in the Algorithm .

Algorithm: Q-Learning

Input : $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$

Learning rate $\alpha \in [0, 1]$

Exploration rate $\varepsilon \in [0, 1]$

Discount factor $\gamma \in [0, 1]$

Number of episodes $M \geq 1$

Output: Q

Initialization:

Initialize $Q(s, a)$ arbitrarily $\forall s \in \mathcal{S}, a \in \mathcal{A}$

$episode = 0$

for $episode = 0$ **to** M **do**

  Initialize s

  while s *is not terminal* **do**

    Choose $a \leftarrow$ `EPSILON-GREEDY`(Q, ε)

    Take action a, observe r, s'

    $\delta \leftarrow r + \gamma \max_{a'} Q(s', a') - Q(s, a)$

    Update $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$

    $s \leftarrow s'$

  end while

end for

return Q

2.3.3 Sarsa

Sarsa [32] algorithm that is named based on the agent's experience tuple $\langle s, a, r, s', a' \rangle$ is the on-policy model-free TD control algorithm for state-action values. Compared to the Q-Learning, Sarsa uses the same policy both in action-selection and Q-value update stages. The pseudocode of the Sarsa is provided in the Algorithm .

Algorithm: Sarsa

Input : $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$

Learning rate $\alpha \in [0, 1]$

Exploration rate $\varepsilon \in [0, 1]$

Discount factor $\gamma \in [0, 1]$

Number of episodes $M \geq 1$

Output: Q

Initialization:

Initialize $Q(s, a)$ arbitrarily $\forall s \in \mathcal{S}, a \in \mathcal{A}$

$episode = 0$

for $episode = 0$ **to** M **do**

  Initialize s, a

  while s *is not terminal* **do**

    Take action a, observe r, s'

    Choose $a' \leftarrow$ `EPSILON-GREEDY`(Q, ε)

    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$

    Update $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$

    $s \leftarrow s', a \leftarrow a'$

  end while

end for

return Q

2.4 Eligibility Traces

Eligibility trace mechanism (λ) [36] builds a bridge between MC methods and TD learning to approach temporal credit assignment problem more efficiently. Eligibility traces generalize TD methods to *TD(λ)* and allow us to perform bootstrapping over multiple time intervals concurrently while behaving as a short term memory for the agent. Furthermore, the mechanism achieves computational efficiency compared to the n-step TD methods since storing only a single trace vector for each state $e(s)$ or for each state-action pair $e(s, a)$ is required. Eligibility trace associated with a

state-action pair (s, a) at time t denoted by $e_t(s, a)$, is defined by

$$e_{t+1}(s,a) = \begin{cases} 1, & \text{if } s = s_t \text{ and } a = a_t. \\ \gamma\lambda e_t(s,a) & \text{otherwise.} \end{cases} \tag{2.21}$$

$e_t(s, a)$ is called a *replacing* eligibility trace [35] that decays by the discount rate γ and trace-decay parameter $\lambda \in [0, 1]$ at each time step. There is also another version of eligibility trace called *accumulating* eligibility trace in which the trace value is incremented by 1 each time the pair (s, a) is visited and then fades away by $\gamma\lambda$ after the Q-value update.

2.5 Sarsa(λ) Algorithm

The extension of Sarsa algorithm with eligibility traces called Sarsa(λ) [39], implements a compound update by

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha\delta_t e_t(s,a), \quad \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{2.22}$$

where δ_t is the one-step TD error for the action values and $e_t(s, a)$ is defined by (2.21) for all s, a. Through the eligibility of state-action pairs, the temporal difference after a transition is propagated back to all (s, a) pairs recently visited by the agent. Algorithm provides the complete pseudocode for Sarsa(λ).

2.6 Reward Shaping

In sparse and/or delayed reward environments, learning becomes challenging since the RL agent receives no reinforcement to update its behavior and knowledge on the environment. To guide the agent in such settings, additional reward signals are provided through reward shaping mechanisms [25]. These additional rewards can be designed in the form of bonuses [12], belief-based signals [24] or intrinsic motivation-based signals depending on state novelty [34] or curiosity [30]. In the most general form of reward shaping, original MDP M defined as $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$ is transformed into M', $M' = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}', \gamma \rangle$ where $\mathcal{R}' = \mathcal{R} + F$ is the transformed reward function and $F : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ denotes the shaping reward function [28, 22]. With the

Algorithm: Sarsa(λ)

Input : $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$

Learning rate $\alpha \in [0, 1]$

Exploration rate $\varepsilon \in [0, 1]$

Discount factor $\gamma \in [0, 1]$

Number of episodes $M \geq 1$

Trace decay rate $\lambda \in [0, 1]$

Output: Q

Initialization:

Initialize $Q(s, a)$ arbitrarily, $\forall s \in \mathcal{S}, a \in \mathcal{A}$

Initialize $e(s, a) = 0, \forall s \in \mathcal{S}, a \in \mathcal{A}$

$episode = 0$

for $episode = 0$ **to** M **do**

  Initialize s, a

  while s *is not terminal* **do**

    Take action a, observe r, s'

    Choose $a' \leftarrow$ `EPSILON-GREEDY`(Q, ε)

    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$

    $e(s, a) = 1$; `/* replacing trace */`

    for *all* $s \in \mathcal{S}, a \in \mathcal{A}$ **do**

      $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$

      $e(s, a) \leftarrow \gamma\lambda e(s, a)$

    end for

    $s \leftarrow s', a \leftarrow a'$

  end while

end for

return Q

addition of shaping reward function, the agent is stimulated to improve its knowledge on the environment.

In particular, the bonus form of reward shaper is commonly used to encourage agent to explore novel states, hence achieve better exploration. The bonus term $\mathcal{B}_t(s, a)$ introduced by [37], is provided to the agent after receiving extrinsic (environmental) reward signal $r_t(s, a)$ for visiting state-action pair (s, a) at time step t. Thus, the final reward value [2] at time step t, $\tilde{r}_t(s, a)$, is defined as

$$\tilde{r}_t(s, a) = r_t(s, a) \oplus \mathcal{B}_t(s, a) \tag{2.23}$$

where $\oplus$ denotes the aggregation of extrinsic reward with the bonus term.

2.6.1 Potential-based Reward Shaping

Potential-based reward shaping (PBRS) is a way to shape rewards to deal with sparse reward function while preserving policy invariance [28]. To speed up the learning in sparse-reward RL tasks, additional rewards in the form of potentials are provided to the agent without changing the optimal policy for the task.

In PBRS, shaping reward function denoted with $F, F : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$ is defined as the difference between real-valued potential functions of the successive states for a state transition. The *potential* of a state represented by the function $\Phi, \Phi : \mathcal{S} \rightarrow \mathbb{R}$ expresses some knowledge on the environment as a real-value given state information. As an example, potential functions can be subgoal-based [29], distance-based [45], plan-based [16] or auxiliary reward functions-based [17].

Formally, the potential-based shaping function $F(s, s')$ for a state transition $s \rightarrow s'$ is defined by

$$F(s, s') = \gamma\Phi(s') - \Phi(s), \tag{2.24}$$

where γ is the discount factor.

[28] proves that if F is a potential-based reward shaping function as defined in the Equation (2.24), then every optimal policy for the modified MDP M' will be an optimal policy in the original MDP M and vice versa. Hence, defining F of the form given in (2.24) is sufficient to guarantee the policy invariance.

2.7 Graph Theory Basics

Our second proposed method in Chaper 5 is a graph-theoretic approach based on the idea of Segmented Q-Cut algorithm [26] which is basically to find the subgoal (bottleneck) states, transitioning the agent's interactions in the environment to a graph structure and modeling the problem as a maximum-flow/minimum-cut problem. Therefore, a brief review on graph theory within the context of RL is presented in this section.

A *graph* G is a capacitated directed network $G = \langle N, A \rangle$ defined over a set of nodes N and a set of arcs A with a non-negative capacity c_{ij} associated with each arc $(i, j) \in A$.

A *subgraph* $G_{sub} = \langle N_{sub}, A_{sub} \rangle$ is a graph whose nodes and arcs are subsets of node and arc sets of G. G_{sub} is an *induced* graph of G if it is induced by the node set N_{sub} where $N_{sub} \subseteq N$ and contains all the arcs of the nodes N_{sub} from G.

The *degree* of a node i denotes the number of arcs adjacent to node i. In a directed graph G, the degree of a node is expressed as indegree and outdegree measures. The *indegree* of a node i where $i \in N$, is the number of arcs that are coming into the node i, and the *outdegree* of a node i shows the number of arcs that are going out from the node i.

A *connected* graph is a graph in which we can find a path between every pair of nodes in the graph. Otherwise, the graph is called *disconnected*. A directed graph is said to be *weakly connected* if we can obtain a connected graph when we replace all of the directed arcs with undirected ones, and *strongly connected* if we can find a directed path between each pair of nodes in the graph in both directions.

A *connected component* of an undirected graph is a subgraph in which we can find a path between each pair of nodes. *Weakly connected component* of a directed graph is a maximal subgraph that is unreachable from other nodes in the graph and in which we can find an undirected path between each pair of nodes. On the other hand, *strongly connected component* of a directed graph is a maximal subgraph that is unreachable from other nodes in the graph and in which we can find directed path between every

pair of nodes in both directions.

Weakly Connected Components (WCC) a.k.a Union Find algorithm introduced by [14], finds the distinct set of weakly connected components of the directed graph. The pseudocode for Union Find is given in the Algorithm .

Algorithm: Union Find (WCC)

Input : A directed graph G

Output: $components$

Initialization: $visited\_nodes = \emptyset, components = \emptyset, connected\_nodes = \emptyset$
for *each node* n *in the graph* G **do**
  if n *is not in* $visited\_nodes$ **then**
    $connected\_nodes = BFS(n)$; `/*breadth-first search*/`
    Append $connected\_nodes$ to $visited\_nodes$
    Append $connected\_nodes$ to $components$
  end if
end for
return $components$

Let G be a connected graph. A *cut* set of graph G is the subset of arcs whose deletion from G makes G disconnected. An arc is called *cut arc* if is the element of the cut set. A *minimum cut* of graph G is the set of arcs whose removal from G divides G into two disjoint sets and minimal with respect to some measure. For instance, the cut can be minimal in terms of the capacity of the cut arcs, the sum of the weights of the cut arcs, or the number of cut arcs etc.

Ratio cut bi-partitioning metric measures the quality or significance of a cut. Let $s-t$ cut be a cut of connected graph G that separates the graph into two partitions denoted as s and t. The quality of $s-t$ cut is computed by

$$q_{cut}(s,t) = \frac{\mid N_s \mid\mid N_t \mid}{\mid A(N_s, N_t) \mid}, \tag{2.25}$$

where N_s and N_t are the set of nodes in the partitions s and t of graph G, respectively, and $\mid A(N_s, N_t) \mid$ is the number of arcs connecting both partitions. Based on this, a high-quality or significant cut is defined as the cut with small number of arcs

separating balanced partitions in the graph.

A *transition graph* in the RL context, is a special kind of graph structure that stores the state transitions in an MDP. Let G_t be a transition graph defined as $G_t = \langle N_t, A_t \rangle$. G_t is a capacitated directed network in which the nodes denote the states whereas the arcs denote the state transitions. To illustrate, the transition from state $s \rightarrow s'$ is reflected in the graph G_t with arc $(s, s') \in A_t$. The arc capacity definition depends on the focused RL task. Previous studies that aim to find subgoal states mostly use state visitation frequency-based arc capacities [26].

2.8 Maximum-Flow/Minimum-Cut Problem

Decomposing the RL tasks by finding subgoals (bottleneck states) to speed up the learning have been a major focus of the previous works in the RL literature. One way of finding subgoals in an RL setting is to express the subgoal-identification problem as a Maximum Flow-Minimum Cut problem [26].

A *flow network* $G_f = \langle N_f, A_f \rangle$ as shown in the Figure 2.2 is a special directed graph with capacitated arcs in which a flow goes through each arc of the network. The flow and the capacity of an arc $(i, j) \in A_f$ are denoted as f_{ij} and c_{ij}, respectively.

The Maximum Flow problem is the problem of finding maximum flow that can be sent from a single-source node $s \in N_f$ to single-sink node $t \in N_f$ in a flow network. The equivalent problem is called Minimum Cut problem that tries to find minimum capacity $s - t$ cut in the network. The capacity of the $s - t$ cut is defined as the sum of the capacity of the arcs in the cut set [1]. The Max-Flow Min-Cut theorem states that the maximum amount of flow that can be sent from s to t is equal to the capacity of the minimum $s - t$ cut of a flow network [9].

2.8.1 Preflow-Push Algorithm

Preflow-Push (push–relabel) is a class of algorithms to solve maximum flow problem. The fundamental idea of this type of algorithms is to maintain a *preflow* at each intermediate stage and progressively transform it into maximum flow by applying two

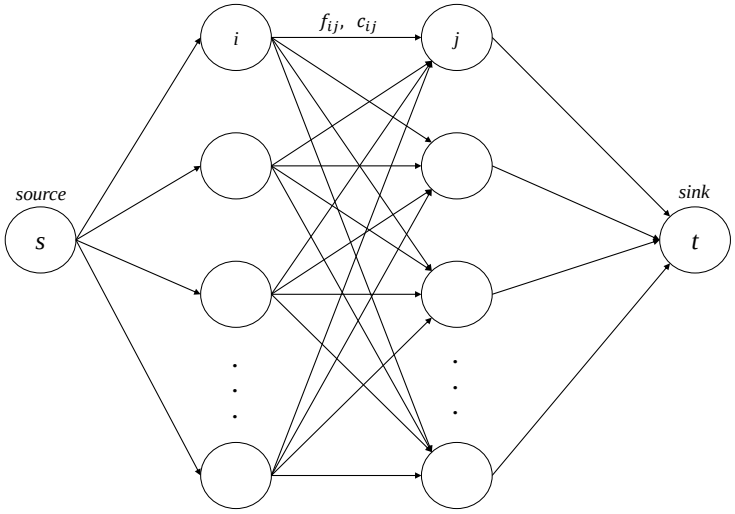


Figure 2.2: The flow network in Maximum-Flow/Minimum-Cut problem.

basic operations called *push* and *relabel*.

- A *preflow* $x, x : A_f \rightarrow \mathbb{R}$ is a function over arcs showing that for all nodes except the source s, net flow entering the node is non-negative while preflow of each arc does not exceed the arc capacities.

- Flow balance constraints may not be satisfied in the execution of these algorithms; hence, excess flow can occur. *Excess* of a node is defined as the difference of total flow coming into the node minus total flow going out of the node. If a node has a positive excess, then it is said as *active* node.

- In addition to excess value, each node is also associated with a label called *height* value. Height of a node is used to determine if the node can push flow or not.

- Depending on height and excess values of nodes, push operation is performed. By pushing flow from the nodes having excess flow with greater height values to the nodes with smaller height values, the flow proceeds on arcs from source to sink.

- Relabel operation is performed to change the height value of a node.

- The algorithm runs until no active node is present in the flow graph.

Preflow-push algorithms are shown to be more powerful and flexible than augmenting path algorithms [1]. A specific variant called highest-label push–relabel algorithm is regarded as benchmark with time complexity of $O(N^2\sqrt{A})$. Therefore, we benefit from this algorithm in our second proposed method introduced in Chapter 5.

2.9 Segmented Q-Cut Algorithm

To accelerate the Q-Learning algorithm [26] proposes automatic identification of subgoals (bottlenecks) in a dynamic environment with Segmented Q-Cut algorithm. Segmented Q-Cut is an extension of Q-Cut algorithm, also proposed by [26] in which the general idea is to create a directed graph of the transition history of the agent and use a Max-Flow/Min-Cut algorithm to find bottlenecks.

A *subgoal* or *bottleneck* is defined as the border state of strongly connected areas of the transition history graph of the agent. Transition graph as explained in the Section 2.7 holds the agent's experiences as state transitions in a directed and capacitated graph. The capacity of the arcs in the transition graph is defined with the *relative frequency* measure. Relative frequency c_{rf} is computed by

$$c_{rf} = \frac{n(i \rightarrow j)}{n(i)}, \tag{2.26}$$

where $n(i \rightarrow j)$ denotes the number of times transition from state i to state j is occurred and $n(i)$ shows the number of times state i is visited.

Q-Cut algorithm as given in the Algorithm chooses source and sink nodes if conditions for performing cut are met, and then applies a cut procedure described in the Algorithm . If the cut quality which is measured by ratio cut bi-partitioning metric is good, meaning that if it is greater than some pre-defined quality threshold, then the identified subgoal is used to create options to speed up learning process.

Segmented Q-Cut algorithm extends the idea of Q-Cut by using previously identified subgoals for state-space partitioning and then finding additional subgoals from generated partitions. The outline of the approach is given in the Algorithm . In this approach, cut procedure is not applied only once, but for each created partition as given in the Algorithm .

Algorithm: Q-Cut

while *True* **do**
    Interact with environment and learn using Macro- Q Learning
    Save state transition history
    If activating cut conditions are met, choose $s, t \in S$ perform CutProcedure (s, t)
end while

Algorithm: CutProcedure()

Input : source node s, sink node t

Translate state transition history to a graph representation
Find a Minimum Cut partition $[N_s, N_t]$ between nodes s and t
if *the cut's quality is "good"* **then**
    Learn the option for reaching new derived bottlenecks from every state in N_s, using Experience-Replay
end if

Algorithm: Segmented Q-Cut

Initialization:
    Create an empty segment N_0
    Include starting state s_0 in segment N_0
    Include starting state s_0 in $S(N_0)$
while *True* **do**
    Interact with environment/Learn using Macro-Q Learning
    Save state transition history
    for *each segment* N **do**
        if *activating cut conditions are met* **then**
            Perform CutProcedureForSegment(N)
        end if
    end for
end while

Algorithm: CutProcedureForSegment()

Input : Segment N

Extend segment N by connectivity testing

Translate state transition history of segment N to a graph representation

for *each* $s \in S(N)$ **do**

    Perform Min-Cut on the extended segment (s as source, choice of t is task depended) **if** *the cut's quality is good (bottlenecks are found)* **then**

        Separate the extended N into two segments N_s and N_t

        Learn the Option for reaching the bottlenecks from every state in N_s, using Experience Replay

        Save new bottlenecks in $S(N_t)$

    end if

end for

CHAPTER 3

RELATED WORK

In this chapter, we provide related studies on efficient exploration and acceleration of learning problems in the RL literature. To address the aforementioned RL problems, this thesis proposes two approaches in which reward shaping is a common aspect. Thus, related works having a reward shaping aspect are presented in this chapter. While Section 3.1 explains the studies on efficient exploration under the single-agent RL perspective, Section 3.2 summarizes works on exploration with coordinated agents. Section 3.3 discusses existing approaches in the literature which use eligibility trace mechanism and establishes a connection between coordinated exploration and eligibility traces. Finally, Section 3.4 explains works on potential-based reward shaping for improvement of learning efficiency.

3.1 Single-Agent Exploration

The exploration problem is addressed with numerous rules and strategies in the single-agent RL literature. Starting with pure randomization, previous works have evolved from basic exploration rules such as ε-greedy [43] a.k.a. *pseudo-stochastic* mechanism [8] and optimistic initialization [39] to more intelligent methods which make use of exploration-specific knowledge to direct the agent's exploration [2].

Recent studies mostly focus on this problem with *directed* exploration methods [2]. Randomized action selection approaches assign action selection probabilities depending on the estimated value functions/rewards or learned policies [41, 43, 7, 3, 44]. Under the same category, intrinsically-motivated exploration techniques utilize internal incentives to promote visiting unexplored regions of the environment. The intrinsic

information can be computed based on the notion of curiosity [34, 33], state novelty [30] or state visitation counts [4] to minimize the error in the agent's predictions on the environment. In the sparse reward domains, existing studies mostly center around providing bonuses for agents to support transition to novel states. Bonus-based exploration methods give a bonus term to the agent in the form of intrinsic reward which can be calculated using state-action visitation counts [38, 4, 21], agent's prediction error of the environment dynamics [34, 33, 2] or the initial value estimates [39, 6]. Whilst our framework introduced in the Chapter 4 seem to have similar taste with count-based bonus approaches, using recency-based state-action visit information embedded in eligibility trace mechanism to measure additional reward term stands its difference.

3.2 Coordinated Exploration

In multi-agent settings, independent exploration of agents is ineffective since agents may visit already explored areas of state-space redundantly. This problem is exacerbated when the reward and transition dependency among agents are ignored. To address this, [42] coordinates agents' exploration with two methods that consider influence among agents for learning in cooperative settings. Their approaches provide influence-based rewards for agents to encourage them to visit critical regions where they can affect the trajectory and rewards of the other agents in the environment. Similarly, [19] introduces a framework which designs multi-agent intrinsic rewards with respect to explored regions by agents to support coordinated exploration. Although these works promote coordination among multiple agents, our perspective for efficient exploration problem is somewhat disparate since they work under fully-cooperative multi-agent tasks.

3.3 Eligibility Traces

The studies which utilize eligibility traces to improve exploration efficiency is notably scarce in the RL literature. [15] only analyzes the influence of various λ values on the different exploration strategies with Sarsa(λ) algorithm. [44] proposes Team Q(λ)

algorithm that combines multiple trajectories generated by parallel agents to create new trajectories. However, their results on the maze environments show that there is no significant difference over the independent learning setting since newly generated trajectories may be derived from "bad" trajectories that are generated in the early stages of the simulation. On the other hand, our experimental studies in the proposed framework provided in Chapter 4 show that usage of eligibility traces is a beneficial approach for a coordinated exploration.

3.4 Potential Based Reward Shaping

To enhance the learning performance of the RL approaches in terms of speed, most of the studies either apply task decomposition or reward shaping. Task-decomposition methods require identification of subgoals or bottlenecks [26] to divide the task into smaller sub-tasks and learn useful skills to successfully complete each sub-task. For instance, Q-Cut [26] identifies subgoals by solving Max-Flow Min-Cut problem on the transition graph of the agent's experiences in the MDP and finds cuts of the entire state-space. Moreover, LCut [10] method focuses on local transition graphs and uses a partitioning algorithm to extract the local cuts. Both studies suggest generating macro-actions in the form of options framework [40] for agent to learn skills to reach those subgoals. On the other hand, reward shaping [25] based methods form a mechanism to provide additional rewards to the agent to improve exploration. PBRS is a specific version of RS which maintains the policy invariance. [29] introduces a PBRS approach in which the potentials are defined in terms of subgoal achievements. However, the agent acquires subgoals from human participants instead of automatic identification. Furthermore, [13] proposes a landmark-based reward shaping to advance learning speed and quality. They define the potentials in terms of the value of landmarks, however, they focus on partially-observable MDPs. Conversely, our approach, introduced in Chapter 5, works on MDPs and uses state-space segment information in the computation of potentials.

CHAPTER 4

POPULATION-BASED EXPLORATION WITH REPULSIVE REWARD SHAPING MECHANISM USING ELIGIBILITY TRACES

This chapter introduces the problem motivation and the proposed method for efficient exploration problem in detail. The proposed approach called *population-based repulsive reward shaping mechanism using eligibility traces* is explained in the Section 4.2. The extensions and variations of the reward shaping mechanism are described in the Section 4.3. The experimental study for the method along with the results and discussion are presented in the Section 4.4.

4.1 Problem Motivation

From the AI point of view, efficient exploration has an important role in accelerating the agent's learning process, however, it may serve a bigger role considering the real-world contexts. For instance, in a search-and-rescue mission after a hazardous situation, avoiding redundant search is crucial to complete the mission effectively. Therefore, multiple rescue teams follow a "divide-and-conquer" approach to investigate the different regions of the hazardous area to find missing people and gather them in an emergency recovery location. Based on the communication between the teams, they share the area repulsively and perform a coordinated search. Inspired by this critical real-world example, we can think of rescue agents that need to perform the same mission instead of the teams. In this case, a method which coordinates agents' exploration in a way that they repel each other when necessary, would be very helpful to complete the mission. Motivated by this example, we propose a framework to attack the problem of efficient exploration by encouraging agents to discover different

regions of the environment repulsively via shaping the rewards.

4.2 Repulsive Reward Shaping Mechanism

The proposed repulsive reward shaping mechanism uses eligibility traces of a population of agents that is stimulated to explore the undiscovered segments of the state-space. We model the mechanism as a framework having a hierarchical structure of agents such that the upper-level is responsible from leading the learning process through reward shaping whereas the lower-level is from the execution of coordinated exploration directed by the upper-level. The general framework of the repulsive reward shaping mechanism is depicted on Figure 4.1.

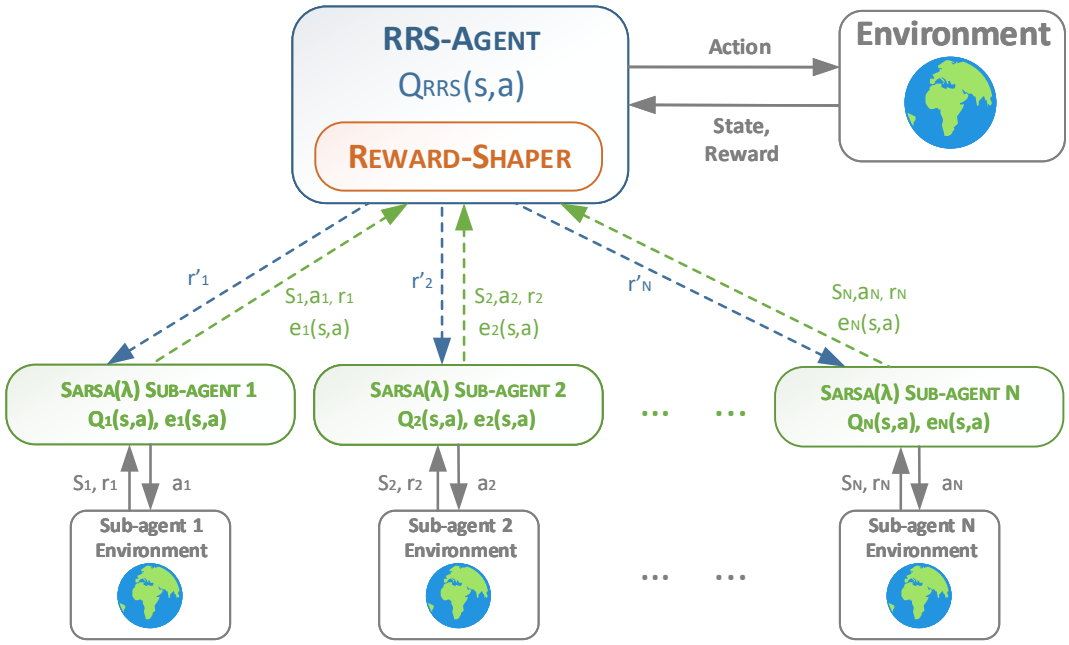


Figure 4.1: The general framework of the method.

Concretely, we formulate the framework as a hierarchical structure for agents in which the high-level consists of a Repulsive-Reward-Shaper agent (*RRS-agent*) and in the low-level a population of identical sub-agents of type Sarsa(λ) with their own Q-tables and policies exist. *RRS-agent* behaves as the principal agent that leads the learning process through acting in the environment and shaping the environmental

reward signal to create repulsion when necessary for its sub-agents that traverse in their own copy of the same environment. *RRS-agent* also learns and updates a global Q-table, Q_{RRS}, using extrinsic environmental reward signals and depending on the observations of all agents in the structure. While Sarsa(λ) sub-agents interact in their own copy of the environment simultaneously, they update their local Q-tables according to the individual experiences and modified rewards sent from the *RRS-agent* in a coordinated manner. The coordination among the population of sub-agents is provided implicitly by the *RRS-agent*. It receives the extrinsic rewards information from its sub-agents and sends the shaped-rewards back to them. Overall, inspired by the parallel computation idea, the main role of sub-agents in this framework is to train a global Q-table with coordinated exploration through repulsive reward shaping based on eligibility traces which will provide an informative start for the *RRS-agent*'s exploration. However, one should notice that although we take the advantage of parallelism idea in this structure, the setting is not perceived completely as distributed agents setting since the actions of sub-agents are implicitly affected by the coordinated exploration mechanism while traversing in their own copy of the same environment.

The mechanism also unifies eligibility trace information collected from sub-agents' population, implicitly. If an agent visits a specific state-action pair $(s, a) \in (\mathcal{S}, \mathcal{A})$ that is already visited *recently* by all other sub-agents, then that sub-agent receives a punishment to repel it from such state-action pairs and to encourage exploring undiscovered regions of the environment.

Consider for a set of $\mathcal{N}$ sub-agents, each sub-agent $i \in \mathcal{N}$ receives an extrinsic reward signal $r_t^i(s, a)$ from the environment for a specific state $s \in \mathcal{S}$ and action choice $a \in \mathcal{A}$. The sub-agent i is punished by an amount of $c_i(s, a)$ depending on its eligibility trace value $e_i(s, a)$ for state-action pair (s, a), in the case of the eligibility trace values of other sub-agents, $e_j(s, a) \geq k, \quad \forall j \in \mathcal{N} \setminus \{i\}$, where k denotes a trace threshold constant, $k \in (0, 1)$.

The final reward value for sub-agent i at a specific state action pair (s, a) and time

step t, $\tilde{r_t}^i(s,a)$ is computed by ***RRS-agent*** as

$$\tilde{r_t}^i(s,a) = \begin{cases} r_t^i(s,a) - c_t^i(s,a), & \text{if } e_j(s,a) \geq k, \forall j \in \mathcal{N} \setminus \{i\}. \\ r_t^i(s,a), & \text{otherwise.} \end{cases} \tag{4.1}$$

Here, the punishment amount for visiting the recently observed (s,a) pair at time step t for sub-agent i, symbolized with $c_t^i(s,a)$, is computed as

$$c_t^i(s,a) = e_t^i(s,a). \tag{4.2}$$

The choice of such punishment amount makes sense in a way that it creates a distinct repulsion force for sub-agents regarding their previously and recently visited state-action pairs of each agent since the punishment value dynamically changes during an episode while being domain-free. Another effect is that the sub-agent receives a higher punishment when it visits a state-action pair that is more recently visited by the population. This choice also supports balancing exploration and exploitation as the trace values fade away during an episode, the repulsion force diminishes. For the trace type, we employed the replacing eligibility trace defined in (2.21) since it is shown by [35, 39] that the replacing traces perform better than accumulating traces.

The flowchart of the overall learning process is given in the Figure 4.2. Learning with the population-based exploration through repulsive reward shaping using eligibility traces has two main sub-processes called training of ***RRS-agent*** and training of sub-agents. Learning process starts with the training of sub-agents and then continues with the training of ***RRS-agent***. During the sub-agents' training, sub-agents interact in their own environment and their environmental reward signals are shaped according to some criteria on their eligibility traces to create repulsion. Based on the shaped rewards, sub-agents learn their Q-value estimates and update their eligibility traces. At this point, the Q-value estimates of ***RRS-agent*** that are stored in Q_{RRS} table is also updated using sub-agents' experiences and original (non-shaped) rewards. After the phase of training of subagents is completed, the actual learning process starts with a pre-trained Q-table. This time ***RRS-agent*** interacts with its environment and learns the Q_{RRS} table. Upon the completion of the last episode, the process terminates and outputs value estimates for each state-action pair.

The detailed pseudocode of learning with population-based exploration through re-

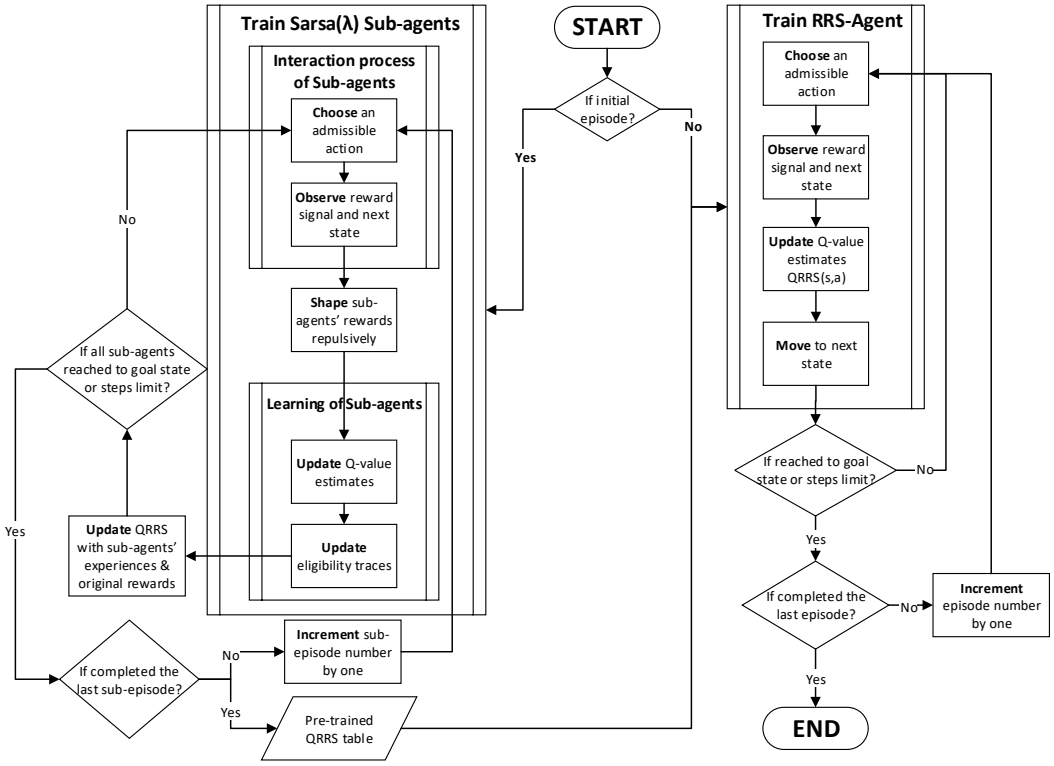


Figure 4.2: The flowchart of the learning process.

pulsive reward shaping mechanism using eligibility traces is given in Algorithm 1. At the beginning of the learning process, before the *RRS-agent*'s training on global Q_{RRS} table starts, sub-agents are trained with `trainSubagents` procedure given in Algorithm 2. This procedure consists of parallel & coordinated training of sub-agents with the reward shaping mechanism. The interesting part of the `trainSubagents` procedure compared to distributed training of independent Sarsa(λ) agents is that after each step, extrinsic rewards received by sub-agents are shaped with the `repulsiveRewardShape` procedure given in Algorithm 3. This procedure takes the necessary information regarding the sub-agents' experiences at a certain time step, shapes the reward signals using repulsion rule explained in (4.1) and returns the modified rewards back. The shaped rewards are then used in the calculation of TD error (line 13 of Algorithm 2) for individual learning. On top of that, the global Q-table Q_{RRS} is also updated with TD error computed using extrinsic rewards (r_i) and sub-agents' experiences (line 14 of Algorithm 2). Sub-agents' training procedure lasts either all of them reach the goal state or $steps$ limit and outputs a pre-trained Q_{RRS} table which will serve as an informative start for the *RRS-agent*'s state-action value

estimates. After training of sub-agents is actualized, *RRS-agent* starts its training on Q_{RRS} table that is initialized with `trainSubagents` procedure. *RRS-agent* then behaves as a Sarsa learner and finally outputs the global state-action value estimates with Q_{RRS}.

4.3 Reward Shaper Variations

Repulsion can be enforced not only with the punishment on sub-agents as introduced in the previous section but also by giving support to the agents who discovered the unexplored regions of the environment. Moreover, we can use different punishment strategies to create repulsion in a coordinated manner. Thus, we investigated the variations and extensions of repulsive reward shaping mechanism under two categories as *bonus-based* and *punishment-based* variations of the reward shaper.

4.3.1 Bonus-based Variations of the Reward Shaper

In this section, we introduce the bonus-based variations of the reward shaper mechanism. The overall idea of the following variations is that instead of punishing the sub-agents for moving to the recently visited regions of the environment, we can provide bonus to the ones who transitions to novel state-action pairs as in the related studies under the category of bonus-based exploration.

4.3.1.1 Bonus-based Reward Shaper

In the bonus-based repulsive reward shaping mechanism, if an agent visits a specific state-action pair $(s, a) \in (\mathcal{S}, \mathcal{A})$ that is *not* visited *recently* by all other sub-agents, then that sub-agent receives a bonus term to encourage such behavior. The novelty of a state-action pair for a specific sub-agent is described as having no recent visitation by all other sub-agents.

Again, consider for a set of $\mathcal{N}$ sub-agents, each sub-agent $i \in \mathcal{N}$ receives an extrinsic reward signal $r_t^i(s, a)$ from the environment for a specific state $s \in \mathcal{S}$ and action

Algorithm 1: Learning with Population-based Exploration Through Repulsive Reward Shaping Using Eligibility Traces

Input : $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$

Set of agents $\mathcal{N} \cup \{RRS\}$

Learning rate $\alpha \in [0, 1]$

Exploration rate $\varepsilon \in [0, 1]$

Discount factor $\gamma \in [0, 1]$

Trace decay rate $\lambda \in [0, 1]$

Trace threshold constant $k \in [0, 1]$

Step limit $steps \geq 1$

Number of episodes $M \geq 1$

Number of sub-episodes $M_{sub} \geq 1$

Output: Q_{RRS}

Initialization:

$Q_{RRS}(s, a) = 0, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$

$episode = 0$

for $episode = 0$ **to** M **do**

  Initialize $s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily for RRS agent

  if $episode = 0$ **then**

    $Q_{RRS} \leftarrow$ `trainSubagents()`

  end if

  while *s is not terminal or $steps$ is not reached* **do**

    Take action a, observe r, s'

    Choose $a' \leftarrow$ `EPSILON-GREEDY`(Q_{RRS}, ε)

    $\delta \leftarrow r + \gamma Q_{RRS}(s', a') - Q_{RRS}(s, a)$

    Update $Q_{RRS}(s, a) \leftarrow Q_{RRS}(s, a) + \alpha\delta$

    $s \leftarrow s', a \leftarrow a'$

  end while

end for

return Q_{RRS}

Algorithm 2: `trainSubagents`

Input : $\mathcal{N}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \alpha, \varepsilon, \gamma, \lambda, k, M_{sub}$

Output: Q_{RRS}

Initialization :

$Q_i(s,a) = 0, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}, i \in \mathcal{N}$

for $subepisode = 0$ *to* M_{sub} **do**

  Initialization pre-episode :

  $e_i(s,a) = 0, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}, i \in \mathcal{N}$

  $observations = \emptyset, actions = \emptyset, traces = \emptyset$

  $rewards = \emptyset, shaped\_rewards = \emptyset$

  Initialize $s_i \in \mathcal{S}, a_i \in \mathcal{A}$ arbitrarily $\forall i \in \mathcal{N}$

  while s_i *is not terminal or* $steps$ *is not reached* $\forall i \in \mathcal{N}$ **do**

    for *each* $i \in \mathcal{N}$ **do**

      Take action a_i, observe r_i, s'_i

      Append s_i to $observations$, a_i to $actions$,

      r_i to $rewards$, e_i to $traces$

      Choose $a'_i \leftarrow$ `EPSILON-GREEDY`(Q_i, ε)

    end for

    $shaped\_rewards \leftarrow$ `repulsiveRewardShape()`

    for *each* $i \in \mathcal{N}$ **do**

      $e_i(s_i, a_i) = 1$; `/* replacing trace */`

      $\delta_i \leftarrow shaped\_rewards[i] + \gamma Q_i(s'_i, a'_i) - Q_i(s_i, a_i)$

      $\delta_i^{RRS} \leftarrow r_i + \gamma Q_{RRS}(s'_i, a'_i) - Q_{RRS}(s_i, a_i)$

      Update $Q_{RRS}(s_i, a_i) \leftarrow Q_{RRS}(s_i, a_i) + \alpha \delta_i^{RRS}$

      for $s \in \mathcal{S}$ *and* $a \in \mathcal{A}$ **do**

        Update $Q_i(s,a) \leftarrow Q_i(s,a) + \alpha \delta_i e_i(s,a)$

        Decay $e_i(s,a) \leftarrow \gamma \lambda e_i(s,a)$

      end for

    end for

    Empty $rewards, shaped\_rewards, traces$

  end while

end for

return Q_{RRS}

Algorithm 3: `repulsiveRewardShape`

Input : $observations, actions, traces, rewards, shaped\_rewards, k$

Output: Modified $shaped\_rewards$

for *each* $i \in \mathcal{N}$ **do**
  $s^* = observations[i], a^* = actions[i]$
  $c_i = 0$; `/* Initialize punishment amount */`
  if $\min_{\forall j \in \mathcal{N} \setminus \{i\}} traces[j](s^*, a^*) \geq k$ **then**
    $c_i = traces[i](s^*, a^*)$
  end if
  $rewards[i] \leftarrow rewards[i] - c_i$
end for
Append $rewards[i]$ to $shaped\_rewards$
return $shaped\_rewards$

choice $a \in \mathcal{A}$. Sub-agent i receives a bonus by an amount of $c_i(s, a)$ depending on its eligibility trace value $e_i(s, a)$ for state-action pair (s, a), in the case of the eligibility trace values of other sub-agents, $e_j(s, a) < k, \quad \forall j \in \mathcal{N} \setminus \{i\}$, where k denotes a trace threshold constant, $k \in (0, 1)$.

Then, the final reward value for sub-agent i at a specific state action pair (s, a) and time step t, $\tilde{r_t}^i(s, a)$ is computed by RRS-agent as

$$\tilde{r_t}^i(s, a) = \begin{cases} r_t^i(s, a) + c_t^i(s, a), & \text{if } e_j(s, a) < k, \forall j \in \mathcal{N} \setminus \{i\}. \\ r_t^i(s, a), & \text{otherwise.} \end{cases} \tag{4.3}$$

The bonus amount $c_t^i(s, a)$ for visiting the recently observed (s, a) pair at time step t for sub-agent i is calculated by

$$c_t^i(s, a) = k - e_t^i(s, a), \tag{4.4}$$

where k denotes the trace threshold constant, $k \in [0, 1]$ and $e_t^i(s, a)$ is the eligibility trace value for (s, a) pair of the time step t. Compared to the punishment case, such bonus term is sensible to use because if the value of $e_t^i(s, a)$ gets bigger (but still less than k), then this means that the pair (s, a) becomes more recently visited by the sub-agent i. Hence, the sub-agent will receive smaller bonus amount. Similar

to the punishment version, this choice creates a different encouragement force for sub-agents considering each one's own visits to state-action pairs. The exploration-exploitation dilemma is also handled with the dynamically changing encouragement support since eligibility trace values decay during an episode.

For this version of the reward shaper, Algorithm 3 should be modified to Algorithm 4 as shown below.

Algorithm 4: `repulsiveRewardShape-BonusBased`

Input : $observations, actions, traces, rewards, shaped\_rewards, k$

Output: Modified $shaped\_rewards$

for *each* $i \in \mathcal{N}$ **do**
  $s^* = observations[i], a^* = actions[i]$
  $c_i = 0$; `/* Initialize bonus amount */`
  if $traces[j](s^*, a^*) < k, \quad \forall j \in \mathcal{N} \setminus \{i\}$ **then**
    $c_i = k - traces[i](s^*, a^*)$
  end if
  $rewards[i] \leftarrow rewards[i] + c_i$
end for
Append $rewards[i]$ to $shaped\_rewards$
return $shaped\_rewards$

4.3.1.2 Bonus-with-Memory-based Reward Shaper

In the bonus-based reward shaper version introduced in Section 4.3.1, the evaluation of whether (s, a) pair is recently visited or not is done considering only the other sub-agents' eligibility trace values. Since we do not take into account the trace value of the sub-agent we are focusing on, this may create a cycling problem. In other words, the sub-agent may visit the same state-action pair which was already discovered by itself again and again, when the pair is not visited by the other sub-agents. To avoid such a case, we introduce *bonus-with-memory-based* reward shaper, in which the condition for giving a bonus depends also on the trace value of the considered sub-agent. The term "memory" in the name of this version comes from the idea that sub-agent's short-term memory itself (eligibility trace) is used to influence its action choices so that it

needs to somehow remember which selections it has done recently.

For the same set of $\mathcal{N}$ sub-agents, RRS-agent computes the final reward value of sub-agent i for visiting the state-action pair (s, a) at time step t, $\tilde{r_t}^i(s, a)$ by

$$\tilde{r_t}^i(s,a) = \begin{cases} r_t^i(s,a) + c_t^i(s,a), & \text{if } e_j(s,a) < k, \forall j \in \mathcal{N} \\ r_t^i(s,a), & \text{otherwise} \end{cases} \tag{4.5}$$

where the bonus term $c_t^i(s, a)$ is evaluated as in (4.4). Note that, in this version the condition in (4.5) considers the complete set of sub-agents, $\mathcal{N}$.

4.3.1.3 Bonus-with-Limited-Steps Reward Shaper

Bonus-with-limited-steps reward shaper approach provides bonus term to the population of sub-agents up to a certain time step. The motivation for this approach comes from the exploration & exploitation trade-off. With the plain bonus-based reward shaper defined in the previous section, it might be a case towards the end of an episode that sub-agents can still be encouraged for visiting state-action pairs which have been discovered in the earlier stages of an episode. Since those state-action pairs have already been visited, redundant exploration might occur if we continue to give bonus to the population. To avoid such problem and balance the trade-off, one way is to limit the time steps in which bonus can be given.

In addition to the parameters of the Algorithm 4, we use $no\_bonus\_after\_step$ parameter to indicate the step number after which the bonus is not given to any sub-agent. We determine the value of the parameter $no\_bonus\_after\_step$ as a constant between 0 and $steps$ limit. For this variant of reward shaper, we modify the Algorithm 4 to the Algorithm 5 below.

4.3.1.4 Bonus-with-Limited-Episode Reward Shaper

Similar to the limited-steps variation, we can also limit the bonus provided to the sub-agents up to a certain episode. By this, we aim to stabilize the learning process and make agents to exploit their learned policies while an efficient exploration of the environment is achieved.

Algorithm 5: `repulsiveRewardShape-BonusWithLimitedSteps`

Input : $observations, actions, traces, rewards, shaped\_rewards, k$ $steps, no\_bonus\_after\_step \in (0, steps)$

Output: Modified $shaped\_rewards$

for *each* $i \in \mathcal{N}$ **do**
  $s^* = observations[i], a^* = actions[i]$
  $c_i = 0$; `/* Initialize bonus amount */`
  if $step\_no \leq no\_bonus\_after\_step$ **then**
    if $traces[j](s^*, a^*) < k, \quad \forall j \in \mathcal{N} \setminus \{i\}$ **then**
      $c_i = k - traces[i](s^*, a^*)$
    end if
    $rewards[i] \leftarrow rewards[i] + c_i$
  end if
end for
Append $rewards[i]$ to $shaped\_rewards$
return $shaped\_rewards$

In this variant, we use $no\_bonus\_after\_episode$ parameter to denote the episode number after which the bonus is not given to any sub-agent. We determine the value of the parameter $no\_bonus\_after\_episode$ as a constant $\in (0, M)$ where M shows the number of episodes. The pseudocode of this procedure is provided in the Algorithm 6.

4.3.2 Punishment-based Variations of the Reward Shaper

In this section, we introduce punishment-based variations of the reward shaper. The general idea of the following reward shaper variations is to prevent redundant exploratory behavior and handle exploration-exploitation trade-off more effectively.

Algorithm 6: `repulsiveRewardShape-BonusWithLimitedEpisode`

Input : $observations, actions, traces, rewards, shaped\_rewards, k$
$M_{sub}, no\_bonus\_after\_episode \in (0, M_{sub})$

Output: Modified $shaped\_rewards$

for *each* $i \in \mathcal{N}$ **do**
  $s^* = observations[i], a^* = actions[i]$
  $c_i = 0$; `/* Initialize bonus amount */`
  if $subepisode \leq no\_bonus\_after\_episode$ **then**
    if $traces[j](s^*, a^*) < k, \quad \forall j \in \mathcal{N} \setminus \{i\}$ **then**
      $c_i = k - traces[i](s^*, a^*)$
    end if
    $rewards[i] \leftarrow rewards[i] + c_i$
  end if
end for
Append $rewards[i]$ to $shaped\_rewards$
return $shaped\_rewards$

4.3.2.1 Punishment-with-Memory-based Reward Shaper

As explained in the Section 4.3.1.2, cycling problem may also arise in the plain punishment-based reward shaper introduced with (4.1). Since the agent is not punished for visiting already explored state-action pairs in the case that such pairs have not been discovered by all other sub-agents yet, there is no discouragement for sub-agent to visit those regions and take the same actions again. This problem can be handled by embedding the eligibility trace value of the sub-agent itself to the punishment condition, so that we obtain the following final reward calculation

$$\tilde{r_t}^i(s,a) = \begin{cases} r_t^i(s,a) - c_t^i(s,a), & \text{if } e_j(s,a) < k, \forall j \in \mathcal{N}. \\ r_t^i(s,a), & \text{otherwise.} \end{cases} \tag{4.6}$$

where the punishment amount $c_t^i(s,a)$ is determined as in (4.2). Notice that in this type of reward shaper, the condition in (4.6) is based on the complete set of sub-agents, $\mathcal{N}$.

4.3.2.2 Punishment-with-Dynamic Threshold Reward Shaper

Instead of a constant trace threshold parameter $k \in (0, 1)$, we can use dynamically changing and preferably decreasing threshold parameter to balance exploration and exploitation. With a high trace threshold value at the beginning of the learning process, we punish the sub-agents only when they visit *very recently* explored regions of the state-space. Hence, exploration is enhanced as the agents are expected to be punished less frequently when there are still major portions of undiscovered regions in the environment. However, as the trace threshold decays during the learning process while episodes are completed, the environment will be discovered gradually by the sub-agents which motivates us to punish them more often to find the smaller regions still waiting to be discovered.

In this type of reward shaper, we define the trace threshold parameter as a function of the episode number,

$$k_t = m - (a * t) \tag{4.7}$$

where k_t denotes trace threshold parameter in episode t, m is a constant that determines the initial value of the threshold, a shows the step size or the rate at which the threshold will decay, and t is the episode number. As shown in the Algorithm 7, the only modification we need to perform on punishment-based reward shaper is that episode number should also be given as input to `repulsiveRewardShape` function. Furthermore, before checking the condition on eligibility traces, trace threshold parameter should be set according to (4.7).

4.3.2.3 Punishment-with-Normal Distribution Reward Shaper

In this version of the punishment-based reward shaper, we inspired by the shape of normal (Gaussian) distribution in the choice of punishment amount. The idea behind this version is, punishment severity should increase gradually over the course of an episode as sub-agents will progressively improve their exploration. After some point, the severity of the punishment should be lessened to avoid unnecessary repulsion since the environment is expected to be already explored towards the end of an episode. Inspired by this, the version is called punishment-with-normal distribution.

Algorithm 7: `repulsiveRewardShape-PunishwDynamicThreshold`

Input : $observations, actions, traces, rewards, shaped\_rewards, m, a, episode$

Output: Modified $shaped\_rewards$

$k = m - (a * episode)$
for *each* $i \in \mathcal{N}$ **do**
  $s^* = observations[i], a^* = actions[i]$
  $c_i = 0$; `/* Initialize punishment amount */`
  if $\min_{\forall j \in \mathcal{N} \setminus \{i\}} traces[j](s^*, a^*) \geq k$ **then**
    $c_i = traces[i](s^*, a^*)$
  end if
  $rewards[i] \leftarrow rewards[i] - c_i$
end for
Append $rewards[i]$ to $shaped\_rewards$
return $shaped\_rewards$

For this, we generate a list of random samples from a normal (Gaussian) distribution having mean 0 and standard deviation of 1. The size of the list is taken as *steps* limit. The complete pseudocode of this version is provided in the Algorithm 8.

4.3.2.4 Punishment-with-Delay Reward Shaper

Punishment-with-delay reward shaper approach generates a repulsion force in a delayed manner instead of directly repelling sub-agents after each time step. No sub-agent is penalized for visiting a discovered state-action pair up to a certain time step in an episode. This approach promotes exploration as we force the sub-agents population to discover unvisited segments of the environment after they gather individual information on their environment without any restrictions.

Here, we use a parameter $delay\_step$ to define the number of steps in which the sub-agents are not punished in an episode. We determine the value of $delay\_step$ as a constant between 0 and $steps$ limit. To reflect the idea of delay, we modify the Algorithm 3 to the Algorithm 9.

Algorithm 8: `repulsiveRewardShape-PunishwNormalDist`

Input : $observations, actions, traces, rewards, shaped\_rewards, step\_no$

Output: Modified $shaped\_rewards$

Initialize $samples \leftarrow []$

$samples \leftarrow$ Generate random samples from $Normal(0, 1)$

Sort $samples$ according to the probability density

for *each* $i \in \mathcal{N}$ **do**

  $s^* = observations[i], a^* = actions[i]$

  $c_i = 0$; `/* Initialize punishment amount */`

  if $\min_{\forall j \in \mathcal{N} \setminus \{i\}} traces[j](s^*, a^*) \geq k$ **then**

    $c_i =\mid samples[step\_no] \mid$

  end if

  $rewards[i] \leftarrow rewards[i] - c_i$

end for

Append $rewards[i]$ to $shaped\_rewards$

return $shaped\_rewards$

4.3.2.5 Punishment-with-Delay-Episode Reward Shaper

Punishment-with-delay-episode reward shaping strategy is used to create delayed repulsion force similar to the variant described in the Section 4.3.2.4. However, instead of a certain time step, we begin to penalize sub-agents for transitioning to explored regions of the environment after a specific episode. We use a parameter called $delay\_episode$ to denote the number of episodes in which the sub-agents are not punished during the learning process and determine its value as a constant between 0 and M, where M shows the total number of episodes. To obtain this version of the mechanism, we need to change the `repulsiveRewardShape` procedure as in the Algorithm 10.

Algorithm 9: `repulsiveRewardShape-PunishmentWithDelay`

Input : $observations, actions, traces, rewards, shaped\_rewards, k$
$steps, delay\_step \in (0, steps)$

Output: Modified $shaped\_rewards$

for *each* $i \in \mathcal{N}$ **do**
  $s^* = observations[i], a^* = actions[i]$
  $c_i = 0$; `/* Initialize punishment amount */`
  if $step\_no > delay\_step$ **then**
    if $\min_{\forall j \in \mathcal{N} \setminus \{i\}} traces[j](s^*, a^*) \geq k$ **then**
      $c_i = traces[i](s^*, a^*)$
    end if
  end if
  $rewards[i] \leftarrow rewards[i] - c_i$
end for
Append $rewards[i]$ to $shaped\_rewards$
return $shaped\_rewards$

4.4 Computational Experiments

In this section, the performance of the proposed method is evaluated with a computational study. In Section 4.4.1, sample problem domains used as the experiment set are introduced. Experiment settings are provided in the Section 4.4.2. Results and discussion are presented in the Section 4.4.3.

4.4.1 Sample Problem Domains

To evaluate the performance of our proposed framework, we carried out the experiments using the experiment set given in the Table 4.1. The set consists of well-known versions of GridWorld navigation domains with varying sizes [27] and two versions of Tower of Hanoi [18] mathematical puzzle game.

- GridWorld

Algorithm 10: `repulsiveRewardShape-PunishmentDelayEpisode`

Input : $observations, actions, traces, rewards, shaped\_rewards, k$
$M_{sub}, delay\_episode \in (0, M_{sub})$

Output: Modified $shaped\_rewards$

for *each* $i \in \mathcal{N}$ **do**
  $s^* = observations[i], a^* = actions[i]$
  $c_i = 0$; `/* Initialize punishment amount */`
  if $subepisode > delay\_episode$ **then**
    if $\min_{\forall j \in \mathcal{N} \setminus \{i\}} traces[j](s^*, a^*) \geq k$ **then**
      $c_i = traces[i](s^*, a^*)$
    end if
  end if
  $rewards[i] \leftarrow rewards[i] - c_i$
end for
Append $rewards[i]$ to $shaped\_rewards$
return $shaped\_rewards$

Table 4.1: The size of the domains.

Domain	Size	$\mid S \mid$	$\mid A \mid$
`Six-Rooms`	32×21	606	4
`Six-Rooms Scaled`	11×7	60	4
`Zigzag Four-Rooms`	43×10	403	4
`Zigzag Four-Rooms Scaled`	15×3	39	4
`Tower of Hanoi 3x3`	3 disks, 3 rods	81	4
`Tower of Hanoi 4x3`	4 disks, 3 rods	243	4

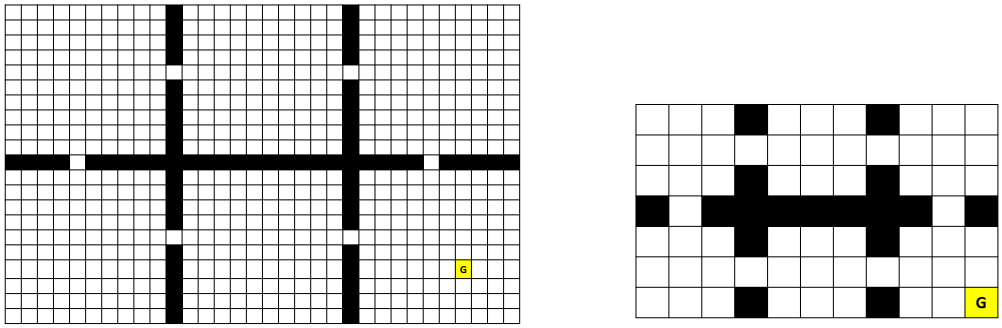
In the experiments we used several GridWorld navigation tasks: `Six-Rooms` [26], `Zigzag Four-Rooms` and scaled versions of these as illustrated in (a), (b), (c) and (d) of Figure 4.3. These domains include several rooms connected by hallways. In these domains, the agent tries to find the goal state starting from an arbitrary state in the upper-left room. The state-space includes a set of possible locations the agent can occupy, whereas the action-space consist of four actions *north, south, east*, and *west*. The agent receives a reward of $+10$ for reaching the goal state denoted with the letter G in all domains. If the agent hits a wall, it receives a reward of -0.1, and for any other movements, there is a small punishment of 0.01 for each step in the `Six-Rooms Scaled` and `Zigzag Four-Rooms Scaled` settings. To observe the performance in sparse reward problems, movements that do not yield to the goal state are not given any reward in `Six-Rooms` and `Zigzag Four-Rooms` environments. The experiments are performed in these domains with the sizes of state-space and action-space as in the Table 4.1.

- Tower of Hanoi

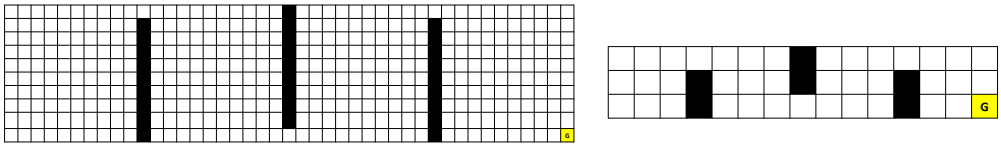
  Tower of Hanoi [18] is a mathematical puzzle played with 3 rods and a number of disks having different diameters. The puzzle with 4 disks begins with a shape illustrated in (a) of Figure 4.4, where the stack of disks having a conical structure and the goal is to move the entire stack to the last rod by sliding the disks onto rods while obeying the puzzle rules to achieve the shape in the (b) of Figure 4.4. To solve the puzzle accurately, the agent should comply the following rules:

  - In each time step, only one disk can be moved.
  - An accurate move is defined as taking the upper disk from one of the rods and placing it to another.
  - In each move, the conical shape in each rod must be preserved, if the rod contains any disk. In other words, a larger disk cannot be placed onto a rod having smaller-diameter disk.

  For this setting, the state is defined as the tuple $\langle$*disk 1 location, disk 2 location, . . . , arm location*$\rangle$. There are 4 possible actions, namely moving *left*, moving

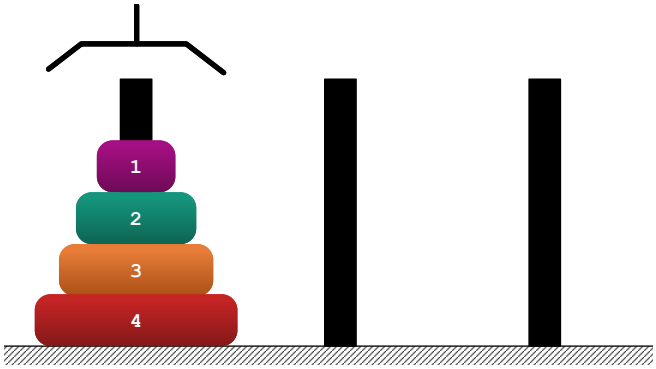


(a) Six-Rooms (b) Six-Rooms Scaled

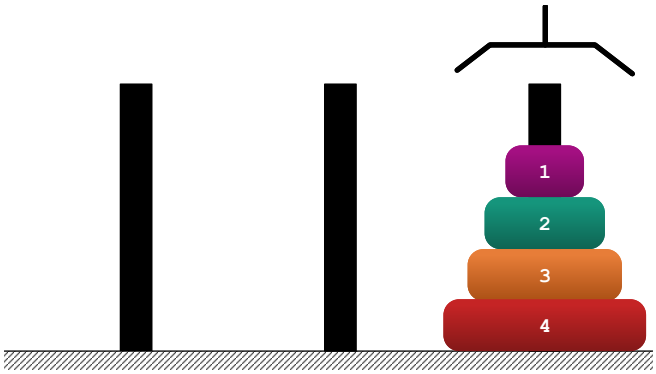


(c) Zigzag Four-Rooms (d) Zigzag Four-Rooms Scaled

Figure 4.3: GridWorld experiment domains.



(a) Tower of Hanoi initial state



(b) Tower of Hanoi goal state

Figure 4.4: Tower of Hanoi with 4 disks & 3 rods domain.

right, *pickup* a disk and *putdown* a disk. The agent receives a reward of $+10$ upon reaching the goal state and 0 for all other cases. We experimented with two versions of this domain having different number of disks as stated in the Table 4.1.

4.4.2 Experimental Settings

In the experiments, learning parameters are set as in the Table 4.2. For repulsive reward shaping mechanisms, we used the parameter set as shown in the Table 4.3. The limit on steps in an episode denoted with $steps$ is set to two different values for each environment. The value is determined as 1000 and 5000 for `Six-Rooms` and `Zigzag Four-Rooms` environments; 1000 and 2000 for scaled versions of these domains. For ToH, 500 and 1000 $steps$ limits are tried. The number of sub-agents is set depending on the domain. The parameter is set to 6 for `Six-Rooms` and its scaled version, 4 for `Zigzag Four-Rooms` and its scaled version, 2 for ToH environments. At the beginning of each episode, the agents are initialized at a random state in the upper left room for GridWorld tasks and the state shown with (a) of Figure 4.4 for ToH task.

All of the experiments are carried out using workstation having Intel® Core™ i7 3.10 GHz processors and 16 GB RAM with 64-bit Microsoft Windows 10 operating system. Each experiment is run for 50 times and the average performance of the runs are reported in the following section.

Table 4.2: Parameter settings for the experiments.

Parameter	**Value**
Learning rate (α)	0.05
Discount rate (γ)	0.9
Exploration probability (ε)	linearly decaying from 0.1 to 0.05
Trace decay rate (λ)	0.9
Number of episodes (M)	1000
Step limit ($steps$)	domain-dependent
Number of sub-agents	domain-dependent

Table 4.3: Parameter settings in the experiments used for repulsive reward shaping.

Parameter	**Value**
Number of sub-episodes (M_{sub})	1000
Trace threshold (k)	0.5
$no\_bonus\_after\_step$	200
$no\_bonus\_after\_episode$	200
$delay\_step$	50
$delay\_episode$	100

4.4.3 Experimental Results and Discussion

This section provides the results of the experiments for performance evaluation of the proposed approaches. First of all, the performances of the frameworks with bonus-based reward shaping mechanisms are presented. Then, the results for punishment-based frameworks are given. Finally, the overall performance comparison is made with the benchmark algorithms.

The learning performances of the methods are evaluated by measuring the average number of steps taken to reach the goal state, average reward per step, and average elapsed time required for learning. Average reward is calculated via dividing total reward in an episode by the episode length and taking the average for all episodes. Since the convergence rate is the significant measure for performance comparison of the methods in the RL literature, we reported the average number of steps and average reward with the 95% bootstrapped confidence intervals in the form of figures. The results display the average of 50 runs for each experiment.

4.4.3.1 *RRS-Agent* with bonus-based variations of the reward shaping mechanism

The performances of the *RRS-Agent* with bonus-based variations of the reward shaping mechanism are summarized in the Table 4.4. Moreover, Figure 4.5 through Figure 4.10 depict the convergence speed of the methods for average number of steps and average reward measures in various problem domains.

As shown in the Table 4.4, *RRS-bonus-withlimitedsteps* outperforms other bonus-based *RRS* methods in terms of all performance measures in `Six-Rooms` domain and its scaled version. Figures 4.5 and 4.6 supports this result as *RRS-bonus-withlimitedsteps* converges much faster to smaller number of steps and higher average reward compared to the others. On the other hand, in a more difficult problem domain which is the `Zigzag Four-Rooms` with limited interaction time (1000 steps), *RRS-bonus-withlimitedepisode* performs better than the others. However, it is also the most inefficient method regarding computation time in this domain. When the interaction time is increased to 5000 steps, *RRS-bonus-withlimitedsteps* becomes again the outperforming framework. However, there is no significant difference between the frameworks considering the converged average number of steps and reward. In the scaled version of this domain, providing limited steps of bonus term works notably better than the others since *RRS-bonus-withlimitedsteps* converges fastest as depicted in the Figure 4.8. For `ToH` environments, we cannot conclude a framework that improves learning performance significantly better in all of them.

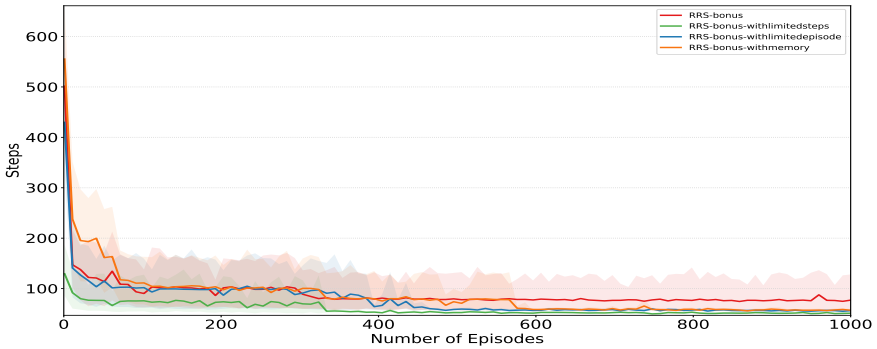
4.4.3.2 *RRS-Agent* with punishment-based variations of the reward shaping mechanism

The performances of the *RRS-Agent* with punishment-based variations of the reward shaping mechanism are provided in Table 4.5. Furthermore, Figure 4.11 through Figure 4.16 sketch the convergence speed of the methods for average number of steps and average reward measures in various problem domains.

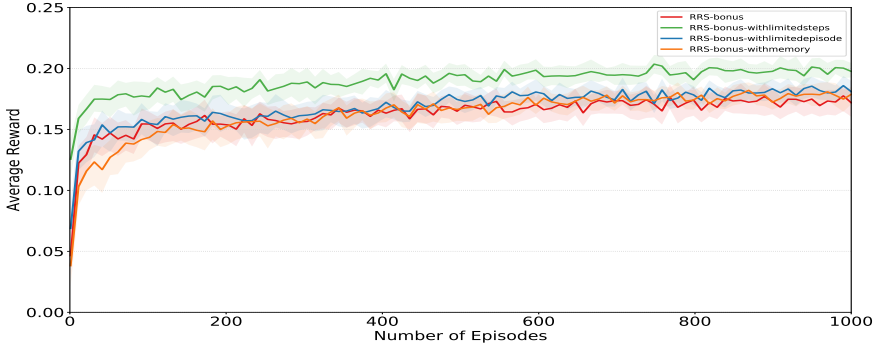
As an overall observation, there is no remarkable difference in average steps and average reward measures among the punishment-based frameworks. Specifically, in the `Six-Rooms` domain, *RRS-punish-dynamicthreshold* beats the other punishment-based frameworks. Although it starts from worse points, *RRS-punish-dynamicthreshold* converges fast to a better number of steps and average reward as depicted in Figure 4.11. However, its good performance does not apply to the rest of the problem domains since it struggles to converge particularly in a more difficult domain, `Zigzag Four-Rooms` with 1000 steps. Considering the rest of the results, we cannot identify a punishment-based framework that performs best among all. Because although

Table 4.4: Learning performances of the methods with bonus-based RRS.

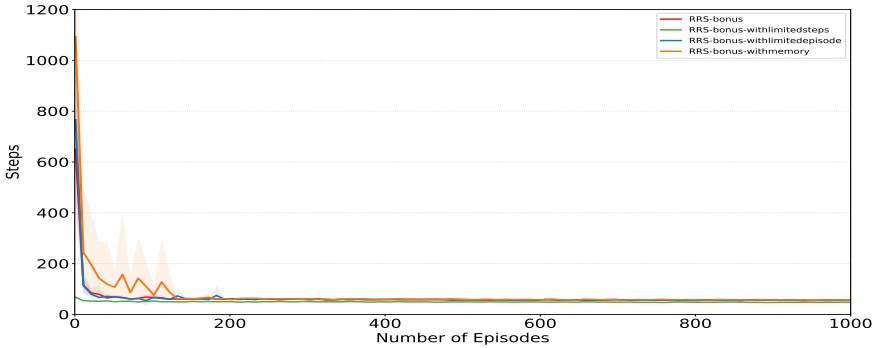
Problem	Method	Average Steps (stdev)		Average Reward (stdev)		Average Elapsed
		over all episodes	of the last episode	over all episodes	of the last episode	Time sec (stdev)
Six-rooms (1000 steps)	RRS-bonus	88.29 (27.16)	77.10 (131.99)	0.16 (0.01)	0.17 (0.04)	7709.77 (1004.57)
	RRS-bonus-withlimitedsteps	**59.93 (11.53)**	**51.22 (5.42)**	**0.19 (0.01)**	**0.20 (0.02)**	6222.48 (886.797)
	RRS-bonus-withlimitedepisode	75.65 (27.90)	56.28 (7.39)	0.17 (0.01)	0.18 (0.03)	7319.60 (978.770)
	RRS-bonus-withmemory	84.56 (43.02)	57.12 (8.47)	0.16 (0.02)	0.18 (0.03)	6835.96 (939.540)
Six-rooms (5000 steps)	RRS-bonus	61.85 (28.60)	54.72 (7.82)	0.17 (0.01)	0.19 (0.03)	50065.9 (8683.82)
	RRS-bonus-withlimitedsteps	**49.41 (1.95)**	**47.36 (6.81)**	**0.21 (0.01)**	**0.22 (0.03)**	26711.6 (7038.28)
	RRS-bonus-withlimitedepisode	62.39 (34.20)	56.56 (7.57)	0.17 (0.01)	0.18 (0.02)	51383.6 (10380.4)
	RRS-bonus-withmemory	70.27 (50.68)	55.68 (7.62)	0.17 (0.01)	0.18 (0.03)	49848.0 (6439.23)
Six-rooms-scaled (1000 steps)	RRS-bonus	20.06 (8.77)	15.40 (1.93)	0.62 (0.03)	0.65 (0.08)	585.108 (112.018)
	RRS-bonus-withlimitedsteps	**14.95 (0.42)**	**14.50 (1.93)**	**0.67 (0.02)**	**0.69 (0.09)**	318.736 (56.7895)
	RRS-bonus-withlimitedepisode	16.59 (4.50)	15.14 (2.17)	0.63 (0.02)	0.66 (0.09)	523.279 (92.3877)
	RRS-bonus-withmemory	62.05 (38.46)	34.78 (137.76)	0.59 (0.05)	0.65 (0.13)	574.727 (109.589)
Six-rooms-scaled (2000 steps)	RRS-bonus	16.06 (1.35)	15.24 (2.46)	0.63 (0.02)	0.66 (0.09)	3542.46 (848.830)
	RRS-bonus-withlimitedsteps	**15.02 (0.46)**	**14.84 (1.87)**	**0.67 (0.02)**	**0.67 (0.08)**	768.155 (75.6613)
	RRS-bonus-withlimitedepisode	15.92 (2.95)	15.28 (2.13)	0.64 (0.02)	0.66 (0.09)	3193.67 (687.978)
	RRS-bonus-withmemory	36.05 (49.83)	15.08 (1.72)	0.62 (0.05)	0.66 (0.08)	4208.24 (824.806)
Zigzag-four-rooms (1000 steps)	RRS-bonus	188.32 (57.63)	123.80 (188.31)	0.11 (0.01)	0.12 (0.03)	5303.89 (820.236)
	RRS-bonus-withlimitedsteps	222.99 (23.71)	135.16 (219.01)	0.11 (0.01)	0.12 (0.04)	4186.33 (592.293)
	RRS-bonus-withlimitedepisode	**149.25 (59.54)**	**80.34 (6.60)**	**0.11 (0.01)**	**0.13 (0.01)**	5304.15 (957.071)
	RRS-bonus-withmemory	376.98 (89.78)	205.94 (316.42)	0.08 (0.02)	0.11 (0.05)	5025.88 (688.050)
Zigzag-four-rooms (5000 steps)	RRS-bonus	89.31 (61.47)	77.10 (6.79)	0.12 (0.01)	0.13 (0.01)	36440.2 (6591.01)
	RRS-bonus-withlimitedsteps	**76.64 (2.91)**	**74.24 (5.42)**	**0.13 (0.00)**	**0.14 (0.01)**	24147.0 (5841.83)
	RRS-bonus-withlimitedepisode	103.42 (80.16)	79.50 (8.68)	0.12 (0.01)	0.13 (0.01)	38837.9 (6184.14)
	RRS-bonus-withmemory	126.63 (157.05)	77.52 (8.03)	0.12 (0.01)	0.13 (0.01)	34711.4 (4358.15)
Zigzag-four-rooms-scaled (1000 steps)	RRS-bonus	943.46 (35.75)	881.56 (318.03)	-0.01 (0.00)	-0.00 (0.02)	1748.47 (352.145)
	RRS-bonus-withlimitedsteps	**73.71 (9.29)**	**59.64 (191.76)**	**0.03 (0.00)**	**0.04 (0.01)**	183.406 (87.4870)
	RRS-bonus-withlimitedepisode	840.76 (56.49)	764.06 (418.08)	-0.00 (0.00)	0.00 (0.02)	1671.21 (333.816)
	RRS-bonus-withmemory	968.96 (15.22)	959.88 (191.65)	-0.01 (0.00)	-0.01 (0.01)	2032.61 (264.136)
Zigzag-four-rooms-scaled (2000 steps)	RRS-bonus	1647.07 (170.27)	1445.06 (888.28)	-0.00 (0.00)	0.00 (0.02)	3451.70 (594.500)
	RRS-bonus-withlimitedsteps	**60.75 (1.81)**	**60.46 (276.95)**	**0.04 (0.00)**	**0.04 (0.01)**	76.2364 (33.0434)
	RRS-bonus-withlimitedepisode	1715.21 (100.47)	1557.66 (817.56)	-0.01 (0.00)	0.00 (0.02)	3349.12 (755.856)
	RRS-bonus-withmemory	1854.01 (53.22)	1829.50 (539.05)	-0.01 (0.00)	-0.01 (0.01)	4225.52 (544.098)
ToH-3x3 (500 steps)	RRS-bonus	422.21 (105.96)	**150.22 (197.34)**	0.05 (0.07)	0.22 (0.13)	780.969 (98.6998)
	RRS-bonus-withlimitedsteps	420.65 (101.06)	163.58 (204.49)	0.05 (0.06)	0.21 (0.14)	761.058 (92.3989)
	RRS-bonus-withlimitedepisode	**410.30 (110.99)**	158.80 (200.41)	0.05 (0.07)	0.21 (0.13)	789.379 (90.8507)
	RRS-bonus-withmemory	414.13 (105.37)	152.50 (201.64)	0.05 (0.06)	**0.23 (0.13)**	482.048 (64.7716)
ToH-3x3 (1000 steps)	RRS-bonus	240.20 (349.89)	33.68 (5.51)	0.22 (0.11)	0.30 (0.04)	1760.16 (219.155)
	RRS-bonus-withlimitedsteps	**236.11 (342.60)**	**32.48 (2.94)**	0.22 (0.11)	**0.31 (0.02)**	1840.21 (175.273)
	RRS-bonus-withlimitedepisode	237.08 (344.74)	34.24 (5.76)	0.22 (0.11)	0.30 (0.04)	1717.61 (196.593)
	RRS-bonus-withmemory	245.48 (349.03)	34.88 (6.12)	0.22 (0.11)	0.29 (0.04)	1165.64 (165.134)
ToH-4x3 (5000 steps)	RRS-bonus	**3733.75 (1422.31)**	508.58 (1157.88)	**0.03 (0.04)**	0.11 (0.05)	12185.3 (1378.40)
	RRS-bonus-withlimitedsteps	3838.83 (1359.60)	788.28 (1637.92)	0.02 (0.03)	0.11 (0.06)	12173.1 (1344.51)
	RRS-bonus-withlimitedepisode	3740.56 (1479.83)	**457.26 (1252.39)**	**0.03 (0.04)**	**0.12 (0.04)**	12294.1 (1307.49)
	RRS-bonus-withmemory	3848.42 (1382.75)	824.66 (1594.96)	0.02 (0.03)	0.10 (0.06)	9424.61 (1284.33)
ToH-4x3 (7000 steps)	RRS-bonus	3277.04 (2868.85)	75.70 (7.54)	0.06 (0.06)	0.13 (0.01)	42879.1 (3964.84)
	RRS-bonus-withlimitedsteps	**2912.14 (2839.66)**	75.90 (8.68)	**0.07 (0.06)**	0.13 (0.01)	45505.9 (578.810)
	RRS-bonus-withlimitedepisode	2999.94 (2850.85)	**71.60 (3.35)**	**0.07 (0.06)**	**0.14 (0.01)**	44525.9 (4681.59)
	RRS-bonus-withmemory	3068.99 (2898.19)	77.20 (12.97)	0.06 (0.06)	0.13 (0.02)	22262.7 (2370.24)



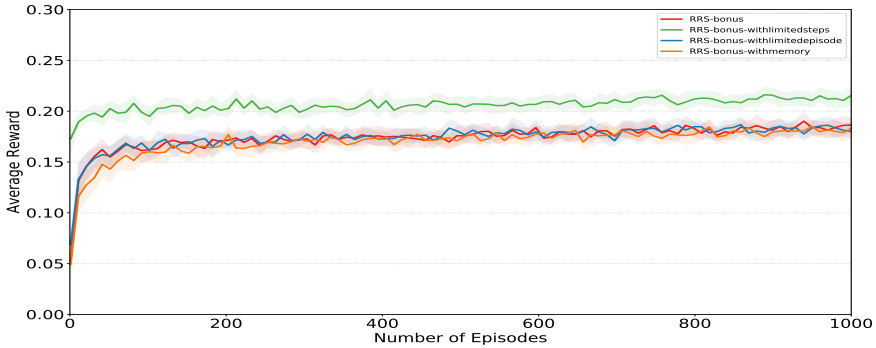
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

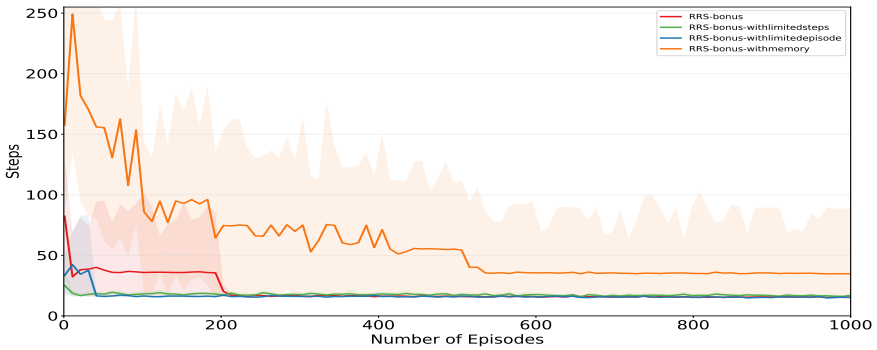


(c) Average steps to reach the goal state under 5000 steps limit.

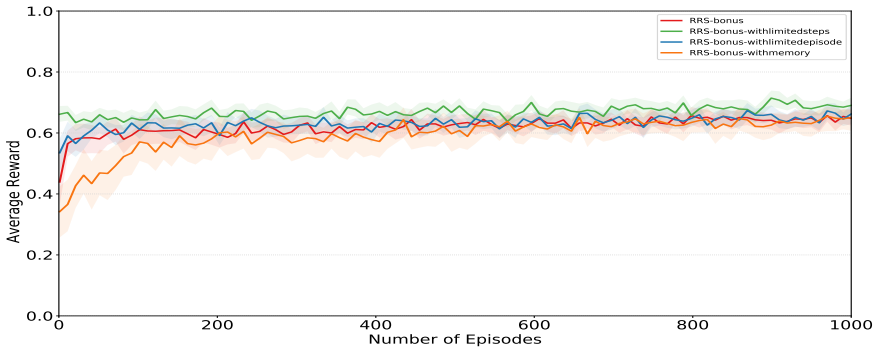


(d) Average reward per episode under 5000 steps limit.

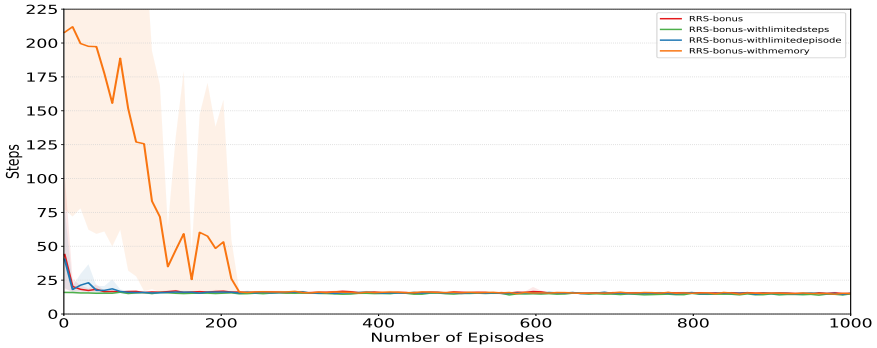
Figure 4.5: Learning performances of the bonus-based reward shaping methods for `Six-Rooms GridWorld` domain under 1000 and 5000 steps limit.



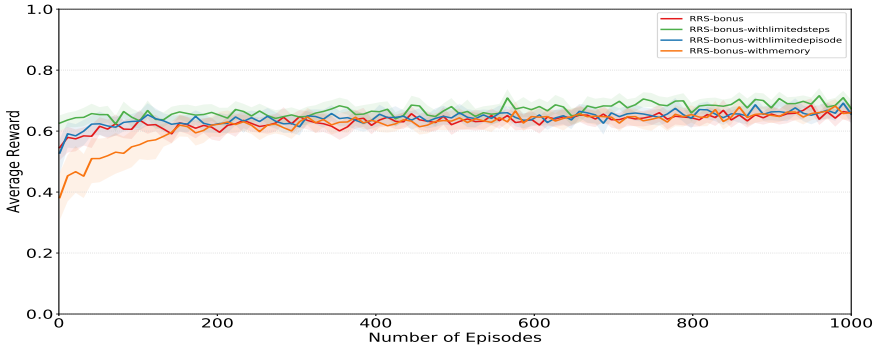
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

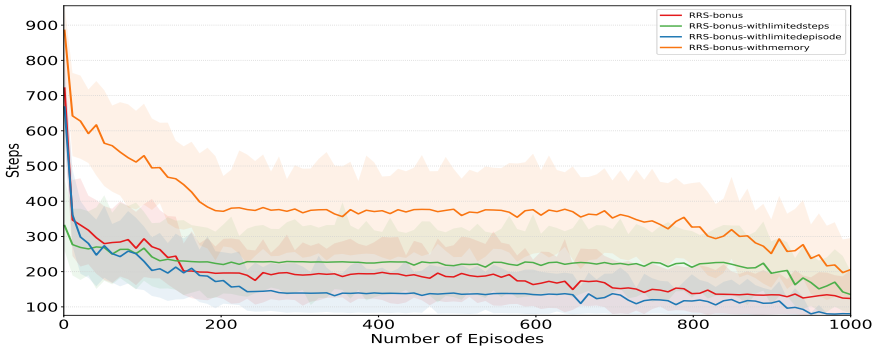


(c) Average steps to reach the goal state under 2000 steps limit.

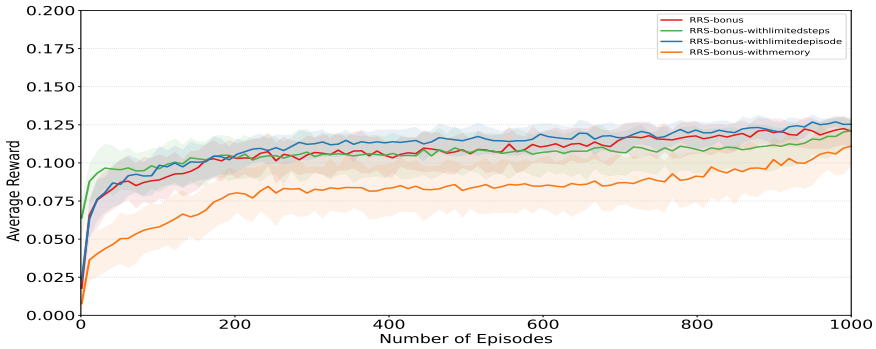


(d) Average reward per episode under 2000 steps limit.

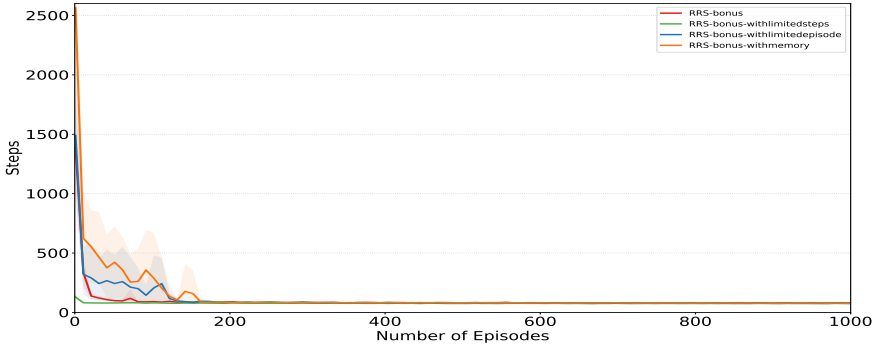
Figure 4.6: Learning performances of the bonus-based reward shaping methods for `Six-Rooms Scaled GridWorld` domain under 1000 and 2000 steps limit.



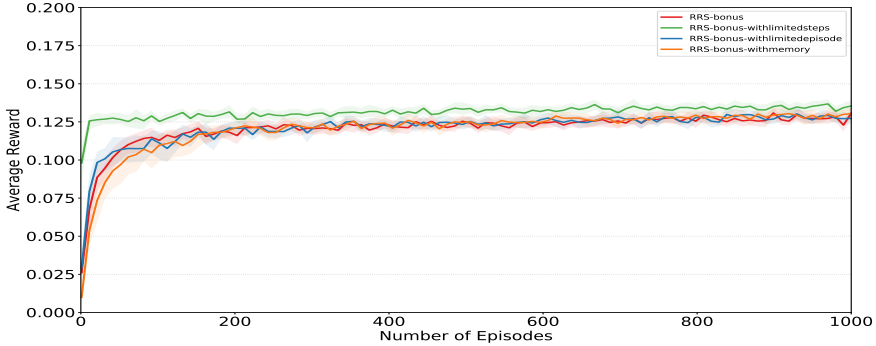
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

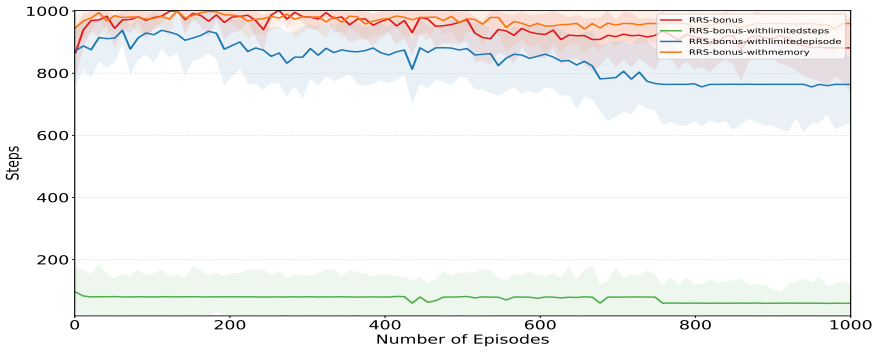


(c) Average steps to reach the goal state under 5000 steps limit.

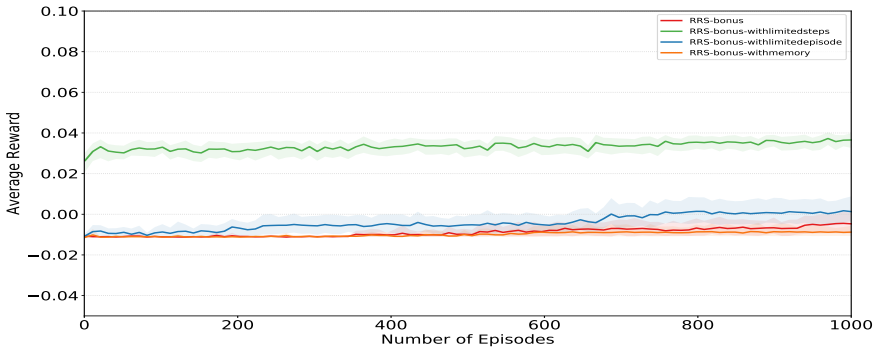


(d) Average reward per episode under 5000 steps limit.

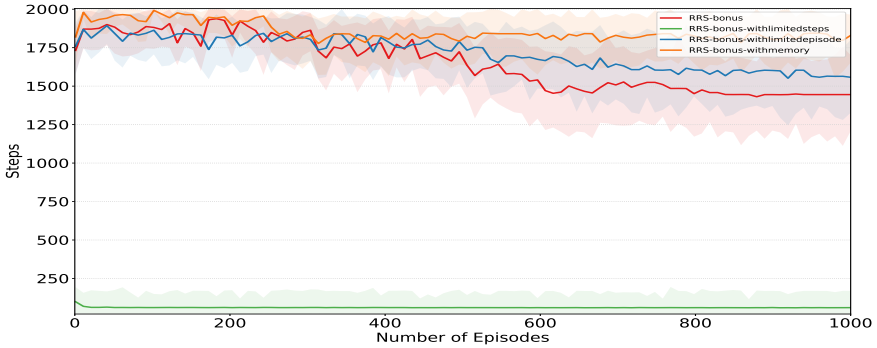
Figure 4.7: Learning performances of the bonus-based reward shaping methods for `Zigzag Four-Rooms GridWorld` domain under 1000 and 5000 steps limit.



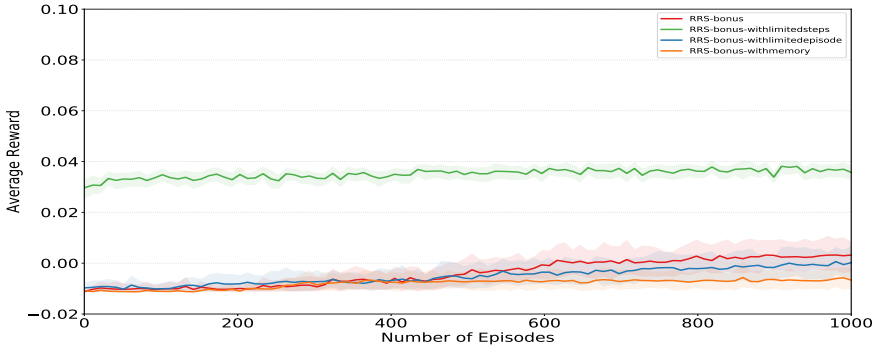
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

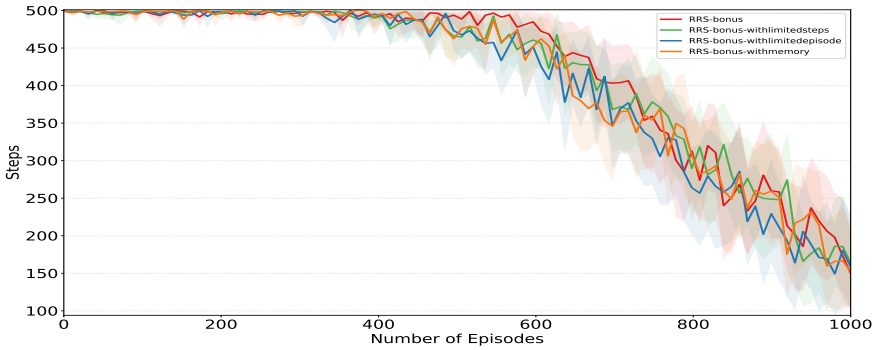


(c) Average steps to reach the goal state under 2000 steps limit.

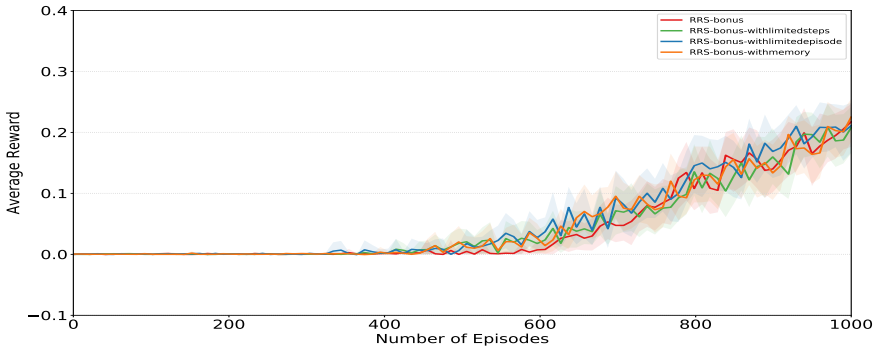


(d) Average reward per episode under 2000 steps limit.

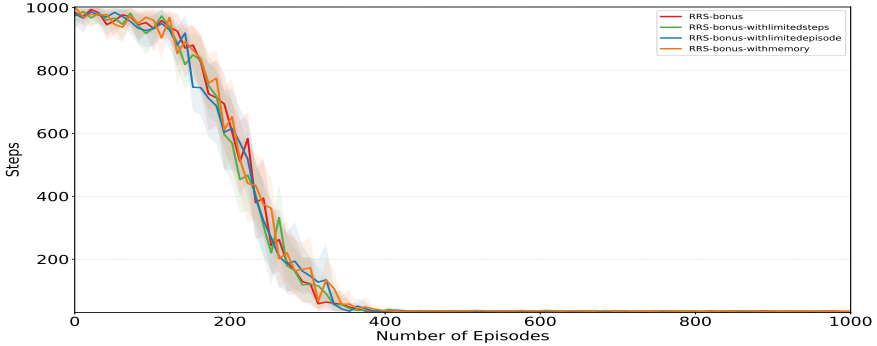
Figure 4.8: Learning performances of the bonus-based reward shaping methods for `Zigzag Four-Rooms Scaled GridWorld` domain under 1000 and 2000 steps limit.



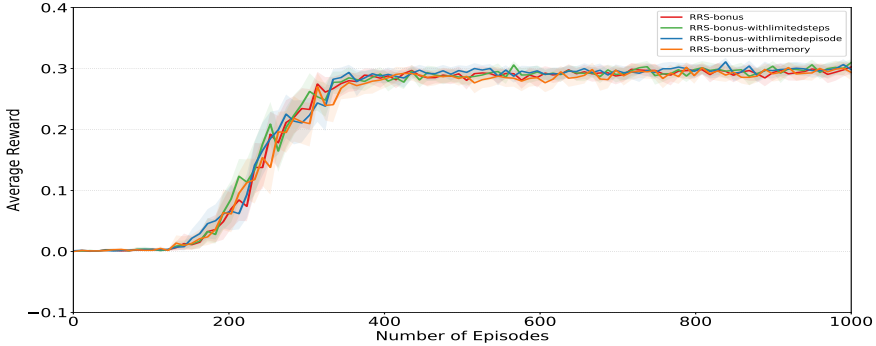
(a) Average steps to reach the goal state in 3 disks version under 500 steps limit.



(b) Average reward per episode in 3 disks version under 500 steps limit.

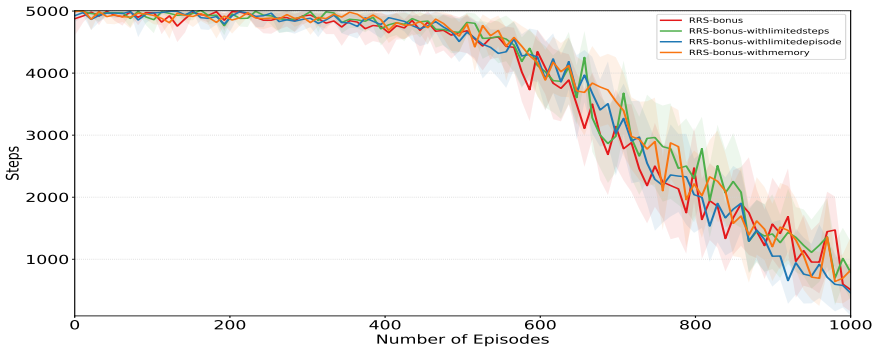


(c) Average steps to reach the goal state in 3 disks version under 1000 steps limit.

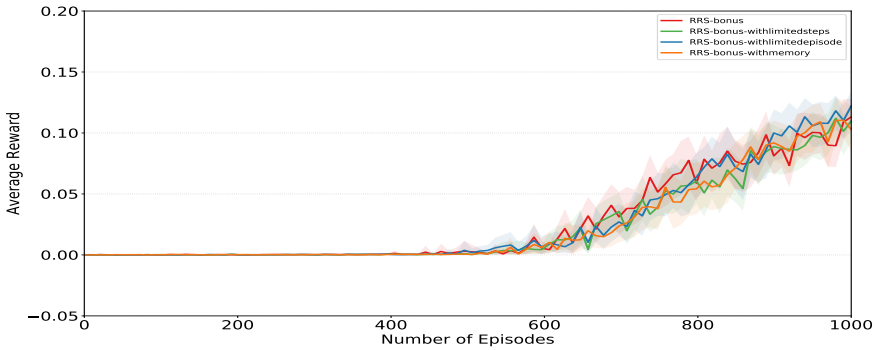


(d) Average reward per episode in 3 disks version under 1000 steps limit.

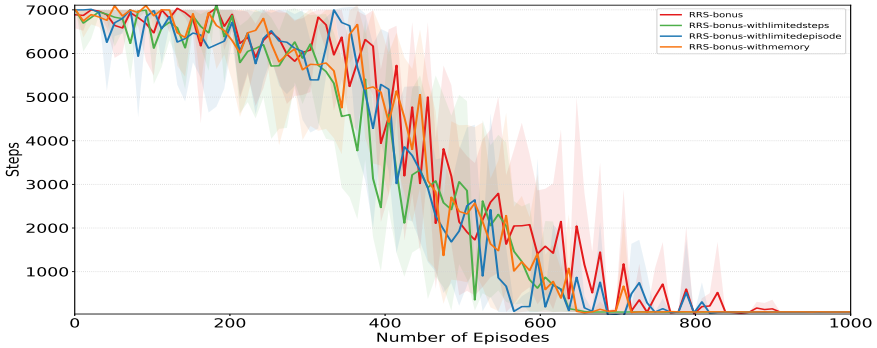
Figure 4.9: Learning performances of the bonus-based reward shaping methods for `Tower of Hanoi` domain under 3 rods and 3 disks version.



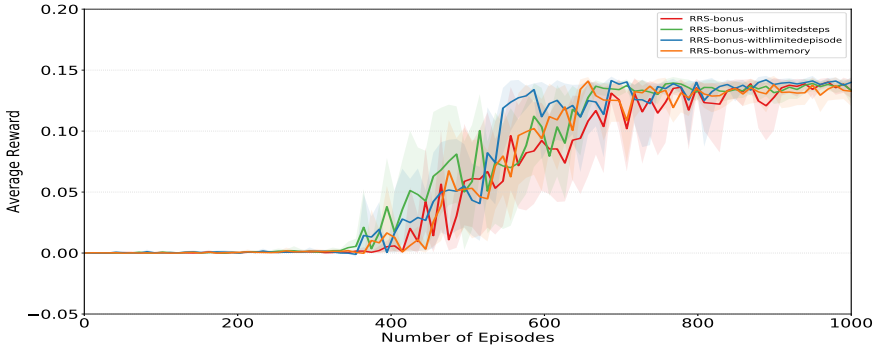
(a) Average steps to reach the goal state in 4 disks version under 5000 steps limit.



(b) Average reward per episode in 4 disks version under 5000 steps limit.



(c) Average steps to reach the goal state in 4 disks version under 7000 steps limit.



(d) Average reward per episode in 4 disks version under 7000 steps limit.

Figure 4.10: Learning performances of the bonus-based reward shaping methods for `Tower of Hanoi` domain under 3 rods and 4 disks version.

RRS-punish has outstanding performance in non-grid world environment `ToH` with 3 disks, delayed versions of *RRS-punish* come into the picture in a more complex version of `ToH` with 4 disks.

4.4.3.3 Performance comparison between bonus-based and punishment-based variations of reward shaping mechanism

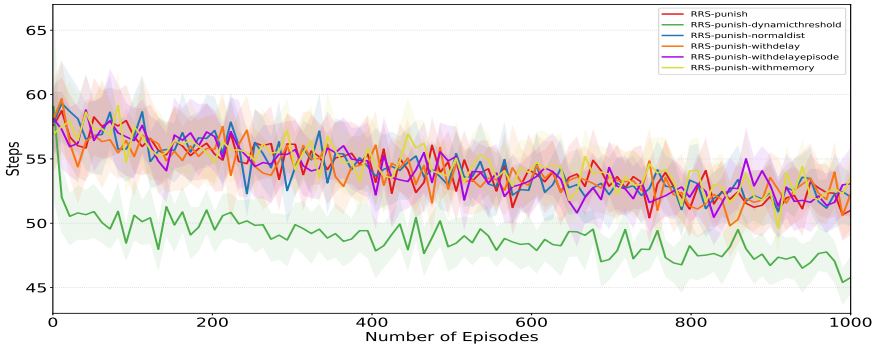
Since sub-agents are trained with an equal number of sub-episodes in each framework, we can directly refer to Tables 4.4 and 4.5 for the learning performance comparison of bonus-based and punishment-based *RRS* methods.

Each *RRS* with punishment method outperforms all bonus-based frameworks in GridWorld problem settings. For non-grid domain `ToH`, bonus category converges to a slightly better number of steps and reward values. Hence, for a better comparison convergence speeds and average elapsed times should be evaluated. As observed from the results, punishment-based frameworks require less amount of time for learning. Thus, a slight improvement on average steps and average reward requires longer computation times with bonus-based frameworks. Based on the previous discussion on bonus-based variations of the reward shaper, we select *RRS-bonus-withlimitedsteps* to compare with punishment-based approaches. From punishment-based frameworks, we determine *RRS-punish-dynamicthreshold* and *RRS-punish* to illustrate the learning speeds on Figures from 4.17 to 4.22.

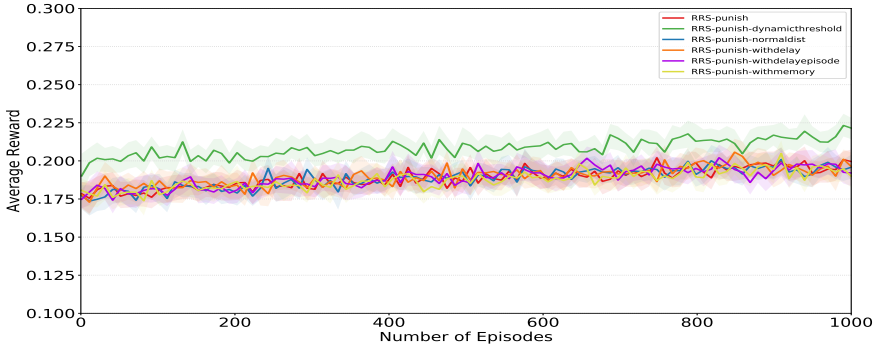
As can be noted from these figures, punishment-based *RRS* learns significantly faster than *RRS-bonus-withlimitedsteps*. We would expect such an outcome because providing bonus to the sub-agents may misdirect them. If a sub-agent receives a bonus after visiting an unexplored state-action pair, then the agent might misinterpret that state-action pair as a subgoal so that it should achieve that high-rewarding pair again and again. Therefore, it is more difficult for the sub-agent to distinguish between receiving a reward for visiting an undiscovered state rather than visiting those specific states. As a result, regional concentrations might be observed in the bonus-based frameworks contrary to repulsion. On the other hand, there is no such misleading in the punishment-based frameworks because the agent will avoid visiting explored state-action pairs as it receives a punishment.

Table 4.5: Learning performances of the methods with punishment-based RRS.

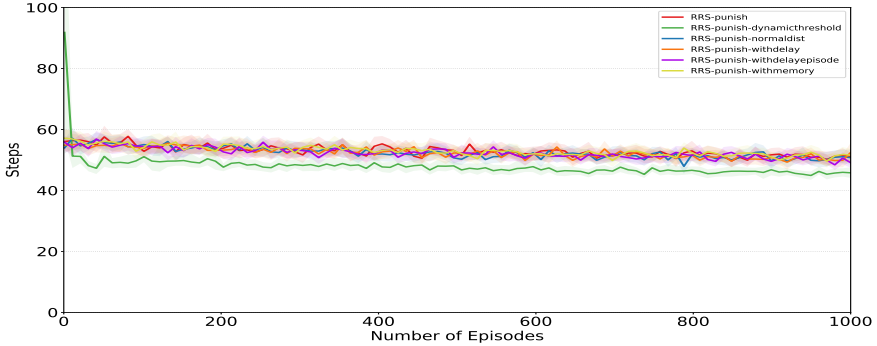
Problem	Method	Average Steps (stdev)		Average Reward (stdev)		Average Elapsed
		over all episodes	of the last episode	over all episodes	of the last episode	Time sec (stdev)
Six-rooms (1000 steps)	RRS-punish	54.23 (1.91)	51.00 (6.76)	0.19 (0.01)	0.20 (0.03)	1450.73 (193.26)
	RRS-punish-dynamicthreshold	**48.88 (1.37)**	**45.78 (5.23)**	**0.21 (0.01)**	**0.22 (0.03)**	1487.48 (172.30)
	RRS-punish-normaldist	54.25 (1.98)	52.04 (7.24)	0.19 (0.01)	0.20 (0.03)	1414.85 (207.68)
	RRS-punish-withdelay	53.96 (1.86)	52.28 (8.17)	0.19 (0.01)	0.20 (0.03)	1428.41 (204.16)
	RRS-punish-withdelayepisode	54.23 (2.05)	53.06 (7.45)	0.19 (0.01)	0.19 (0.03)	1059.43 (152.86)
	RRS-punish-withmemory	54.48 (1.87)	53.54 (6.85)	0.19 (0.01)	0.19 (0.02)	1440.28 (207.92)
Six-rooms (5000 steps)	RRS-punish	52.93 (1.74)	51.20 (5.67)	0.19 (0.01)	0.20 (0.02)	3759.99 (367.69)
	RRS-punish-dynamicthreshold	**47.77 (3.50)**	**45.80 (5.92)**	**0.21 (0.01)**	**0.22 (0.03)**	3945.60 (406.97)
	RRS-punish-normaldist	52.43 (1.71)	50.76 (7.72)	0.20 (0.01)	0.20 (0.03)	3844.82 (375.83)
	RRS-punish-withdelay	52.19 (1.64)	52.44 (8.25)	0.20 (0.01)	0.20 (0.03)	3866.24 (370.06)
	RRS-punish-withdelayepisode	52.38 (1.72)	49.12 (5.55)	0.19 (0.01)	0.21 (0.02)	2222.18 (224.54)
	RRS-punish-withmemory	52.99 (1.73)	51.60 (7.00)	0.19 (0.01)	0.20 (0.03)	3959.42 (414.75)
Six-rooms-scaled (1000 steps)	RRS-punish	15.12 (0.45)	14.80 (1.94)	0.66 (0.02)	0.68 (0.09)	718.742 (78.574)
	RRS-punish-dynamicthreshold	15.10 (0.43)	14.88 (1.69)	0.66 (0.02)	0.67 (0.08)	368.702 (72.224)
	RRS-punish-normaldist	15.15 (0.44)	14.66 (1.82)	0.66 (0.02)	0.68 (0.08)	517.660 (98.078)
	RRS-punish-withdelay	15.10 (0.46)	14.64 (1.83)	0.66 (0.02)	0.68 (0.08)	381.415 (72.927)
	RRS-punish-withdelayepisode	15.09 (0.45)	14.86 (2.07)	0.66 (0.02)	0.67 (0.09)	263.277 (46.708)
	RRS-punish-withmemory	**15.08 (0.41)**	**14.28 (1.80)**	0.66 (0.02)	**0.70 (0.08)**	389.838 (75.570)
Six-rooms-scaled (2000 steps)	RRS-punish	15.15 (0.44)	14.42 (1.61)	0.66 (0.02)	0.69 (0.07)	1350.57 (139.89)
	RRS-punish-dynamicthreshold	15.09 (0.44)	14.68 (2.15)	0.66 (0.02)	0.68 (0.10)	1368.67 (138.39)
	RRS-punish-normaldist	**14.97 (0.43)**	14.22 (1.66)	**0.67 (0.02)**	0.70 (0.08)	1378.18 (140.94)
	RRS-punish-withdelay	15.09 (0.44)	**14.06 (1.50)**	0.66 (0.02)	**0.71 (0.07)**	1369.04 (143.47)
	RRS-punish-withdelayepisode	15.08 (0.43)	14.22 (1.66)	0.66 (0.02)	0.70 (0.08)	646.543 (60.064)
	RRS-punish-withmemory	15.14 (0.44)	14.40 (1.87)	0.66 (0.02)	0.69 (0.09)	1436.75 (149.57)
Zigzag-four-rooms (1000 steps)	RRS-punish	77.21 (1.74)	**72.86 (6.40)**	0.13 (0.00)	**0.14 (0.01)**	2128.26 (749.87)
	RRS-punish-dynamicthreshold	79.43 (3.11)	75.56 (5.88)	0.13 (0.00)	0.13 (0.01)	3462.65 (729.94)
	RRS-punish-normaldist	76.85 (1.76)	75.06 (7.27)	0.13 (0.00)	0.13 (0.01)	2409.07 (935.23)
	RRS-punish-withdelay	77.35 (1.87)	75.44 (7.14)	0.13 (0.00)	0.13 (0.01)	2056.08 (807.81)
	RRS-punish-withdelayepisode	77.17 (1.91)	73.84 (6.20)	0.13 (0.00)	**0.14 (0.01)**	2084.22 (788.45)
	RRS-punish-withmemory	**76.82 (1.83)**	74.28 (7.28)	0.13 (0.00)	**0.14 (0.01)**	2070.72 (690.95)
Zigzag-four-rooms (5000 steps)	RRS-punish	78.31 (1.95)	75.30 (6.29)	0.13 (0.00)	0.13 (0.01)	777.433 (143.21)
	RRS-punish-dynamicthreshold	**76.57 (6.43)**	75.00 (0.00)	0.13 (0.01)	0.13 (0.00)	873.549 (296.62)
	RRS-punish-normaldist	78.24 (1.96)	**74.28 (7.15)**	0.13 (0.00)	**0.14 (0.01)**	729.784 (116.45)
	RRS-punish-withdelay	77.65 (1.89)	75.20 (6.47)	0.13 (0.00)	0.13 (0.01)	740.091 (164.36)
	RRS-punish-withdelayepisode	78.23 (1.93)	75.38 (7.93)	0.13 (0.00)	0.13 (0.01)	755.566 (141.14)
	RRS-punish-withmemory	78.24 (1.97)	75.76 (7.44)	0.13 (0.00)	0.13 (0.01)	753.330 (110.05)
Zigzag-four-rooms-scaled (1000 steps)	RRS-punish	20.94 (0.49)	**19.68 (1.53)**	0.04 (0.00)	0.04 (0.00)	43.6584 (10.669)
	RRS-punish-dynamicthreshold	**20.89 (0.50)**	20.50 (1.75)	0.04 (0.00)	0.04 (0.01)	43.4500 (11.609)
	RRS-punish-normaldist	20.94 (0.49)	20.52 (1.93)	0.04 (0.00)	0.04 (0.01)	43.0697 (10.189)
	RRS-punish-withdelay	20.95 (0.50)	20.92 (2.12)	0.04 (0.00)	0.04 (0.01)	38.8097 (11.667)
	RRS-punish-withdelayepisode	20.91 (0.49)	20.28 (2.16)	0.04 (0.00)	0.04 (0.01)	39.0218 (6.3556)
	RRS-punish-withmemory	20.95 (0.51)	20.16 (2.14)	0.04 (0.00)	0.04 (0.01)	40.0525 (6.7515)
Zigzag-four-rooms-scaled (2000 steps)	RRS-punish	**20.88 (0.49)**	**19.86(1.67)**	0.04 (0.00)	0.04 (0.00)	41.9938 (9.9399)
	RRS-punish-dynamicthreshold	20.90 (0.50)	20.28 (1.87)	0.04 (0.00)	0.04 (0.01)	40.0634 (8.3209)
	RRS-punish-normaldist	20.95 (0.48)	20.76 (2.25)	0.04 (0.00)	0.04 (0.01)	40.2862 (13.876)
	RRS-punish-withdelay	20.90 (0.49)	20.42 (1.98)	0.04 (0.00)	0.04 (0.01)	34.8034 (6.6774)
	RRS-punish-withdelayepisode	20.97 (0.48)	20.28 (1.72)	0.04 (0.00)	0.04 (0.00)	38.7256 (6.5398)
	RRS-punish-withmemory	20.91 (0.48)	19.96 (2.12)	0.04 (0.00)	0.04 (0.01)	41.9240 (9.7160)
ToH-3x3 (500 steps)	RRS-punish	242.88 (343.56)	34.24 (5.97)	0.22 (0.11)	**0.30 (0.04)**	384.977 (54.139)
	RRS-punish-dynamicthreshold	**232.62 (340.36)**	34.06 (5.29)	0.22 (0.11)	**0.30 (0.04)**	408.789 (49.509)
	RRS-punish-normaldist	243.76 (347.61)	**33.76 (5.67)**	0.22 (0.11)	**0.30 (0.04)**	387.715 (44.941)
	RRS-punish-withdelay	249.28 (340.81)	35.54 (6.66)	0.22 (0.11)	0.29 (0.04)	398.664 (53.845)
	RRS-punish-withdelayepisode	239.21 (341.74)	36.12 (6.88)	0.22 (0.11)	0.29 (0.04)	754.467 (78.650)
	RRS-punish-withmemory	252.36 (350.62)	34.66 (6.09)	0.22 (0.12)	**0.30 (0.04)**	388.170 (48.296)
ToH-3x3 (1000 steps)	RRS-punish	**409.02 (117.85)**	**110.44 (170.79)**	**0.05 (0.07)**	**0.25 (0.12)**	828.401 (108.28)
	RRS-punish-dynamicthreshold	417.03 (105.60)	194.20 (206.76)	**0.05 (0.07)**	0.18 (0.14)	875.337 (110.59)
	RRS-punish-normaldist	424.78 (96.77)	158.00 (201.98)	0.04 (0.06)	0.22 (0.14)	847.627 (119.00)
	RRS-punish-withdelay	417.08 (102.95)	183.34 (216.71)	**0.05 (0.06)**	0.20 (0.14)	869.209 (128.37)
	RRS-punish-withdelayepisode	411.08 (111.81)	163.86 (199.29)	**0.05 (0.07)**	0.21 (0.14)	1915.63 (200.41)
	RRS-punish-withmemory	423.34 (100.83)	170.36 (206.04)	0.04 (0.06)	0.20 (0.14)	855.488 (116.71)
ToH-4x3 (5000 steps)	RRS-punish	3739.23 (1490.55)	684.54 (1338.11)	0.03 (0.04)	0.10 (0.06)	5412.95 (578.28)
	RRS-punish-dynamicthreshold	3912.02 (1303.52)	1086.30 (1843.99)	0.02 (0.03)	0.10 (0.06)	5559.18 (723.00)
	RRS-punish-normaldist	3814.14 (1450.31)	767.48 (1633.72)	0.02 (0.04)	**0.11 (0.05)**	6338.78 (592.09)
	RRS-punish-withdelay	3832.95 (1375.77)	670.90 (1519.27)	0.02 (0.03)	**0.11 (0.05)**	5737.12 (702.62)
	RRS-punish-withdelayepisode	**3720.50 (1508.66)**	746.76 (1606.35)	**0.03 (0.04)**	**0.11 (0.05)**	19315.9 (2138.8)
	RRS-punish-withmemory	3784.42 (1403.10)	**522.20 (1150.69)**	0.02 (0.03)	**0.11 (0.05)**	5514.82 (701.58)
ToH-4x3 (7000 steps)	RRS-punish	3016.79 (2859.61)	72.30 (6.21)	**0.07 (0.06)**	**0.14 (0.01)**	23990.5 (3469.5)
	RRS-punish-dynamicthreshold	3207.83 (2861.76)	72.10 (2.39)	0.06 (0.06)	**0.14 (0.00)**	18182.3 (1548.9)
	RRS-punish-normaldist	3043.02 (2872.44)	**70.60 (3.07)**	**0.07 (0.06)**	**0.14 (0.01)**	14766.0 (2658.4)
	RRS-punish-withdelay	**2939.96 (2915.50)**	76.30 (12.70)	**0.07 (0.06)**	0.13 (0.02)	18343.0 (2020.7)
	RRS-punish-withdelayepisode	3282.46 (2827.45)	79.60 (15.81)	0.06 (0.06)	0.13 (0.02)	57395.1 (489.74)
	RRS-punish-withmemory	3108.34 (2818.36)	75.60 (7.31)	0.06 (0.06)	0.13 (0.01)	17605.9 (1593.0)



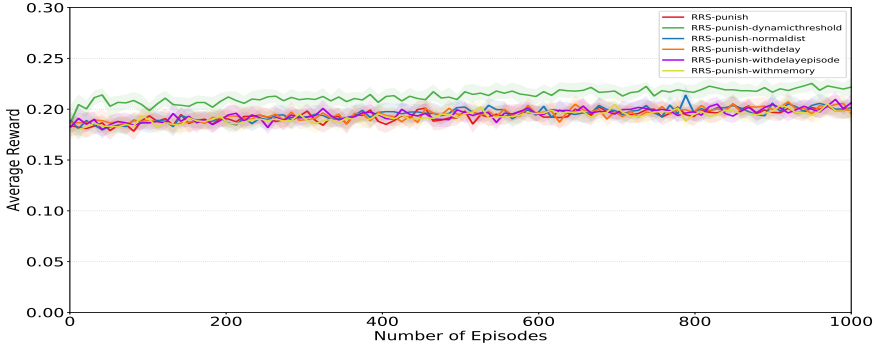
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

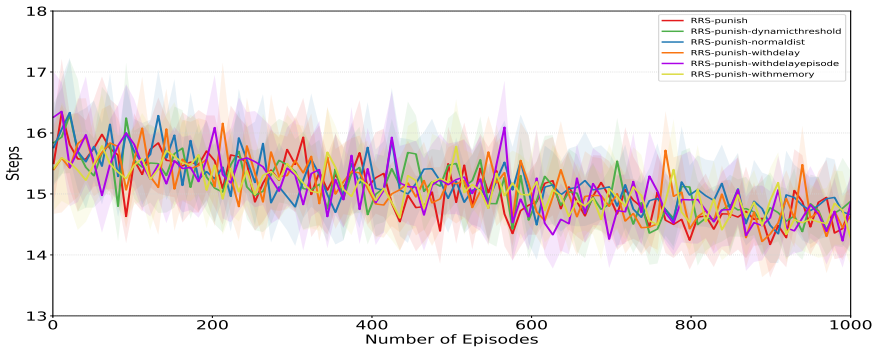


(c) Average steps to reach the goal state under 5000 steps limit.

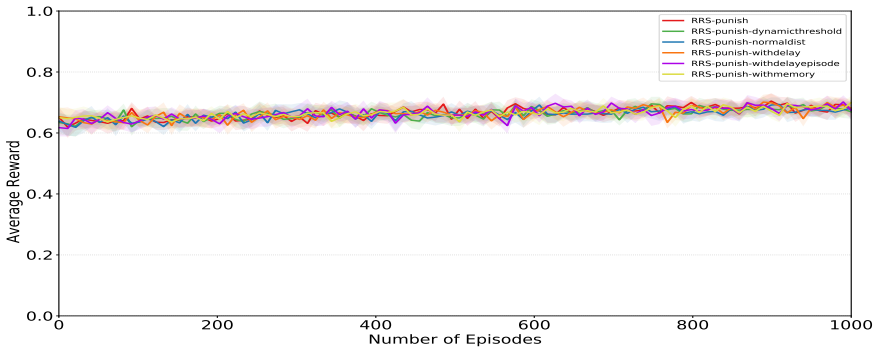


(d) Average reward per episode under 5000 steps limit.

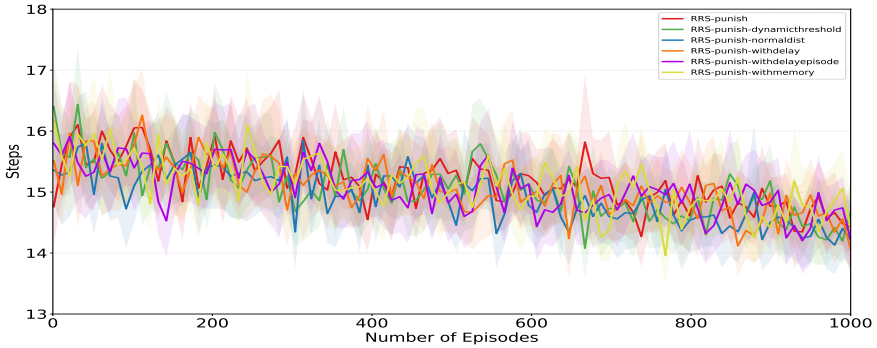
Figure 4.11: Learning performances of the punishment-based reward shaping methods for `Six-Rooms GridWorld` domain under 1000 and 5000 steps limit.



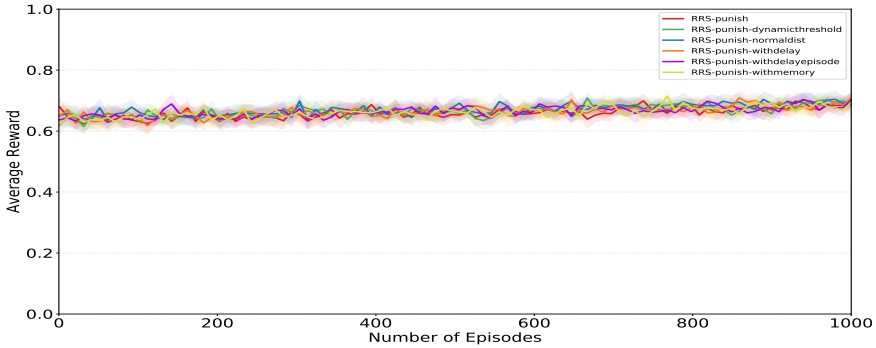
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

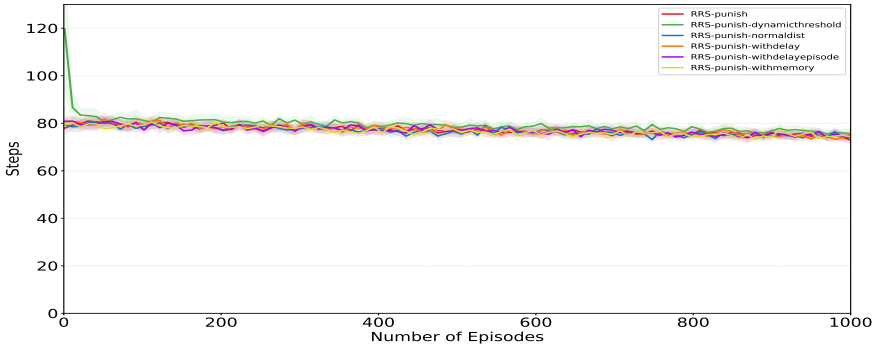


(c) Average steps to reach the goal state under 2000 steps limit.

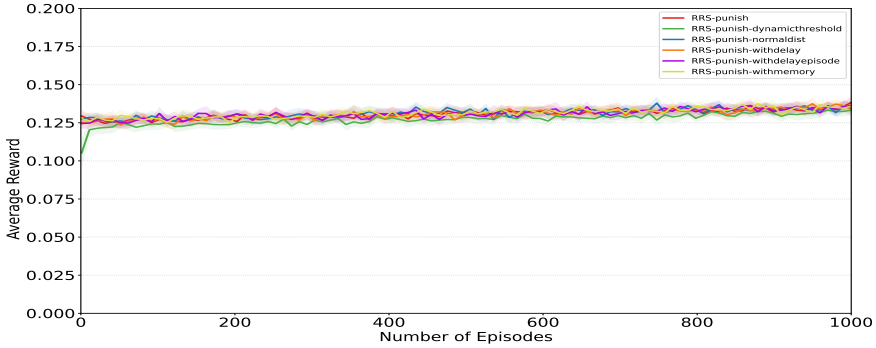


(d) Average reward per episode under 2000 steps limit.

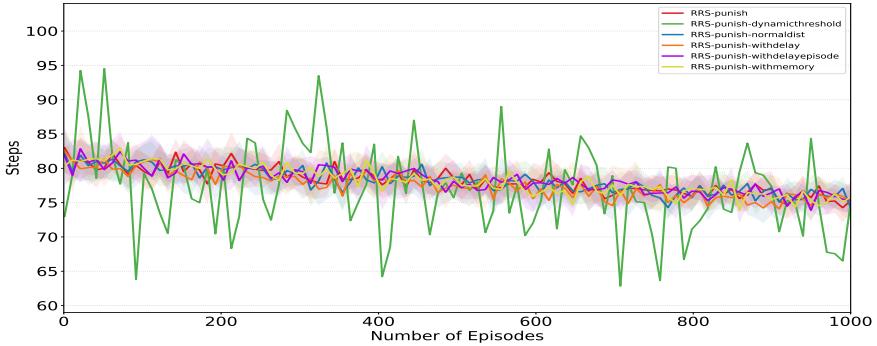
Figure 4.12: Learning performances of the punishment-based reward shaping methods for `Six-Rooms Scaled GridWorld` domain under 1000 and 2000 steps limit.



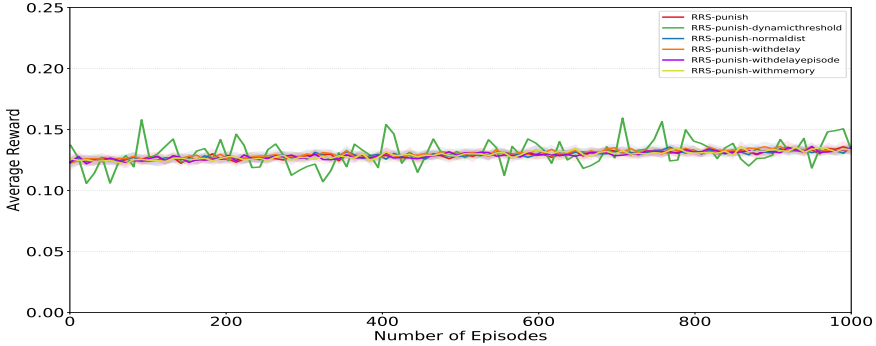
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

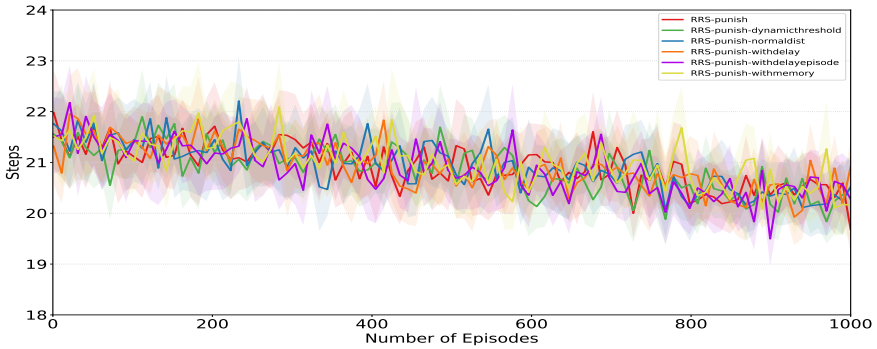


(c) Average steps to reach the goal state under 5000 steps limit.

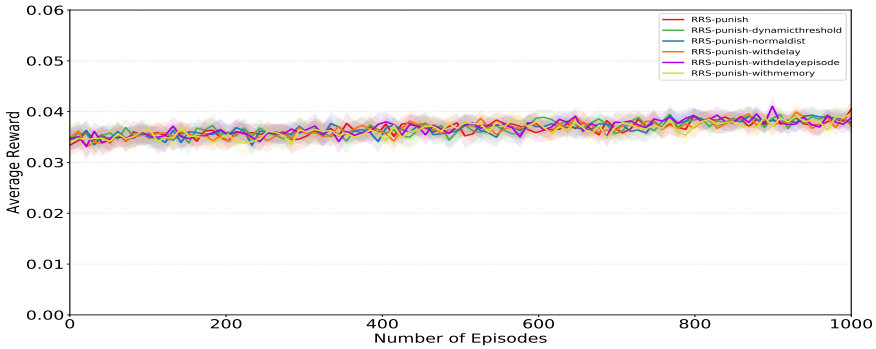


(d) Average reward per episode under 5000 steps limit.

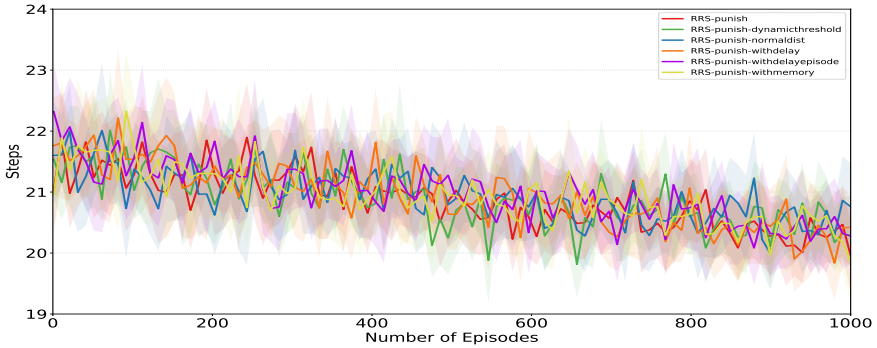
Figure 4.13: Learning performances of the punishment-based reward shaping methods for `Zigzag Four-Rooms GridWorld` domain under 1000 and 5000 steps limit.



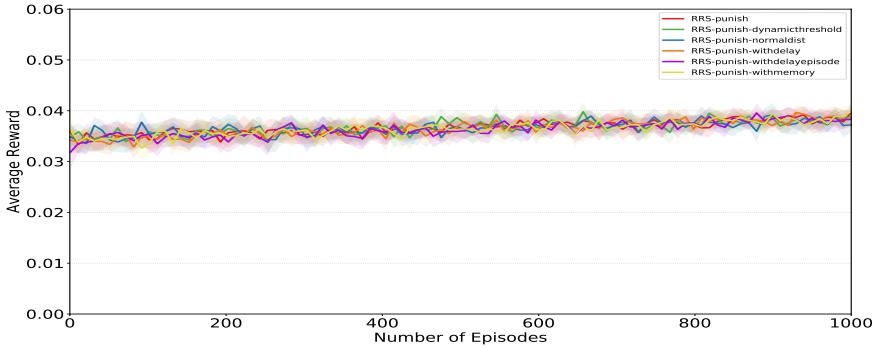
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

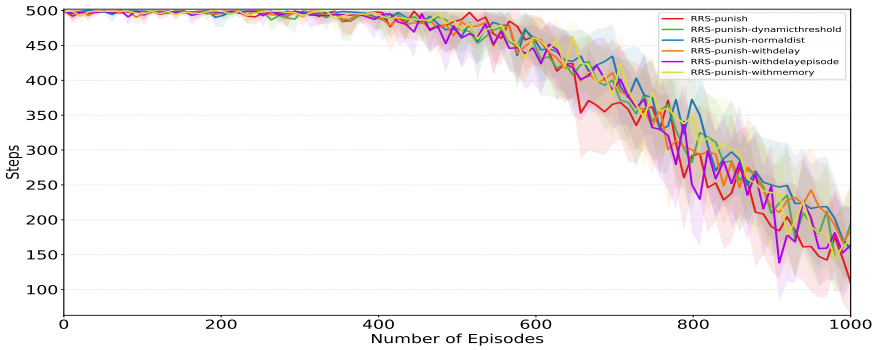


(c) Average steps to reach the goal state under 2000 steps limit.

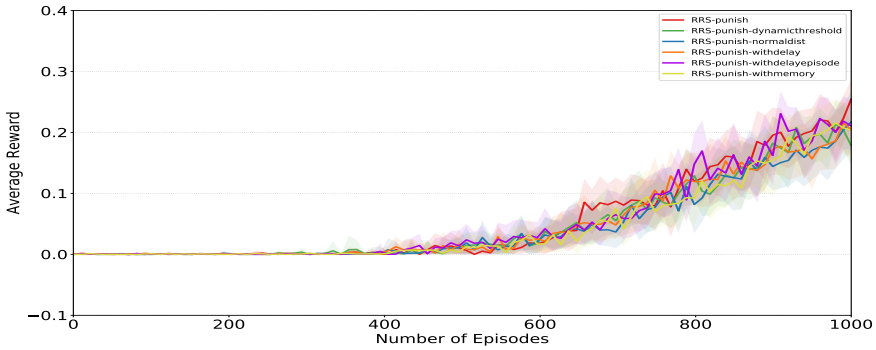


(d) Average reward per episode under 2000 steps limit.

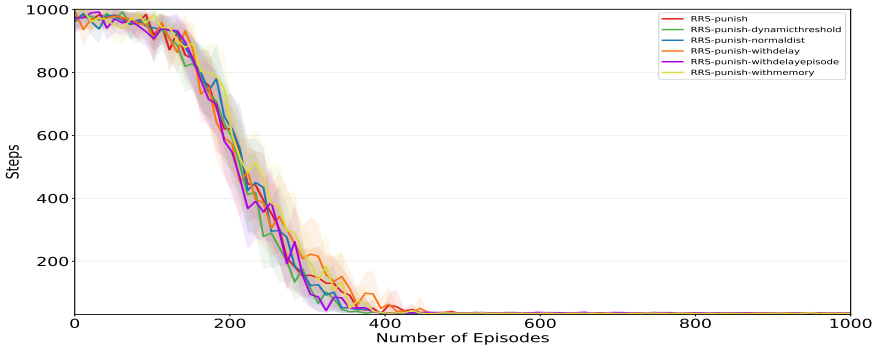
Figure 4.14: Learning performances of the punishment-based reward shaping methods for `Zigzag Four-Rooms Scaled GridWorld` domain under 1000 and 2000 steps limit.



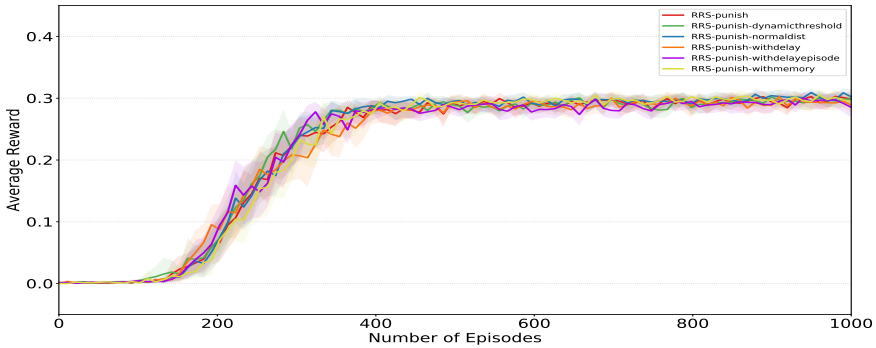
(a) Average steps to reach the goal state in 3 disks version under 500 steps limit.



(b) Average reward per episode in 3 disks version under 500 steps limit.

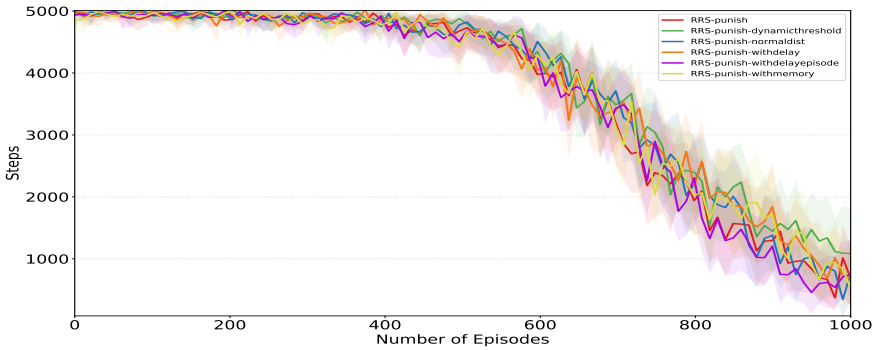


(c) Average steps to reach the goal state in 3 disks version under 1000 steps limit.

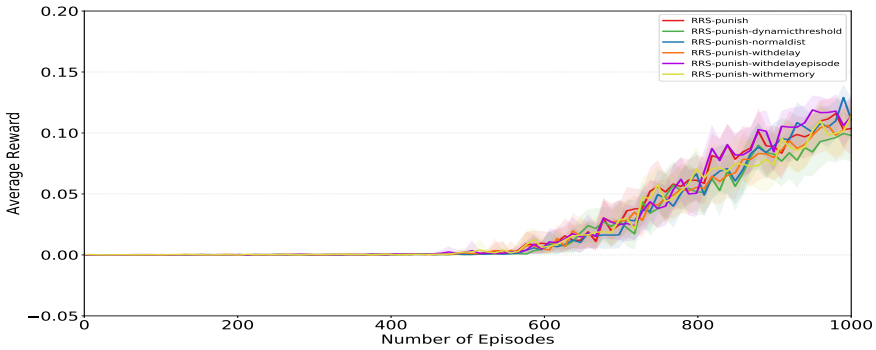


(d) Average reward per episode in 3 disks version under 1000 steps limit.

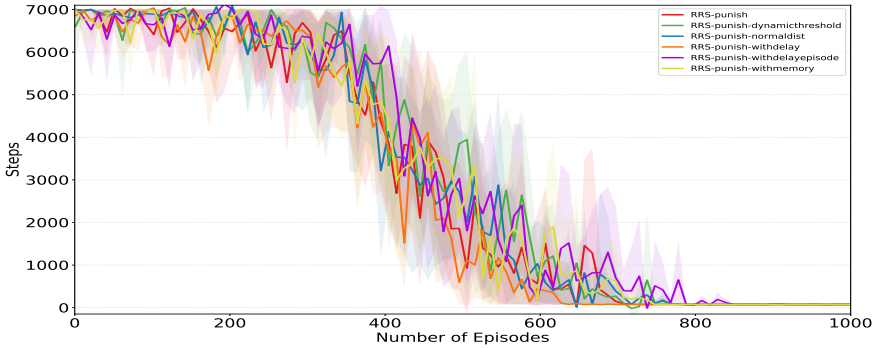
Figure 4.15: Learning performances of the punishment-based reward shaping methods for `Tower of Hanoi` domain under 3 rods and 3 disks version.



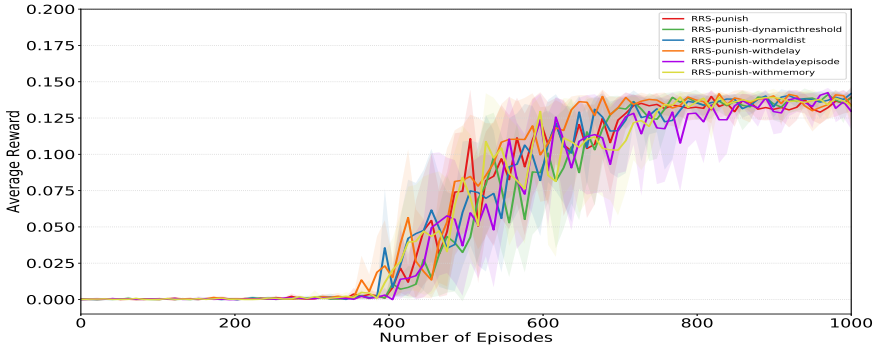
(a) Average steps to reach the goal state in 4 disks version under 5000 steps limit.



(b) Average reward per episode in 4 disks version under 5000 steps limit.



(c) Average steps to reach the goal state in 4 disks version under 7000 steps limit.



(d) Average reward per episode in 4 disks version under 7000 steps limit.

Figure 4.16: Learning performances of the punishment-based reward shaping methods for `Tower of Hanoi` domain under 3 rods and 4 disks version.

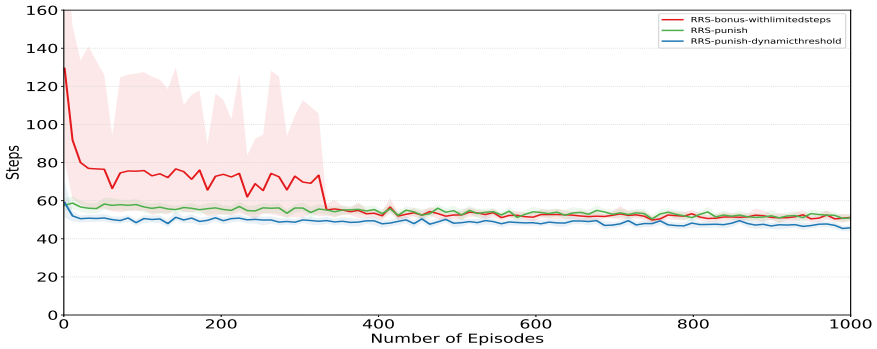
Following these results, our choice is to propose the *RRS-punish* framework and evaluate its learning performance with the benchmarks. Since *RRS-punish* does not involve a design of dynamic threshold parameter which is an important factor that can affect the performance, it is easier to implement a plain version of punishment-based *RRS*. Hence, in the subsequent discussion, we compare the overall learning performance of *RRS-punish* with the benchmark algorithms.

4.4.3.4 Overall performance comparison

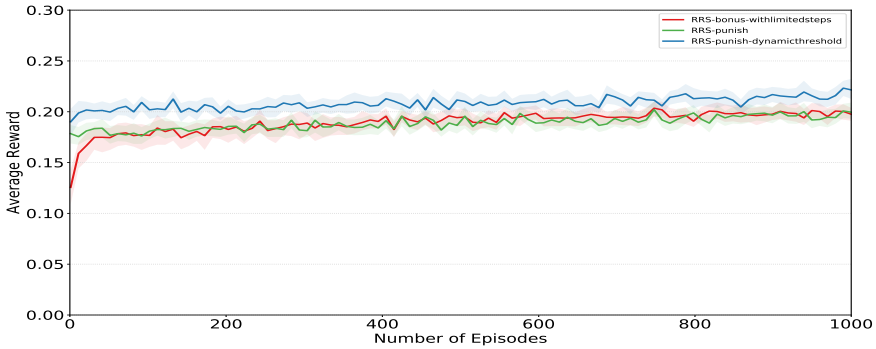
As the benchmark, we used Sarsa and Sarsa(λ) single-agent learners since on-policy algorithms are more suitable for the evaluation of our method as we defined the sub-agents with Sarsa(λ) type.

Furthermore, we also provide an "ablation study" to investigate the contribution of coordination mechanism through repulsive reward shaping in the introduced soft-hierarchical model between *RRS-Agent* and sub-agents. We remove the reward-shaping mechanism between these two levels of agents and call the framework as *SuperAgent-SubAgents*. In the *SuperAgent-SubAgents* framework, there is a hierarchical structure between a super-agent and Sarsa(λ) sub-agents similar to the *RRS-Agent* framework excluding the coordination implied by the repulsive reward shaping mechanism. Hence, this framework differs from the *RRS-Agent* structure only in a way that rewards are not shaped during the training of sub-agents to create coordination. However, super-agent still learns from a pre-trained global Q-table with the experiences of sub-agents. We add *SuperAgent-SubAgents* framework to the benchmarks and provide the overall performance comparison in Table 4.6. We also depict the learning speeds in Figures 4.27 to 4.32. To evaluate the exploration power of the proposed framework, we illustrate the state-space coverages for GridWorld domains in Figures 4.23 to 4.26.

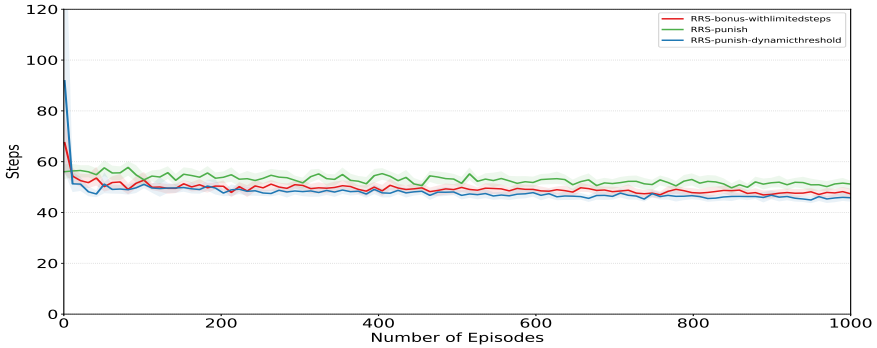
Through a series of sub-episode number tuning sessions, we determined the number of sub-episodes for *RRS-punish* framework as 25. Thus, training of sub-agents will continue for 25 episodes and the exploration amount after the exploration phase i.e. training of sub-agents is shown in the state-space coverage figures.



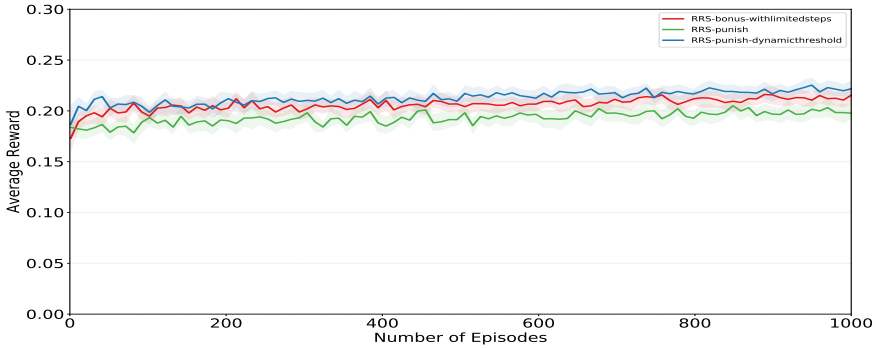
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

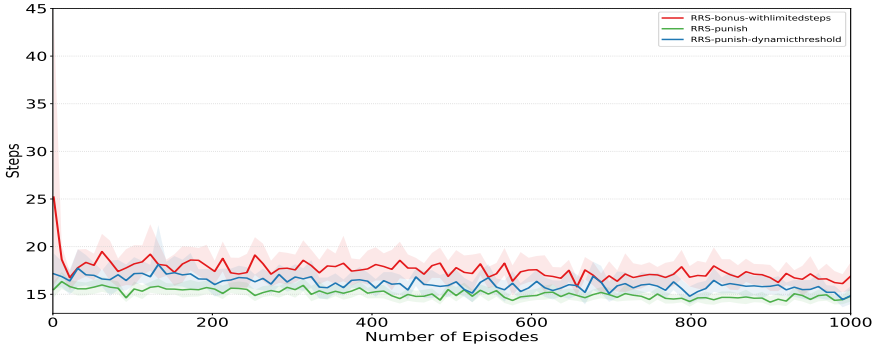


(c) Average steps to reach the goal state under 5000 steps limit.

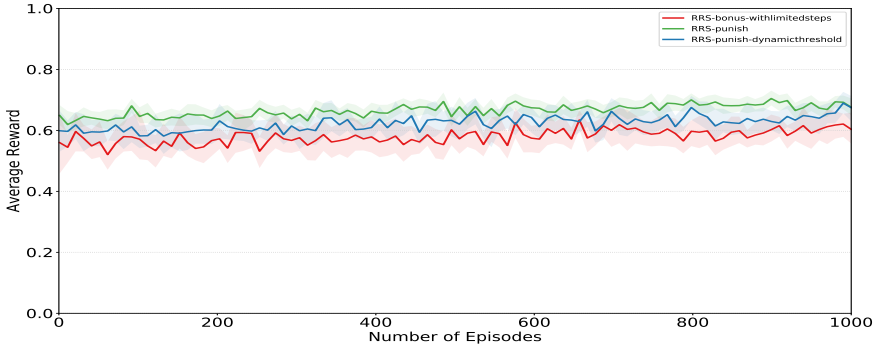


(d) Average reward per episode under 5000 steps limit.

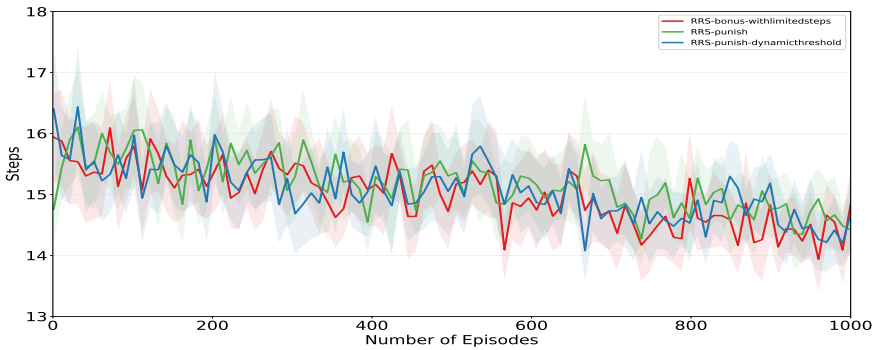
Figure 4.17: Learning performance comparison between punishment-based and bonus-based RRS frameworks in `Six-Rooms` domain.



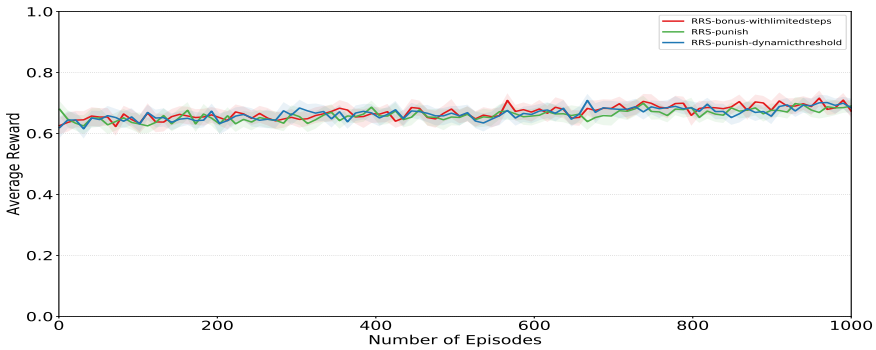
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

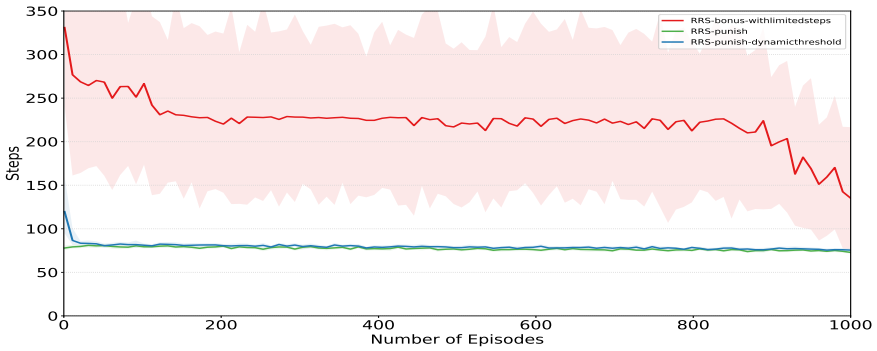


(c) Average steps to reach the goal state under 2000 steps limit.

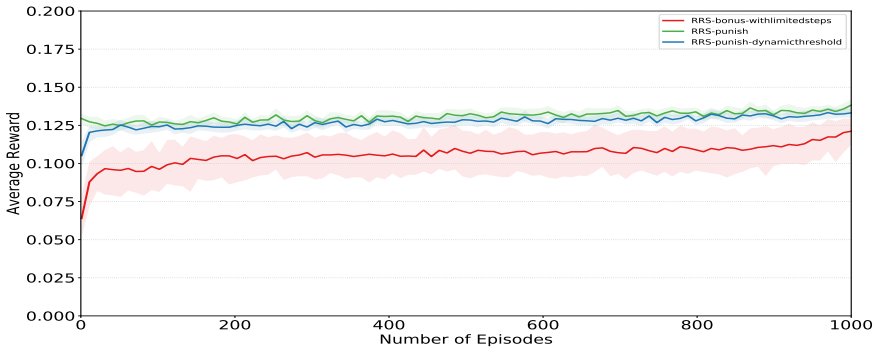


(d) Average reward per episode under 2000 steps limit.

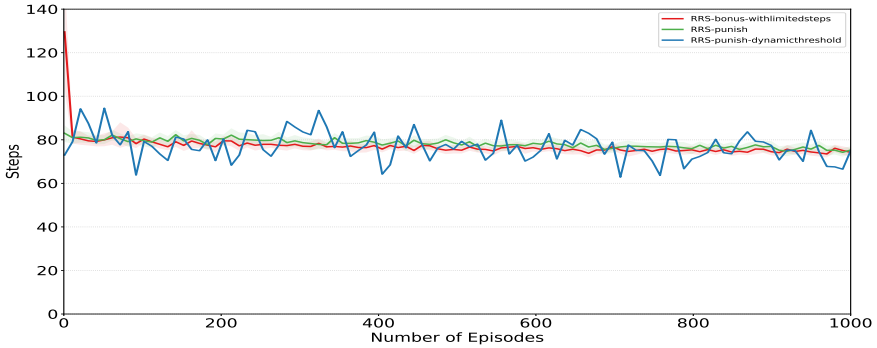
Figure 4.18: Learning performance comparison between punishment-based and bonus-based RRS frameworks in `Six-Rooms Scaled` domain.



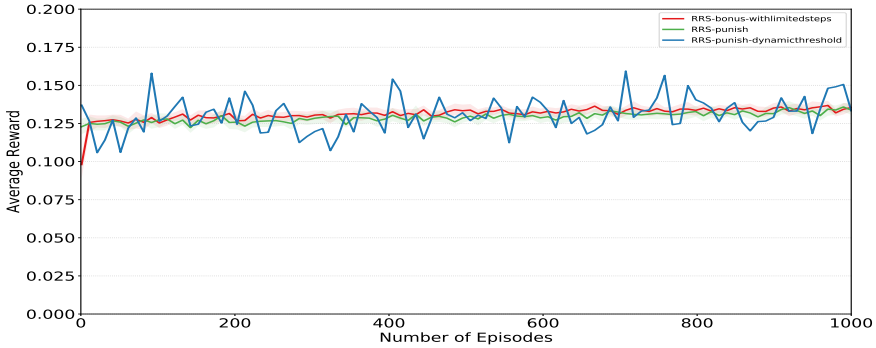
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

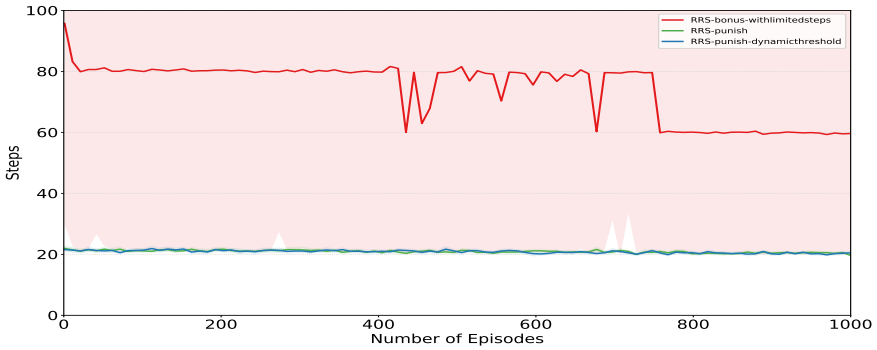


(c) Average steps to reach the goal state under 5000 steps limit.

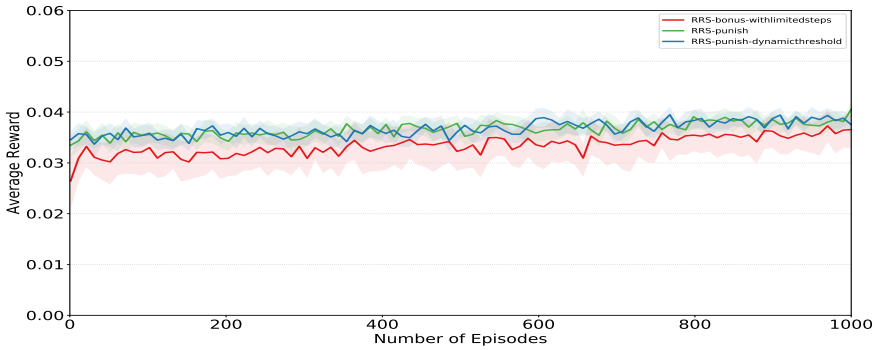


(d) Average reward per episode under 5000 steps limit.

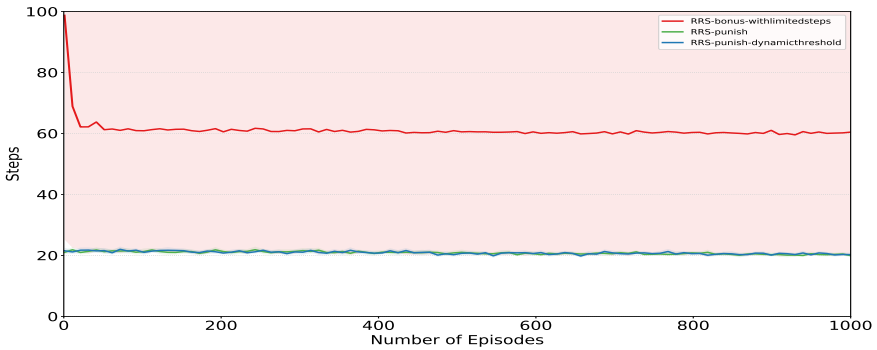
Figure 4.19: Learning performance comparison between punishment-based and bonus-based RRS frameworks in `Zigzag Four-Rooms` domain.



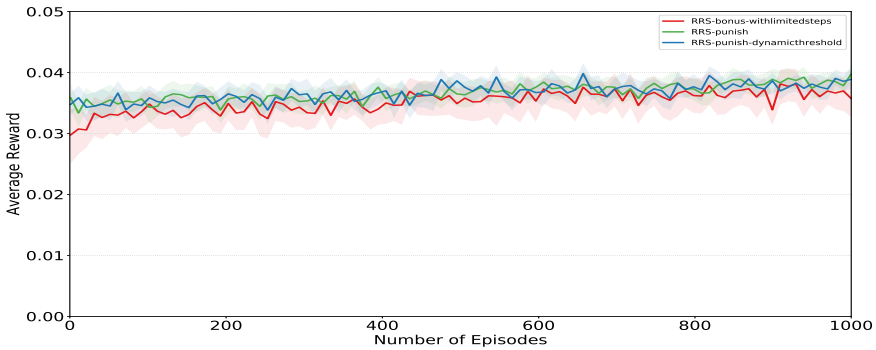
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

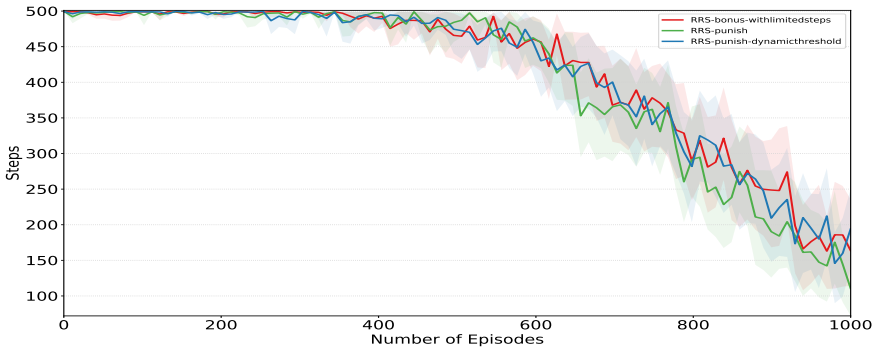


(c) Average steps to reach the goal state under 2000 steps limit.

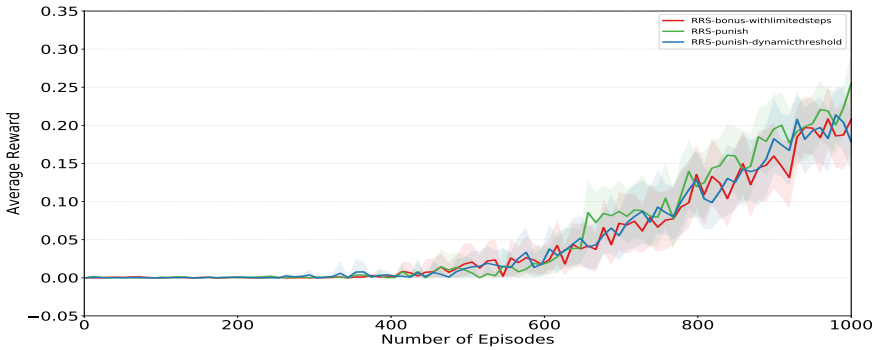


(d) Average reward per episode under 2000 steps limit.

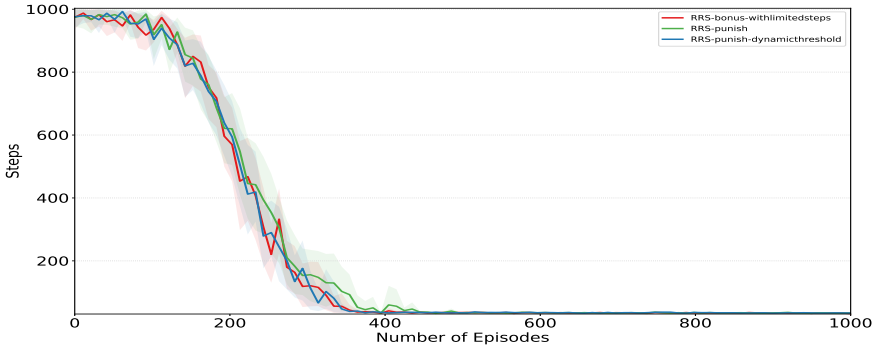
Figure 4.20: Learning performance comparison between punishment-based and bonus-based RRS frameworks in `Zigzag Four Rooms Scaled` domain.



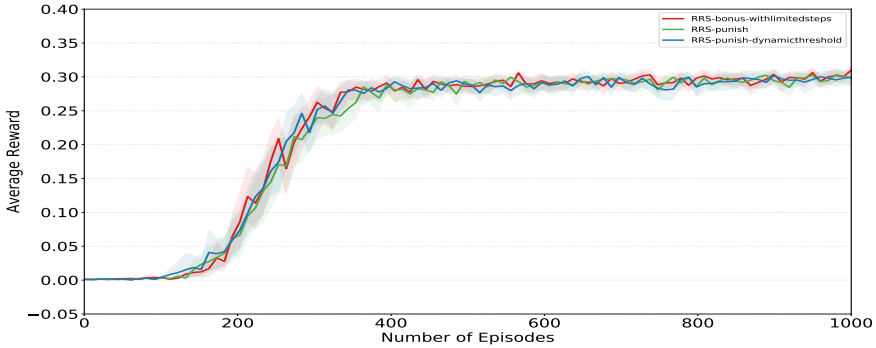
(a) Average steps to reach the goal state under 500 steps limit.



(b) Average reward per episode under 500 steps limit.

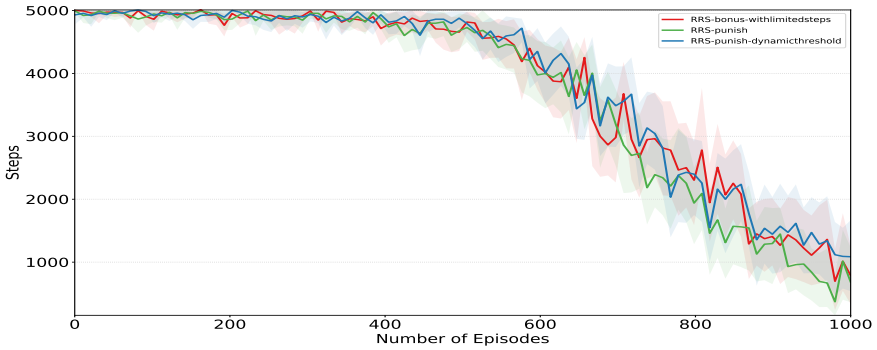


(c) Average steps to reach the goal state under 1000 steps limit.

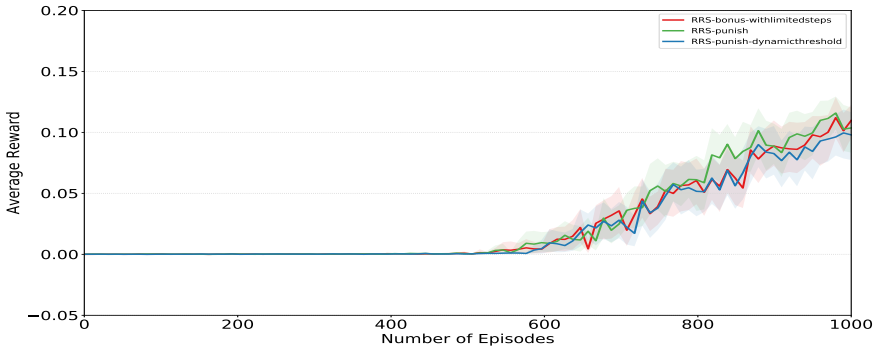


(d) Average reward per episode under 1000 steps limit.

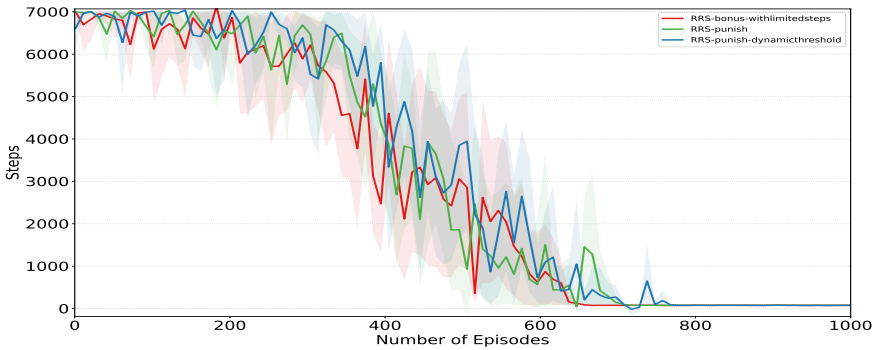
Figure 4.21: Learning performance comparison between punishment-based and bonus-based RRS frameworks in `Tower of Hanoi` domain with 3 disks.



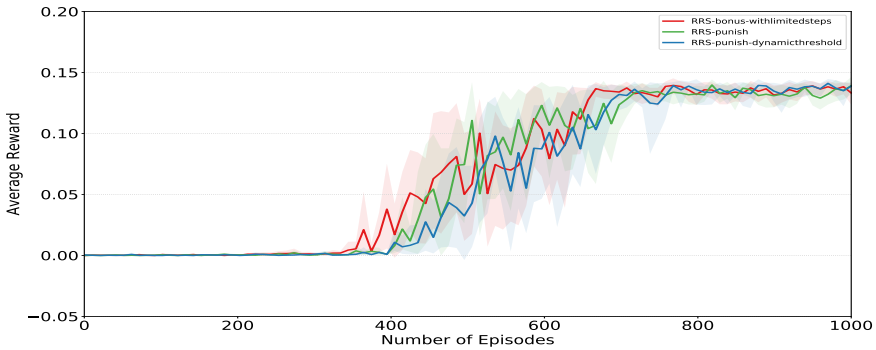
(a) Average steps to reach the goal state under 5000 steps limit.



(b) Average reward per episode under 5000 steps limit.



(c) Average steps to reach the goal state under 7000 steps limit.



(d) Average reward per episode under 7000 steps limit.

Figure 4.22: Learning performance comparison between punishment-based and bonus-based RRS frameworks in `Tower of Hanoi` domain with 4 disks.

From Table 4.6, we observe that our proposed framework *RRS-punish* outperforms all benchmarks in terms of average steps and average reward criteria in all of the GridWorld settings with varying interaction times with the exception of the most complex domain `Zigzag Four-Rooms` with 1000 steps. However, in that setting, the standard deviation of the measures for Sarsa(λ) learner is notably high. Furthermore, learning rates in Figures 4.27 to 4.30 reflect that, RRS-punish starts from a better initial values of the number of steps and average reward and converge fast with the help of training of sub-agents.

In terms of average elapsed time, our method is beaten by Sarsa and Sarsa(λ) learners as expected, due to the large portion of time spent for the training of sub-agents in the initial episode of the learning process. However, if we focus on the learning time of *RRS-Agent*, we see that the agent learns remarkably fast after sub-agents' training is completed. This shows us *population-based repulsive reward-shaping mechanism* indeed provides an informative start for the *RRS-Agent*.

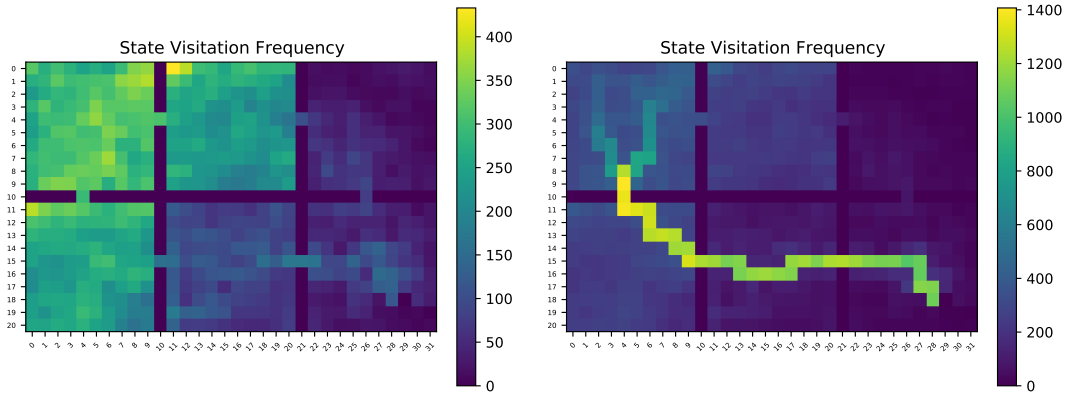
Finally, our method performs better in `ToH` domains regarding converged values of the number of steps and average reward when the interaction time is longer. However, for the settings in which the $steps$ limit becomes more restrictive, Sarsa(λ) gives better results.

The benefit of the exploration with the proposed *RRS* mechanism is sketched in the state-space coverage figures. In the Figures 4.23-4.26, state visitation frequencies are depicted as heat maps. Since GridWorld tasks are suitable for such illustrations, we provide the evaluation for those problem domains only. As can be seen from the figures, almost all states are discovered after the exploration phase i.e. training of sub-agents. This is a promising result since the sub-agents did not stuck at only one room during their interaction time. For instance, *RRS-punish* struggles in `Zigzag Four-Rooms` domain with 1000 steps as (a) of Figure 4.24 shows that sub-agents spent most of their time in the room where they start at. However, for the rest of the domains, the state-space is well-explored. Furthermore, we also show the state visitation frequencies after the *RRS-Agent*'s learning phase is completed in these figures. We can observe that through effective exploration the *RRS-Agent* is able to converge to a policy that helps it to find the goal state.

For the ablation study, it can be observed from the results that coordinated exploration with the repulsive reward shaping mechanism significantly impacts the learning performance. Removing the mechanism results in a poorly performing framework that could not stabilize its policy. Having a hierarchical structure still helps since the learning performance starts with good points for performance measures, however, learning from independent sub-agents is not beneficial for state-action value estimates of *SuperAgent* because of the disruption in the temporal structure of experiences to update *SuperAgent*'s Q-table.

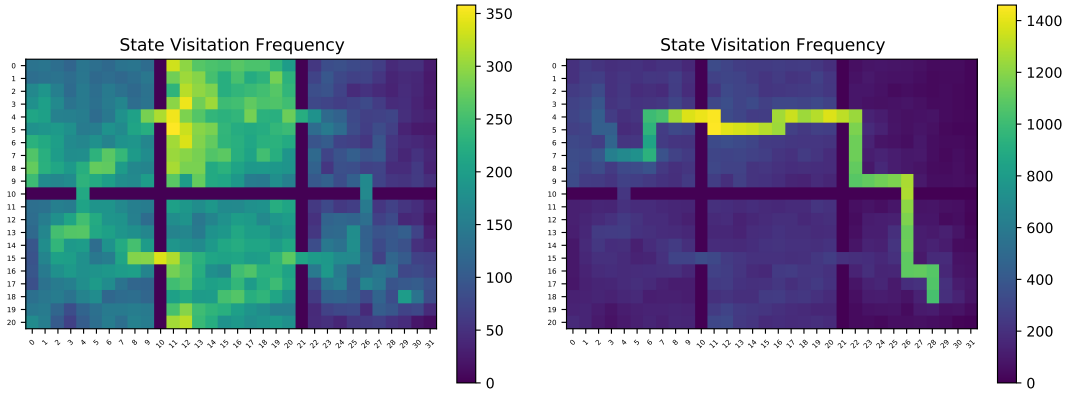
Table 4.6: Overall performance comparison of the proposed method with benchmarks.

Problem	Method	Average Steps (stdev)		Average Reward (stdev)		Average Elapsed	Average Subagents'	Average RRS-Agent
		over all episodes	of the last episode	over all episodes	of the last episode	Time sec (stdev)	Training Time sec (stdev)	Learning Time sec (stdev)
Six-rooms (1000 steps)	RRS-punish	**54.33 (29.33)**	**46.86 (6.31)**	**0.21 (0.02)**	**0.22 (0.03)**	258.91 (48.63)	251.65 (48.42)	7.25 (1.44)
	Sarsa(\lambda)	127.79 (151.22)	67.32 (8.96)	0.13 (0.03)	0.15 (0.02)	254.86 (144.41)	-	-
	Sarsa	984.52 (20.06)	936.20 (224.89)	0.00 (0.00)	0.01 (0.05)	90.73 (8.63)	-	-
	SuperAgent-SubAgents	463.47 (70.32)	518.65 (300.47)	0.04 (0.01)	0.03 (0.03)	691.92 (191.46)	-	-
Six-rooms (5000 steps)	RRS-punish	**52.58 (23.93)**	**46.04 (6.06)**	**0.21 (0.02)**	**0.22 (0.03)**	363.71 (116.45)	356.66 (116.38)	7.05 (0.65)
	Sarsa(\lambda)	86.73 (176.02)	67.06 (11.93)	0.14 (0.01)	0.15 (0.03)	166.42 (70.7)	-	-
	Sarsa	554.61 (1164.67)	47.96 (4.43)	0.17 (0.07)	0.21 (0.02)	48.08 (7.37)	-	-
	SuperAgent-SubAgents	547.86 (124.56)	431.30 (241.17)	0.00 (0.00)	0.00 (0.00)	1015.03 (186.42)	-	-
Six-rooms-scaled (1000 steps)	RRS-punish	**15.83 (6.90)**	**13.98 (1.58)**	**0.65 (0.04)**	**0.71 (0.07)**	10.29 (1.14)	7.99 (1.16)	2.29 (0.07)
	Sarsa(\lambda)	19.68 (17.75)	17.40 (2.65)	0.54 (0.03)	0.58 (0.09)	6.65 (0.71)	-	-
	Sarsa	22.65 (32.48)	14.76 (1.54)	0.61 (0.16)	0.67 (0.08)	1.24 (0.11)	-	-
	SuperAgent-SubAgents	57.70 (16.99)	31.20 (19.10)	0.32 (0.03)	0.38 (0.15)	112.71 (23.55)	-	-
Six-rooms-scaled (2000 steps)	RRS-punish	**15.76 (7.31)**	**13.90 (2.16)**	**0.66 (0.04)**	**0.72 (0.09)**	10.40 (1.26)	8.10 (1.28)	2.30 (0.06)
	Sarsa(\lambda)	18.76 (16.98)	16.84 (3.07)	0.57 (0.04)	0.60 (0.11)	6.3 (0.72)	-	-
	Sarsa	22.69 (31.74)	14.70 (1.47)	0.61 (0.16)	0.68 (0.07)	1.24 (0.11)	-	-
	SuperAgent-SubAgents	39.35 (12.85)	27.85 (13.45)	0.37 (0.04)	0.38 (0.10)	213.76 (43.34)	-	-
Zigzag-four-rooms (1000 steps)	RRS-punish	991.29 (13.44)	953.24 (189.65)	0.00 (0.00)	0.01 (0.03)	501.03 (18.26)	372.95 (16.10)	128.07 (4.83)
	Sarsa(\lambda)	**549.64 (223.06)**	**290.32 (376.45)**	**0.05 (0.03)**	**0.09 (0.05)**	1097.76 (668.88)	-	-
	Sarsa	998.49 (1.84)	999.00 (0.00)	0.00 (0.00)	0.00 (0.00)	89.16 (8.16)	-	-
	SuperAgent-SubAgents	742.07 (55.10)	671.35 (371.10)	0.02 (0.00)	0.02 (0.03)	1211.71 (566.75)	-	-
Zigzag-four-rooms (5000 steps)	RRS-punish	**96.69 (110.38)**	**72.74 (5.03)**	**0.13 (0.01)**	**0.14 (0.01)**	834.0 (214.23)	821.28 (214.68)	12.72 (3.31)
	Sarsa(\lambda)	126.53 (277.07)	88.78 (10.52)	0.10 (0.01)	0.11 (0.01)	235.43 (114.61)	-	-
	Sarsa	1193.06 (1764.87)	74.46 (6.61)	0.09 (0.06)	**0.14 (0.01)**	106.3 (14.15)	-	-
	SuperAgent-SubAgents	1055.99 (293.00)	1485.95 (1504.88)	0.03 (0.01)	0.02 (0.03)	1395.57 (521.14)	-	-
Zigzag-four-rooms-scaled (1000 steps)	RRS-punish	**22.17 (7.23)**	**20.00 (1.75)**	**0.04 (0.00)**	**0.04 (0.01)**	16.17 (5.48)	13.05 (5.51)	3.12 (0.11)
	Sarsa(\lambda)	33.72 (59.34)	21.34 (2.32)	0.03 (0.01)	**0.04 (0.01)**	9.93 (4.77)	-	-
	Sarsa	31.70 (36.53)	20.52 (1.75)	0.03 (0.01)	**0.04 (0.00)**	1.74 (0.12)	-	-
	SuperAgent-SubAgents	29.21 (8.11)	22.48 (2.15)	0.03 (0.00)	0.03 (0.01)	39.51 (10.37)	-	-
Zigzag-four-rooms-scaled (2000 steps)	RRS-punish	**22.67 (11.47)**	**20.22 (2.25)**	**0.04 (0.00)**	**0.04 (0.01)**	15.98 (6.65)	12.79 (6.70)	3.19 (0.16)
	Sarsa(\lambda)	28.24 (49.31)	20.92 (2.32)	0.03 (0.00)	**0.04 (0.01)**	8.58 (3.22)	-	-
	Sarsa	31.65 (36.46)	20.58 (1.78)	0.03 (0.01)	**0.04 (0.01)**	1.66 (0.09)	-	-
	SuperAgent-SubAgents	33.64 (16.02)	22.80 (2.29)	0.03 (0.00)	0.03 (0.01)	37.42 (9.43)	-	-
ToH-3x3 (500 steps)	RRS-punish	430.92 (97.18)	182.80 (207.29)	0.04 (0.06)	0.19 (0.14)	65.07 (6.2)	9.67 (1.43)	55.39 (6.27)
	Sarsa(\lambda)	**101.17 (102.88)**	**44.36 (12.94)**	**0.21 (0.06)**	**0.24 (0.07)**	75.53 (47.45)	-	-
	Sarsa	411.68 (108.93)	157.42 (196.68)	0.05 (0.07)	0.21 (0.13)	36.25 (5.2)	-	-
	SuperAgent-SubAgents	477.27 (28.72)	444.02 (144.16)	0.01 (0.02)	0.03 (0.09)	173 (15.86)	-	-
ToH-3x3 (1000 steps)	RRS-punish	250.87 (349.17)	**34.02 (5.79)**	0.22 (0.11)	**0.31 (0.04)**	53.73 (6.51)	21.54 (3.63)	32.19 (5.33)
	Sarsa(\lambda)	**59.66 (90.98)**	41.42 (11.35)	**0.24 (0.03)**	0.26 (0.06)	43.09 (17.26)	-	-
	Sarsa	241.44 (347.19)	32.94 (3.70)	0.22 (0.11)	0.30 (0.03)	21.48 (4.21)	-	-
	SuperAgent-SubAgents	630.30 (190.09)	611.20 (459.50)	0.11 (0.06)	0.12 (0.15)	329.13 (31.39)	-	-
ToH-4x3 (5000 steps)	RRS-punish	3739.23 (1490.55)	684.54 (1338.11)	0.03 (0.04)	0.10 (0.06)	5412.95 (578.28)	4902.95 (543.74)	510.00 (93.81)
	Sarsa(\lambda)	**274.40 (707.27)**	**98.90 (23.24)**	**0.10 (0.02)**	**0.11 (0.02)**	586.28 (201.10)	-	-
	Sarsa	3723.07 (1432.67)	844.22 (1688.91)	0.03 (0.04)	0.11 (0.06)	356.00 (50.86)	-	-
	SuperAgent-SubAgents	3604.95 (1501.06)	851.68 (1628.02)	0.03 (0.04)	0.11 (0.06)	1886.73 (194.79)	-	-
ToH-4x3 (7000 steps)	RRS-punish	3016.79 (2859.61)	**72.30 (6.21)**	0.07 (0.06)	**0.14 (0.01)**	23990.58 (3469.57)	23567.3 (3467.75)	423.28 (35.56)
	Sarsa(\lambda)	**192.04 (585.59)**	102.50 (25.06)	**0.10 (0.01)**	0.10 (0.03)	271.67 (105.70)	-	-
	Sarsa	3207.17 (2813.45)	78.70 (11.70)	0.06 (0.05)	0.13 (0.02)	304.26 (33.40)	-	-
	SuperAgent-SubAgents	5413.58 (1574.61)	4921.50 (3173.43)	0.02 (0.03)	0.04 (0.06)	2831.06 (24.14)	-	-



(a) After the exploration phase (training of subagents) under 1000 steps limit.

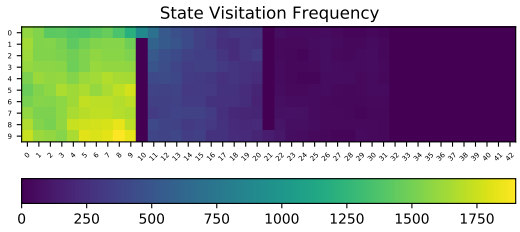
(b) After the *RRS-Agent*'s learning phase under 1000 steps limit.



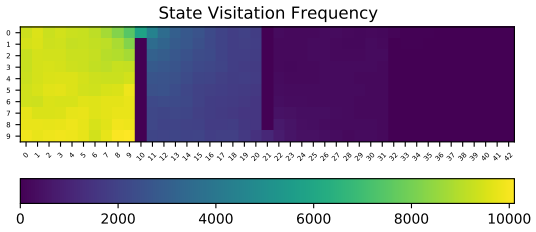
(c) After the exploration phase (training of subagents) under 5000 steps limit.

(d) After the *RRS-Agent*'s learning phase under 5000 steps limit.

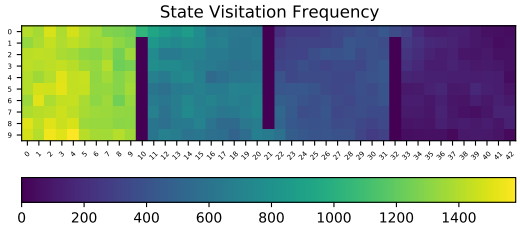
Figure 4.23: State space coverage of the proposed method for `Six-Rooms GridWorld` domain under 1000 and 5000 steps limit.



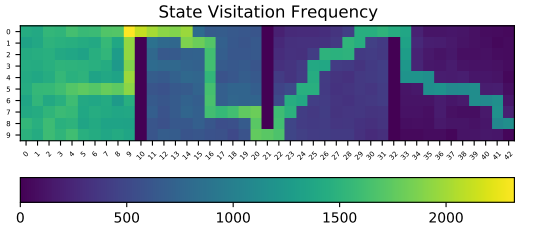
(a) After the exploration phase (training of subagents) under 1000 steps limit.



(b) After the *RRS-Agent*'s learning phase under 1000 steps limit.

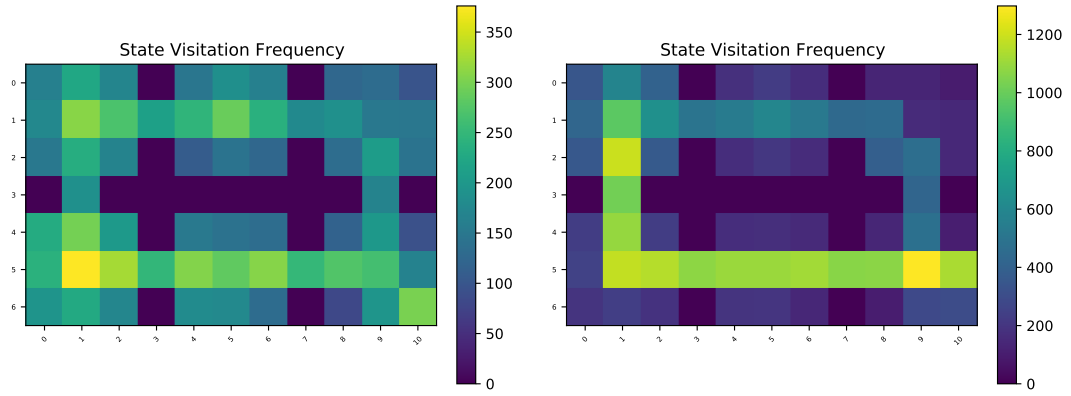


(c) After the exploration phase (training of subagents) under 5000 steps limit.



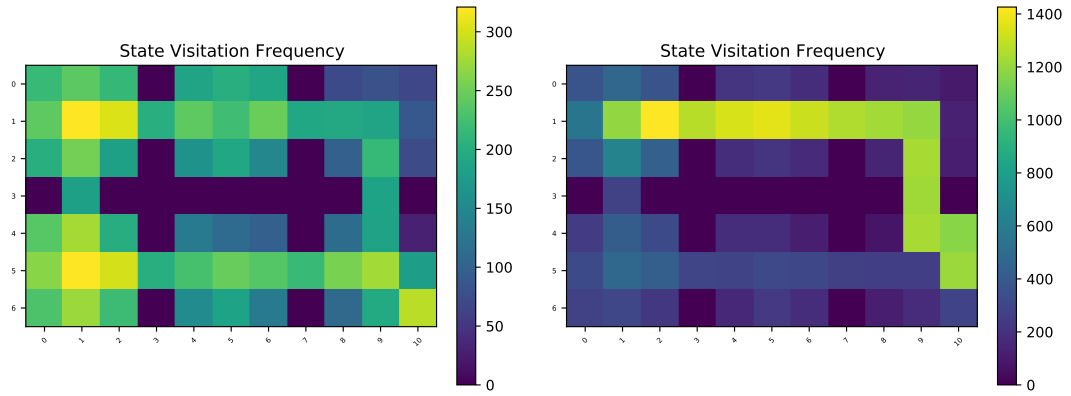
(d) After the *RRS-Agent*'s learning phase under 5000 steps limit.

Figure 4.24: State space coverage of the proposed method for `Zigzag Four Rooms GridWorld` domain under 1000 and 5000 steps limit.



(a) After the exploration phase (training of subagents) under 1000 steps limit.

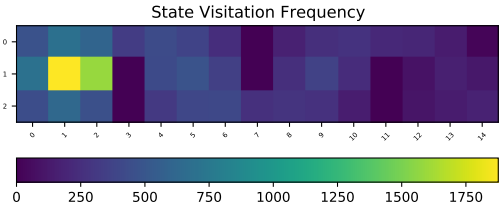
(b) After the *RRS-Agent*'s learning phase under 1000 steps limit.



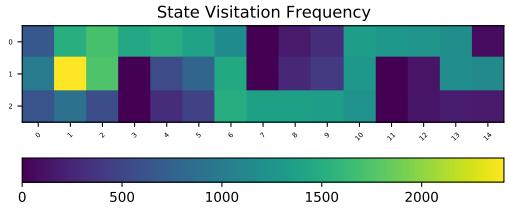
(c) After the exploration phase (training of subagents) under 2000 steps limit.

(d) After the *RRS-Agent*'s learning phase under 2000 steps limit.

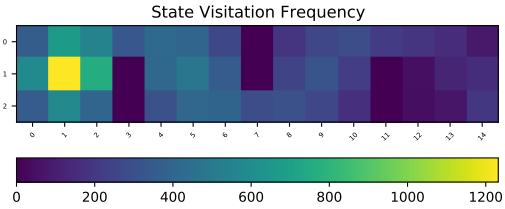
Figure 4.25: State space coverage of the proposed method for `Six-Rooms Scaled GridWorld` domain under 1000 and 2000 steps limit.



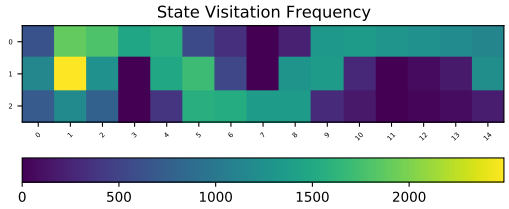
(a) After the exploration phase (training of subagents) under 1000 steps limit.



(b) After the *RRS-Agent*'s learning phase under 1000 steps limit.

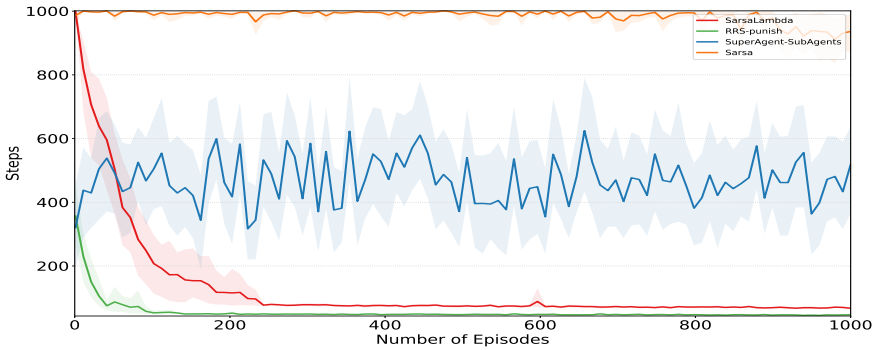


(c) After the exploration phase (training of subagents) under 1000 steps limit.

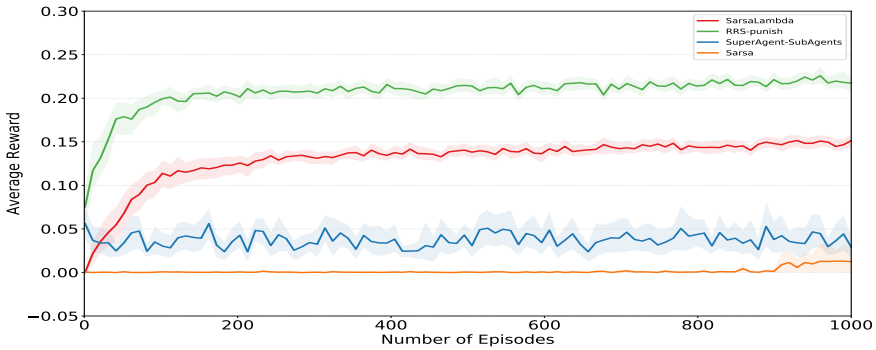


(d) After the *RRS-Agent*'s learning phase under 1000 steps limit.

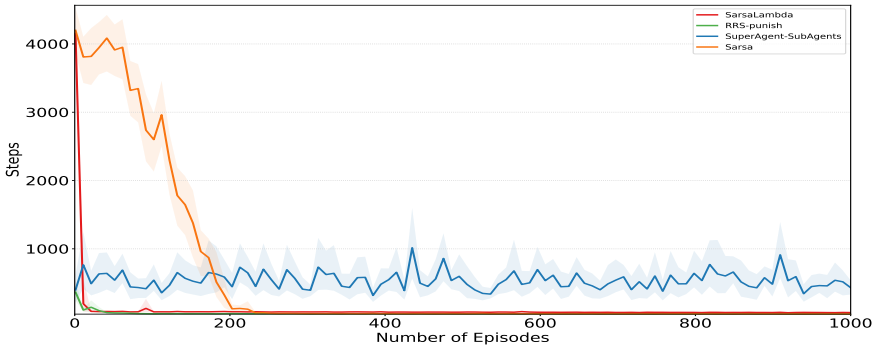
Figure 4.26: State space coverage of the proposed method for `Zigzag Four Rooms Scaled GridWorld` domain under 1000 and 2000 steps limit.



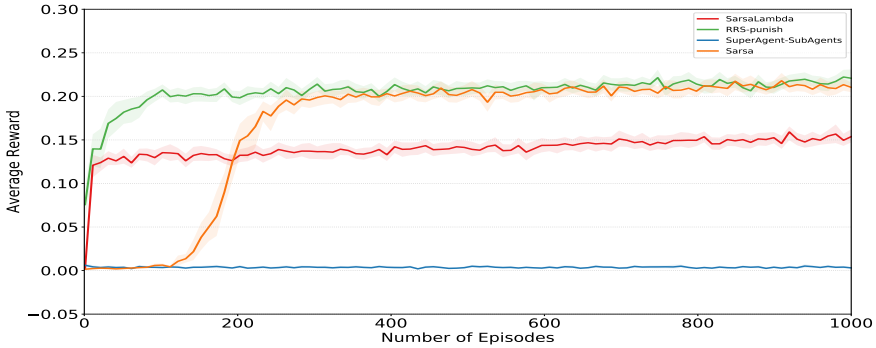
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

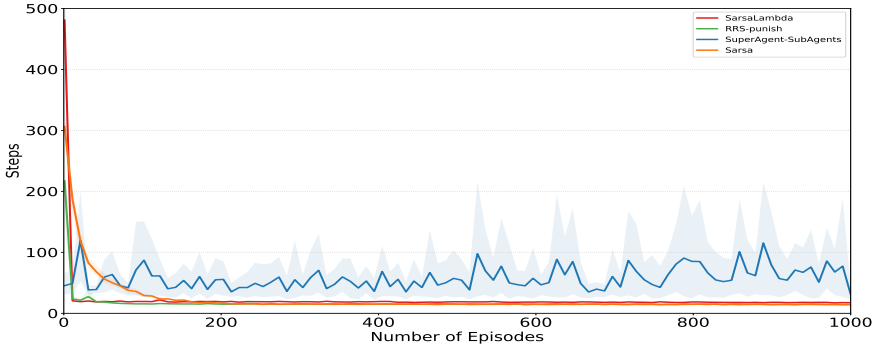


(c) Average steps to reach the goal state under 5000 steps limit.

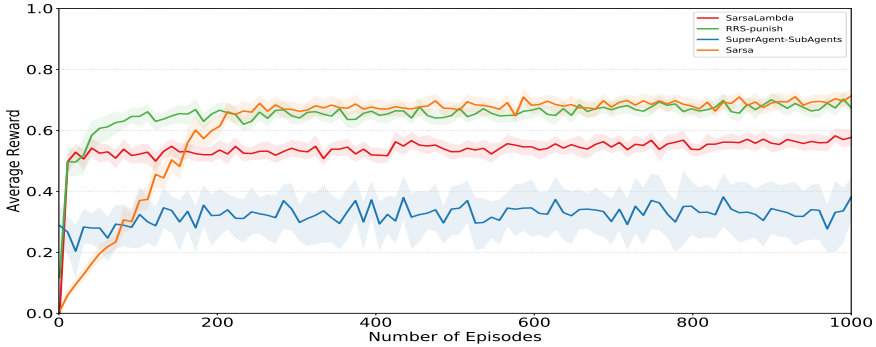


(d) Average reward per episode under 5000 steps limit.

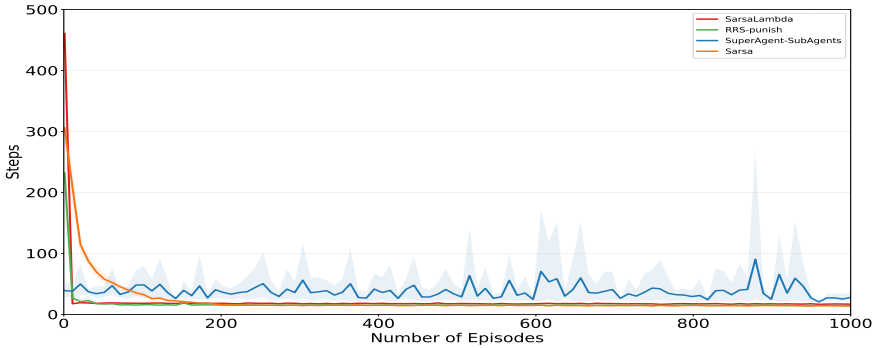
Figure 4.27: Learning performances of the proposed method and benchmarks for `Six-Rooms` domain under 1000 and 5000 steps limit.



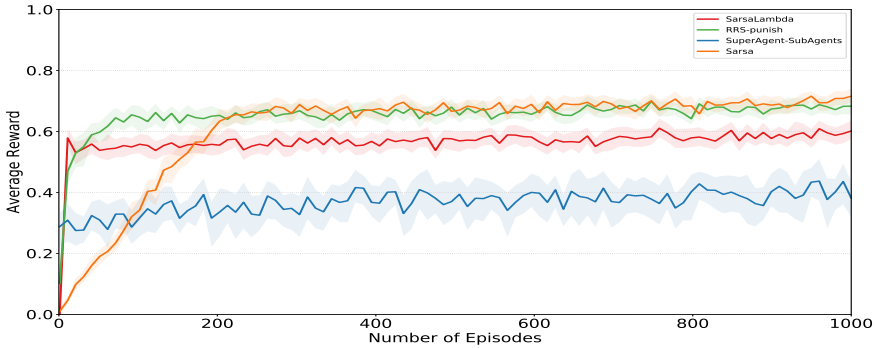
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

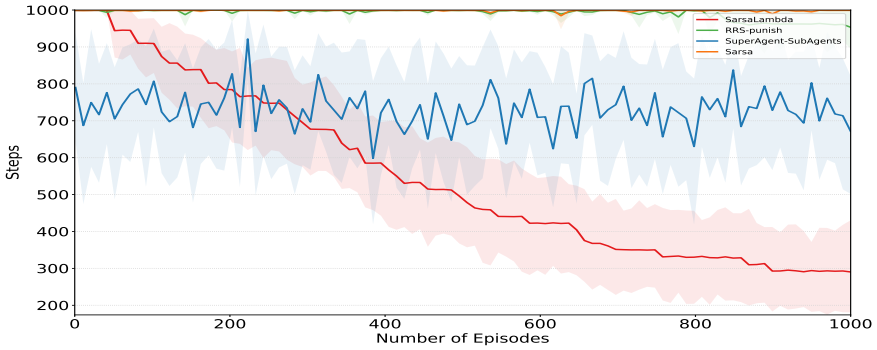


(c) Average steps to reach the goal state under 2000 steps limit.

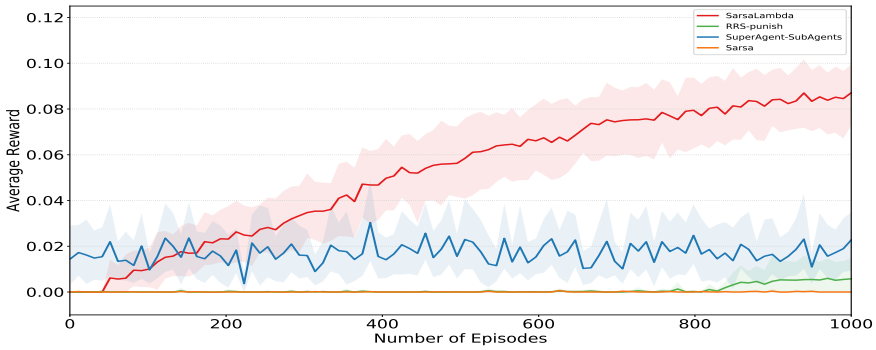


(d) Average reward per episode under 2000 steps limit.

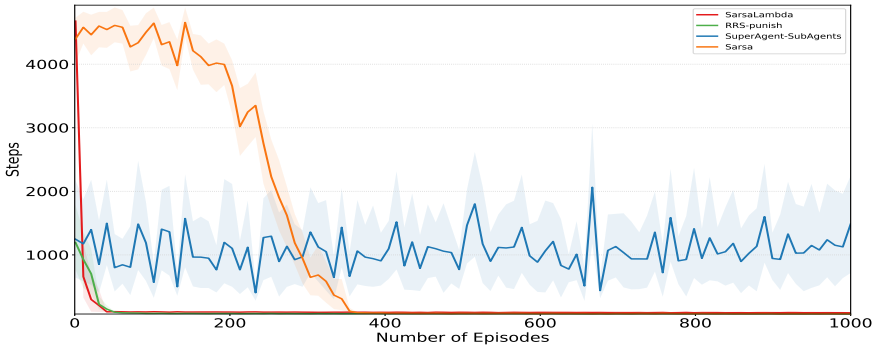
Figure 4.28: Learning performances of the proposed method and benchmarks for `Six-Rooms Scaled` domain under 1000 and 2000 steps limit.



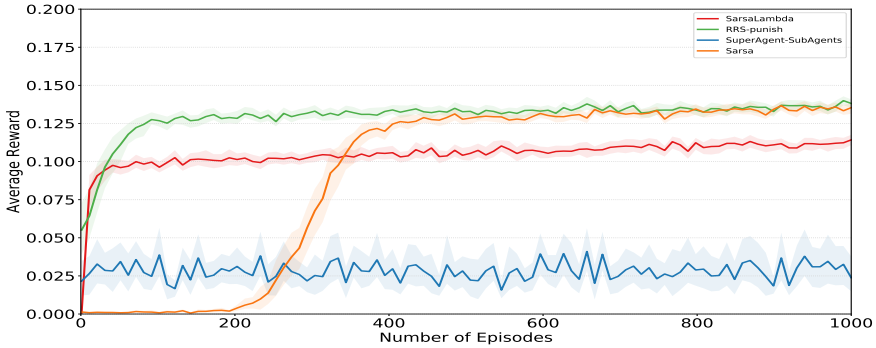
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

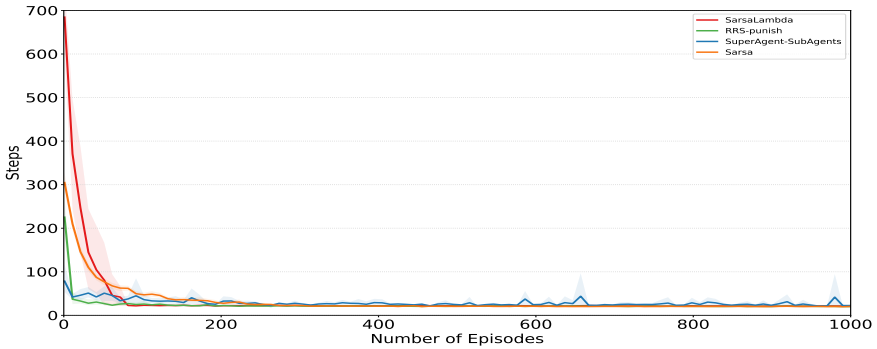


(c) Average steps to reach the goal state under 5000 steps limit.

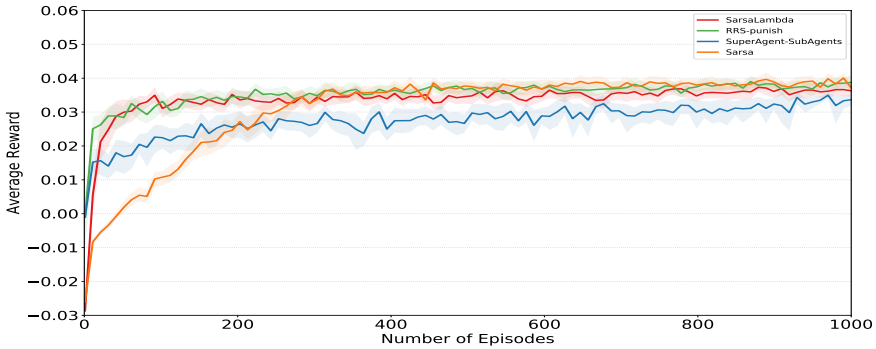


(d) Average reward per episode under 5000 steps limit.

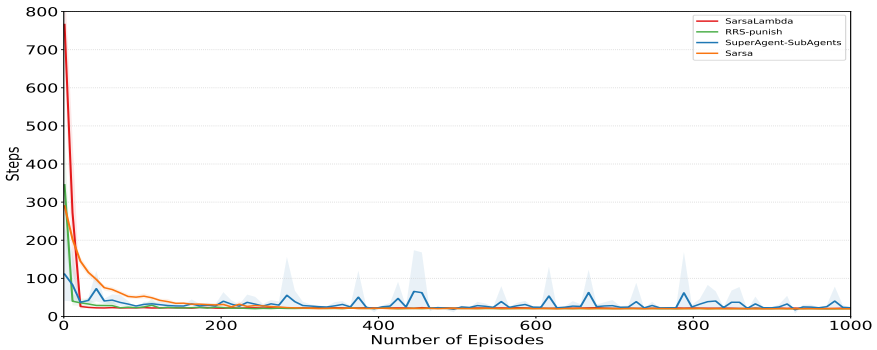
Figure 4.29: Learning performances of the proposed method and benchmarks for `Zigzag Four-Rooms` domain under 1000 and 5000 steps limit.



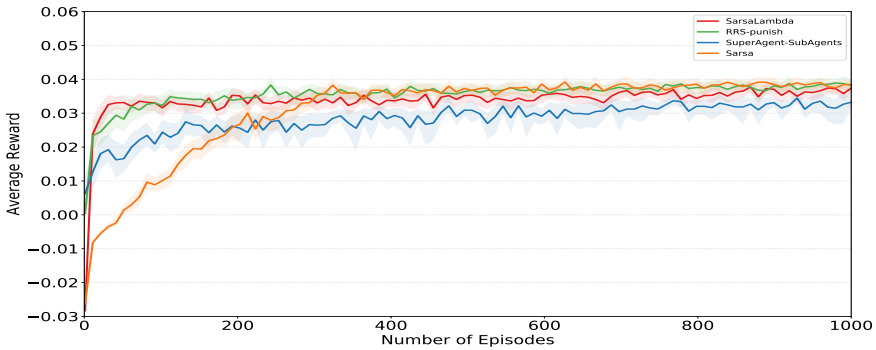
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

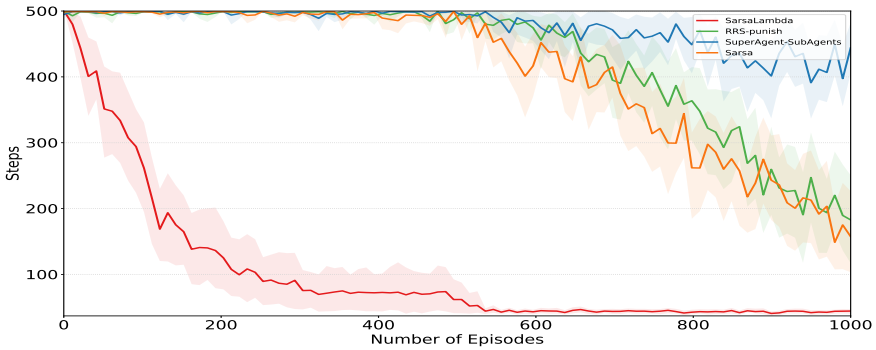


(c) Average steps to reach the goal state under 2000 steps limit.

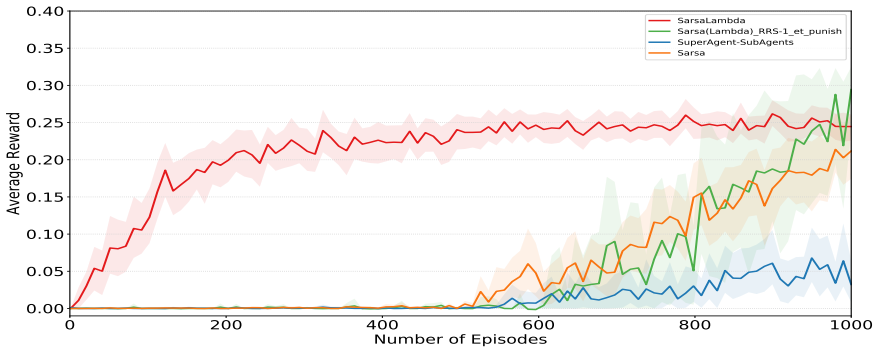


(d) Average reward per episode under 2000 steps limit.

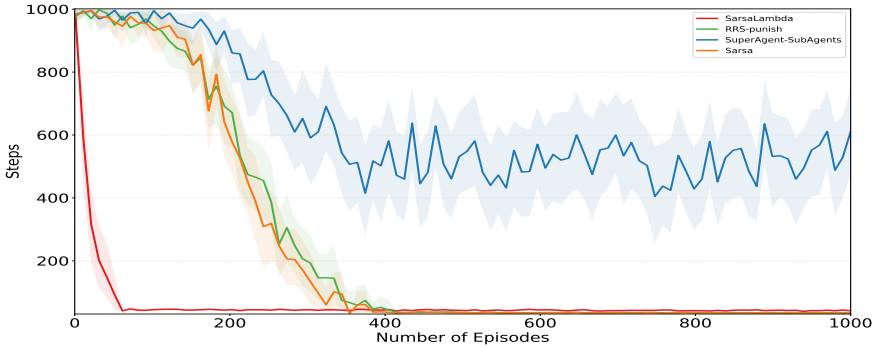
Figure 4.30: Learning performances of the proposed method and benchmarks for `Zigzag Four-Rooms Scaled` domain under 1000 and 2000 steps limit.



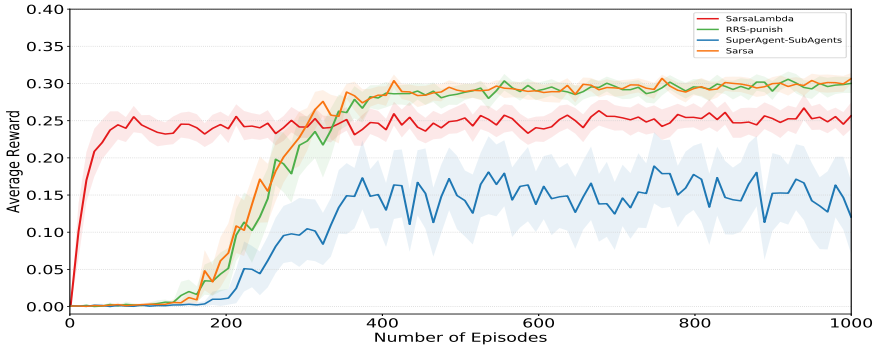
(a) Average steps to reach the goal state in 3 disks version under 500 steps limit.



(b) Average reward per episode in 3 disks version under 500 steps limit.

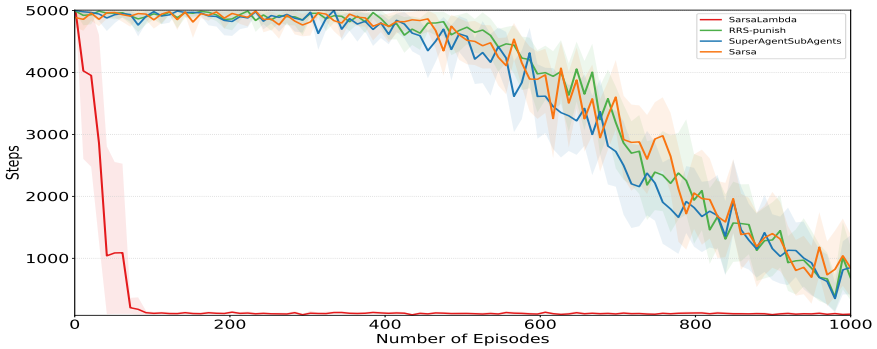


(c) Average steps to reach the goal state in 3 disks version under 1000 steps limit.

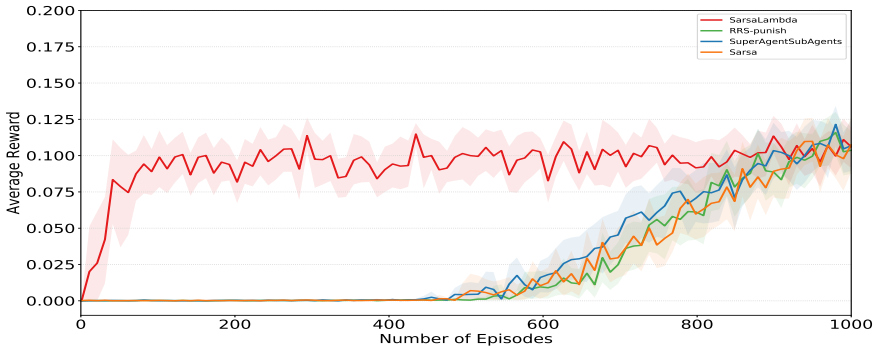


(d) Average reward per episode in 3 disks version under 1000 steps limit.

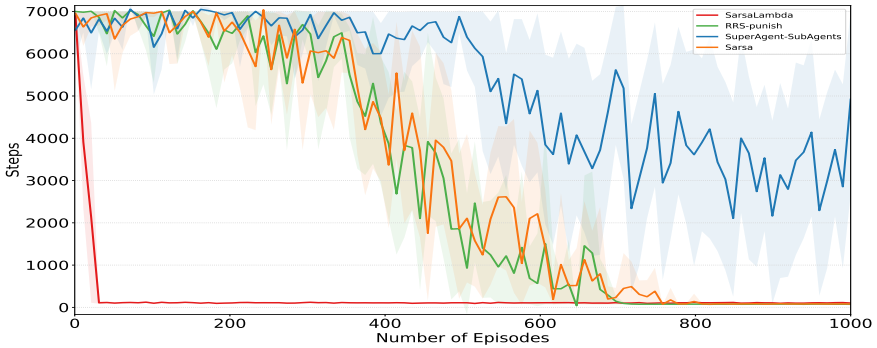
Figure 4.31: Learning performances of the proposed method and benchmarks for `Tower of Hanoi` domain under 3 rods and 3 disks version.



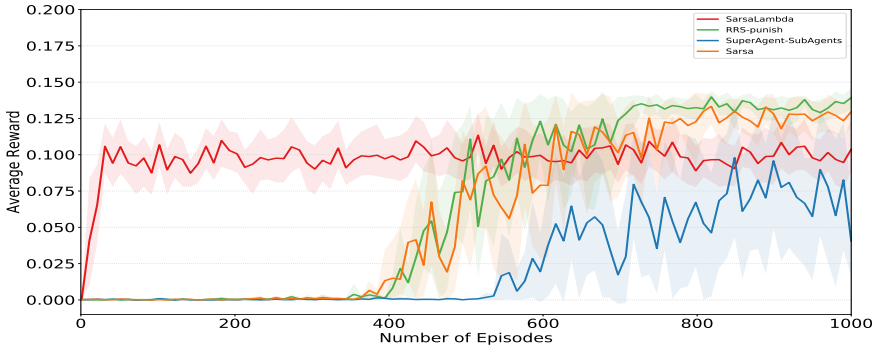
(a) Average steps to reach the goal state in 3 disks version under 5000 steps limit.



(b) Average reward per episode in 3 disks version under 5000 steps limit.



(c) Average steps to reach the goal state in 3 disks version under 7000 steps limit.



(d) Average reward per episode in 3 disks version under 7000 steps limit.

Figure 4.32: Learning performances of the proposed method and benchmarks for `Tower of Hanoi` domain under 3 rods and 4 disks version.

CHAPTER 5

IMPROVING LEARNING EFFICIENCY BY POTENTIAL-BASED REWARD SHAPING USING STATE-SPACE SEGMENTATION WITH THE EXTENDED SEGMENTED Q-CUT ALGORITHM

This chapter explains the motivation for learning efficiency problem in RL literature and presents the proposed method which aims to solve this problem. The Section 5.2 introduces our method called *potential-based reward shaping using state-space segmentation with Extended Segmented Q-Cut (ESegQ-Cut) algorithm* and Section 5.2.1 elaborates the process of segmentation of the state-space whilst Section 5.2.2 explains the potential-based reward shaping with state-space segment information. Finally, the computational study along with the results and discussions are presented in the Section 5.3.

5.1 Problem Motivation

Sparse and/or delayed reward environments have been a major challenge for the RL approaches. Although RL framework has gained recent success in such domains, it still suffers from slow learning. Particularly, when it is applied in the real-world problems, the required time for learning is exacerbated due to large state & action spaces. An effective direction to deal with this problem is to decompose the complex task into sub-tasks so that each simpler sub-task can be learned in parallel to reduce the learning time. Another direction is to utilize the domain knowledge extracted during the agent-environment interaction into agent's learning process. Reward shaping is one of the ways that aims to use the extracted knowledge to improve the speed of the learning process. Since reward signals are the essential part of the efficiency of

the RL approaches, we attack the slow learning problem in terms of rewards. To this end, our proposed method introduces a potential-based reward shaping mechanism that depends on the segmentation of the state-space computed with the ESegQ-Cut algorithm.

5.2 Reward Shaping Based on State-Space Segmentation with the Extended Segmented Q-Cut Algorithm

The proposed potential-based reward shaping method relies on the state-space segmentation information extracted by the ESegQ-Cut algorithm. The aim of the method is to use the shaped rewards in the learning process of the RL agent which are computed using potential-based reward shaping function that depends on the segment information of the states. The state-space segmentation is obtained via translating the experiences of the agent in the environment into a transition graph and then applying ESegQ-Cut algorithm. The method is modeled as a nested structure as shown in the in the Figure 5.1.

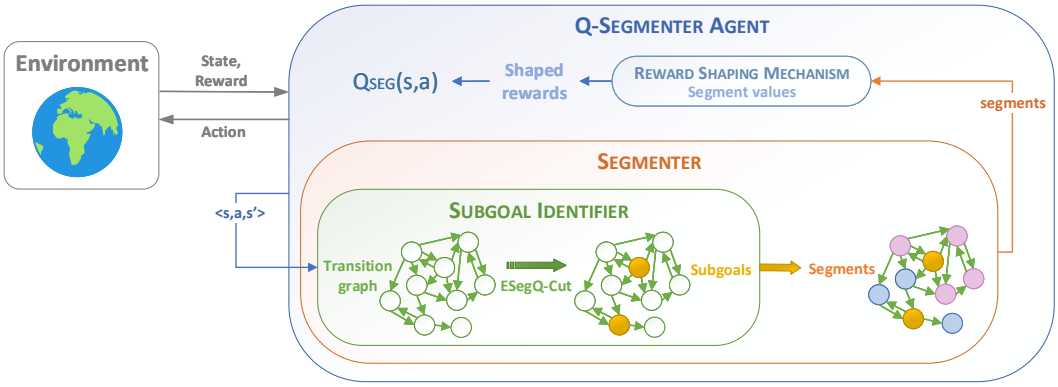


Figure 5.1: The model of the proposed method.

In particular, the model consists of a *Q-Segmenter* agent designed with a nested structure. The *Q-Segmenter* agent behaves as the learning component in this model. It interacts with the environment and approximates the state-action values with a Q-table named as Q_{SEG} by applying Q-Learning algorithm. *Q-Segmenter* agent derives the additional information on the environment through its *Segmenter* and *Subgoal-Identifier* inner components. This additional information is utilized in the agent's

learning process via reward shaping mechanism. After each transition, *Q-Segmenter* agent uses shaped rewards when updating its state-action value estimates instead of directly using extrinsic environmental reward signals. The agent computes the shaped rewards using the information on the state-space segmentation.

Q-Segmenter agent has a *Segmenter* component whose main role is to compute the state-space segments using the subgoals identified by its inner component called *Subgoal-Identifier* and then conveying the extracted segments information to the *Q-Segmenter* agent. Moreover, *Segmenter* behaves as a bridge between *Q-Segmenter* agent and *Subgoal-Identifier* component by transferring the transition experience of *Q-Segmenter* denoted as a tuple $\langle s, a, s' \rangle$ to *Subgoal-Identifier* component.

The innermost element of this model is *Subgoal-Identifier* whose main task is to first translate the transition experiences of *Q-Segmenter* into a transition graph up to a specific period and then, extract the subgoals by performing ESegQ-Cut algorithm on the final transition graph. The tasks of this component also include transferring the identified subgoals information to outer component *Segmenter*.

The pseudocode for the method is given in the Algorithm 11. The algorithm takes the necessary inputs for applying the Q-Learning algorithm along with the inputs related to extract state-space segments information such as number of episodes for performing random walk and cut-quality threshold. The method starts with initializing Q-table Q_{SEG} for each state and action, required lists to hold segments and the value of the segments information and then setting episode number to zero. At the beginning of the initial episode, before learning of the *Q-Segmenter* agent starts, *Q-Segmenter* completes a random walk phase in the environment via `randomWalk()` method for the sake of identifying state-space segments. The agent obtains the state-space segments after completing the random walk, and computes the values of the segments via `getSegmentValues()` method if any segment is identified. With the knowledge on segments and their values, *Q-Segmenter* agent starts the learning phase by applying Q-Learning algorithm. Compared to the standard Q-Learning algorithm given in Algorithm , *Q-Segmenter* agent shapes its environmental reward signal r after taking action a in state s. The shaped reward signal is denoted by $\tilde{r}$ and computed with the method `shapeReward()`. *Q-Segmenter* then uses $\tilde{r}$ in the

update rule for its Q-table Q_{SEG}. Furthermore, the agent also updates the segment values with `updateSegmentValues()` method after each episode.

The flowchart of the learning process with potential-based reward shaping using state-space segmentation with ESegQ-Cut algorithm is presented in the Figure 5.2. As a summary, the learning process starts with agent random walking in the environment for a specific period and accumulating the transitions on a transition graph. After random walk is completed, ESegQ-Cut algorithm is applied on the transition graph and subgoal states are identified. Using the identified subgoals, state-space segments are constructed. After random walk subprocedure is completed in the learning process, training of *Q-Segmenter* agent starts. The agent interacts with the environment shapes the environmental reward signal using the extracted knowledge on state-space segments. It then updates its Q-value estimates with Q-Learning update rule. At the end of each episode, the agent updates its knowledge on state-space segments in terms of their value which will be utilized in the following episode. Training of Q-Segmenter is completed after agent performs pre-determined number of episodes. The details on the methods are given in the following sections.

5.2.1 State-Space Segmentation

5.2.1.1 Random Walk

In the initial episode of the learning, *Q-Segmenter* goes through random walk phase in the environment to identify state-space segments. The general idea of this phase is illustrated in the Figure 5.3. The phase starts with the interaction between *Q-Segmenter* and the environment. Through random walk, the agent generates trajectories and collected trajectories from all episodes of the random walk phase are transformed into a transition graph. The transition graph is then given to Extended Segmented Q-Cut algorithm. At this stage, first Segmented Q-Cut algorithm is applied and subgoal states are determined in the graph, then using these subgoals, the segments of the state-space are extracted. Random walk phase is finalized when the identified segments information is returned to the learning phase.

The random walk explained in the Algorithm 12 continues for M_{cut} many episodes.

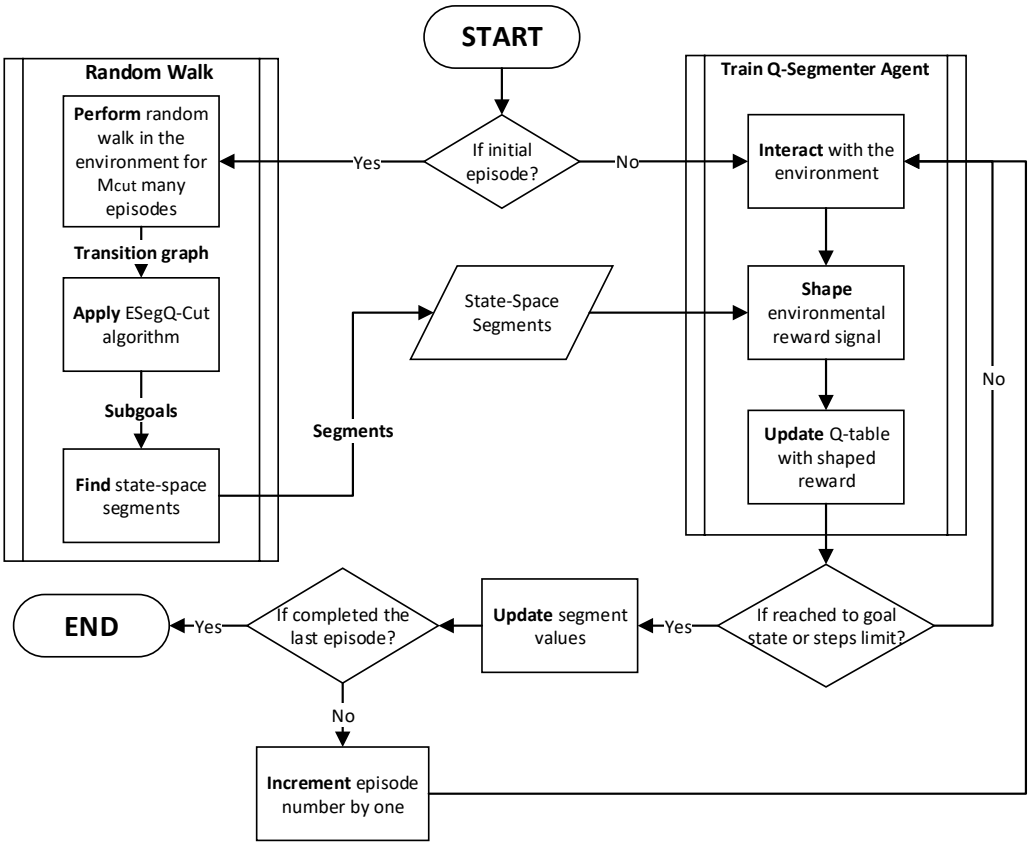


Figure 5.2: The flowchart of the learning process with the proposed method.

Algorithm 11: Learning with Potential-based Reward Shaping Using State-Space Segmentation with the Extended Segmented Q-Cut Algorithm

Input : $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$

Learning rate $\alpha \in (0, 1]$

Exploration rate $\varepsilon \in (0, 1]$

Discount factor $\gamma \in (0, 1]$

Step limit $steps \geq 1$

Number of episodes $M \geq 1$

Number of episodes for random walk $M_{cut} \geq 1$

Cut quality threshold $c_q \geq 1$

Output: Q_{SEG}

Initialization:

$Q_{SEG}(s, a) = 0, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$

$segments \leftarrow [\,], segments\_values \leftarrow [\,], episode = 0$

for $episode = 0$ **to** M **do**

  if *episode = 0* **then**

    $segments \leftarrow$ `randomWalk`$(steps, M_{cut}, c_q, segments)$

    if $segments$ *is not empty* **then**

      $segments\_values \leftarrow$ `getSegmentValues()`

    end if

  end if

  Initialize $s \in \mathcal{S}$

  while s *is not terminal or* $steps$ *is not reached* **do**

    Choose $a \leftarrow$ `EPSILON-GREEDY`(Q_{SEG}, ε)

    Take action a, observe r, s'

    $\tilde{r} \leftarrow$ `shapeReward`$(r, segments, segments\_values)$

    $\delta \leftarrow \tilde{r} + \gamma \max_{a'} Q_{SEG}(s', a') - Q_{SEG}(s, a)$

    Update $Q_{SEG}(s, a) \leftarrow Q_{SEG}(s, a) + \alpha\delta$

    $s \leftarrow s'$

  end while

  Update $segments\_values \leftarrow$ `updateSegmentValues()`

end for

return Q_{SEG}

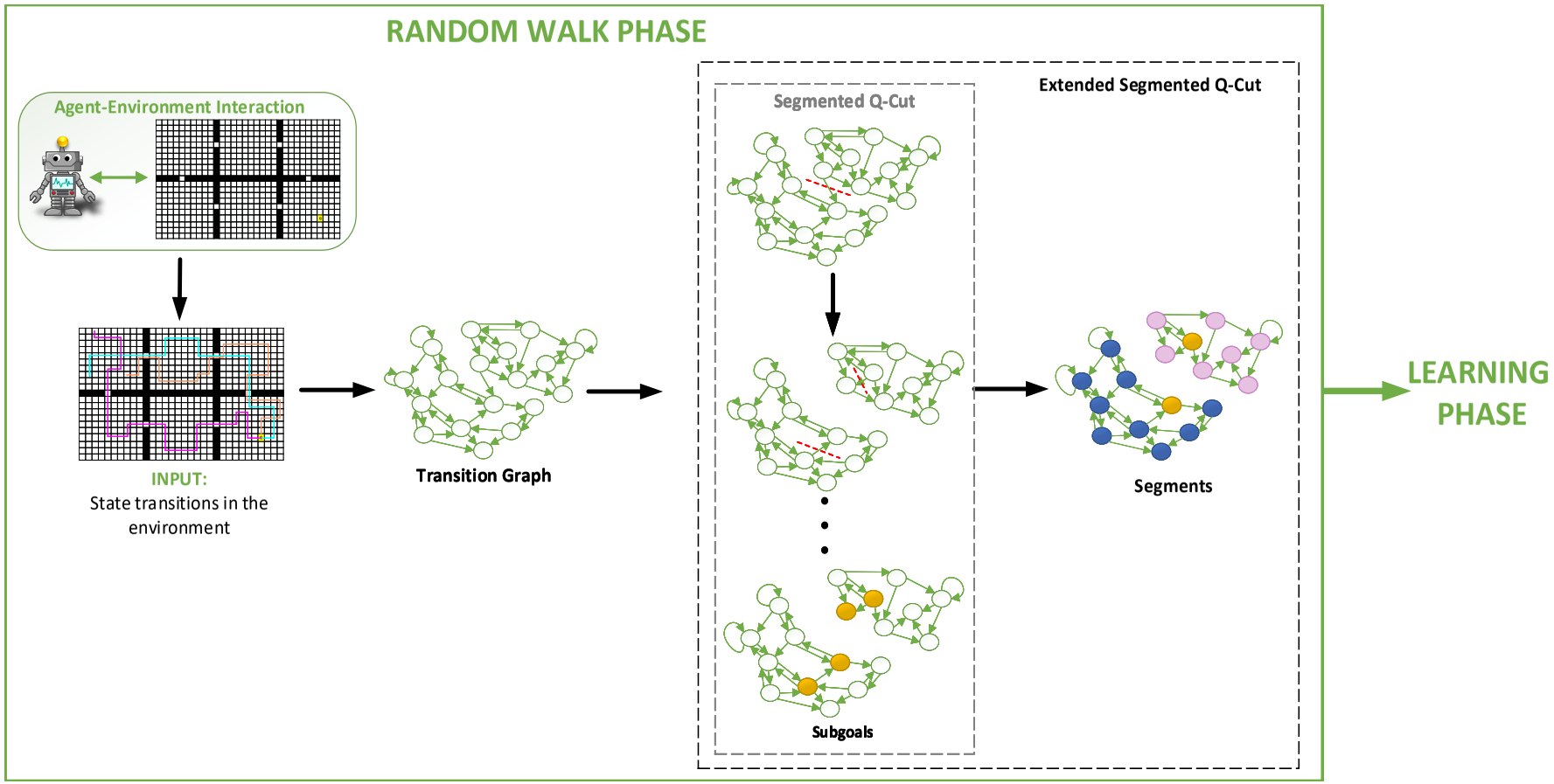


Figure 5.3: A schematic representation for random walk phase.

Similar to Q-Learning algorithm, each episode of the random walk ends until the terminal (goal) state is found or $steps$ limit is reached. At each time step in an episode, the agent chooses a random action $a \in \mathcal{A}$ at a state s and observes its consequences as reward signal r and next state s'. The tuple $\langle s, a, s' \rangle$ experienced by the agent is added to the transition graph G. As explained in the Section 2.7, graph G has nodes which denote the states and arcs that indicate the transition between the states. At each time the transition $\langle s, a, s' \rangle$ is added to the graph G, the frequency of observing transition from s to s' is incremented by one. Moreover, the frequency of visiting state s is also incremented by one. Then, since the arc capacity is defined in terms of relative frequency as defined in the Equation 2.26, the capacity of the arc (s, s') that reflects the transition $s \rightarrow s'$ is adjusted accordingly using these updated frequencies. After M_{cut} episodes are completed in the random walk, the transition graph is accumulated with the agent's experiences. The graph is then given to the `ESegQ-Cut()` method as an input that first computes the subgoals and then extracts the segments of the state-space . Finally, the extracted state-space segmentation information is returned to the *Q-Segmenter* agent for use in the learning process.

5.2.1.2 Extended Segmented Q-Cut

The main idea of ESegQ-Cut method is to identify the subgoals with Segmented Q-Cut algorithm, and then extract the state-space segments using identified subgoals.

The procedure starts with appending all the nodes in transition graph G to $segments$ list and initializing the required lists to store subgoal related information. In addition, $source$ and $sink$ nodes are also determined at this point. Since the agent may start at a different state in each episode of the random walk, $source$ node is selected as a dummy node added to the graph. Similarly, each episode of the random walk may terminate at the goal state or an arbitrary state depending on the $steps$ limit. Therefore, $sink$ node is selected as the goal state if it exists in the graph G, otherwise a dummy goal state is chosen.

The procedure continues with the application of the Segmented Q-Cut algorithm with the `Cut()` method explained in Algorithm 14. While we can identify good-quality cut point(s) on the graph which is denoted by the boolean variable can_divide, we

Algorithm 12: `randomWalk`

Input : $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, $steps$, M_{cut}, c_q, $segments$

Output: $segments$

Initialization: $episode = 0$

Graph $G = \langle N, A \rangle, N = \emptyset, A = \emptyset$

for $episode = 0$ **to** M_{cut} **do**

  Initialize $s \in \mathcal{S}$

  while s *is not terminal or* $steps$ *is not reached* **do**

    Randomly choose an action $a \in \mathcal{A}$

    Take action a, observe r, s'

    Add the transition $\langle s, a, s' \rangle$ to the graph G

      Increment the frequency of observing transition $s \rightarrow s'$

      Increment the frequency of visiting state s

      Adjust the capacity of the arc (s, s') with updated frequencies

    $s \leftarrow s'$

  end while

end for

$segments \leftarrow$ `ESegQ-Cut`$(G, c_q, segments)$

return $segments$

divide the graph into two segments and proceed with each created segment. Quality of a cut is expressed as the ratio cut bi-partitioning metric as defined in the Equation 2.25. Since good quality cut should have a small number of arcs while separating significant balanced areas, only the cuts having quality level greater than pre-determined threshold c_q are acceptable. Thus, Segmented Q-Cut method runs until no new cut which satisfies the quality condition is identified. At the end of this method, subgoals are detected as the cut points.

The extension of the Segmented Q-Cut method starts after obtaining the subgoals. To extract the state-space segments, we first search for adjacent subgoals in the identified $subgoals$ list. We group the adjacent subgoals and then for each group, we choose the subgoal having minimum degree and append it to the $selected\_subgoals$ list. The reason why we choose the subgoal having minimum degree among the group is due to the definition of a significant i.e good quality cut. As explained in [26], a significant source-sink cut ($s - t$ cut) is the cut with small number of arcs and the one creates enough states both in source segment N_s and sink segment N_t. To this end, we aim to obtain only significant subgoals with this extension.

To move from subgoals to state-space segments, nodes denoting the selected significant subgoals are removed from transition graph G. With Union Find algorithm, weakly connected components of the resulting graph are detected. Each detected weakly connected component is treated as a new segment and added to the $segments$ list. Finally, each identified subgoal in $selected\_subgoals$ is also added to the $segments$ list as a separate segment. The procedure terminates after extracted $segments$ are returned.

5.2.2 Reward Shaping Based on State-Space Segmentation

Within the completion of the random walk procedure, *Q-Segmenter* agent gains a knowledge on the environment through state-space segments. The agent should then benefit from this knowledge in the learning phase. However, the question is, how should the agent utilize state-space segment information on its learning process to speed up the learning? The proposed method suggests applying potential-based reward shaping using the segment information in terms of values. The basic idea is

Algorithm 13: `ESegQ-Cut`

Input : Graph $G = \langle N, A \rangle, c_q, segments$

Output: $segments$

Initialization:

$segments \leftarrow$ all nodes in the graph G

$source \leftarrow$ dummy node

$sink \leftarrow$ node denoting the goal state if exists in G, otherwise dummy goal

$subgoals \leftarrow$[ ], $subgoal\_groups \leftarrow$[ ], $selected\_subgoals \leftarrow$[ ], $wcc \leftarrow$ [ ]

$can\_divide \leftarrow True$

while *can_divide* **do**

  for *each segment in the* $segments$ **do**

    $can\_divide \leftarrow can\_divide$ and

     `Cut`$(G, c_q, segments, source, sink, subgoals)$

  end for

end while

$subgoals \leftarrow$ `getSubgoals(); /* call Subgoal-Identifier's method */`

$subgoal\_groups \leftarrow$ Identify the adjacent subgoals in $subgoals$, if any

for *each* $group$ *in* $subgoal\_groups$ **do**

  Append subgoal having minimum degree in the $group$ to $selected\_subgoals$

end for

for *each node* n *in* $selected\_subgoals$ **do**

  $A \leftarrow A \setminus \{(n, i)\}, \forall i \in N$

  $A \leftarrow A \setminus \{(i, n)\}, \forall i \in N$

  $N \leftarrow N \setminus \{n\}$

end for

$wcc \leftarrow$ *Union Find(G)* ; `/* weakly connected components of resulting` G `*/`

$segments \leftarrow segments \cup \{component\}, \forall component \in wcc$

$segments \leftarrow segments \cup \{subgoal\}, \forall subgoal \in selected\_subgoals$

return $segments$

Algorithm 14: `Cut`

Input : Graph $G = \langle N, A \rangle$, c_q, $segments$, $source$, $sink$, $subgoals$

Initialization: Subgraph $G_{sub} = \langle N_{sub}, A_{sub} \rangle$, $N_{sub} = \emptyset$, $A_{sub} = \emptyset$

$G_{sub} \leftarrow$ Induced subgraph of the graph G containing the nodes in $segments$ and the arcs between those nodes

$min\_cut\_value, partitions \leftarrow$ `minimumCut`$(G_{sub}, source, sink)$

Obtain the source segment list N_s and sink segment list N_t from $partitions$

$n_s \leftarrow \mid N_s \mid$; `/* size of source segment */`

$n_t \leftarrow \mid N_t \mid$; `/* size of sink segment */`

$A_{s,t} \leftarrow$ number of nodes connecting $N_s \& N_t$

$quality \leftarrow \frac{n_s \cdot n_t}{A_{s,t}}$

if $quality > c_q$ **then**

  Identify $source_s, sink_s$ for N_s ; `/* new source and new sink for` N_s `*/`

  Identify $source_t, sink_t$ for N_t ; `/* new source and new sink for` N_t `*/`

  $subgoals \leftarrow subgoals \cup \{sink_s\}$

  $subgoals \leftarrow subgoals \cup \{source_t\}$

  $segments \leftarrow$ [ ]

  $segments \leftarrow segments \cup \{N_s, N_t\}$

  return **True**

end if

return **False**

to compute the values of the segments and reflect the potential of the states in terms of segment values. Depending on the potentials, environmental reward signals are shaped and utilized in the Q-value estimations. The illustration that summarizes the learning phase is given in the Figure 5.4. In the following sections, potential-based reward shaping strategy depending on the value of segments is presented.

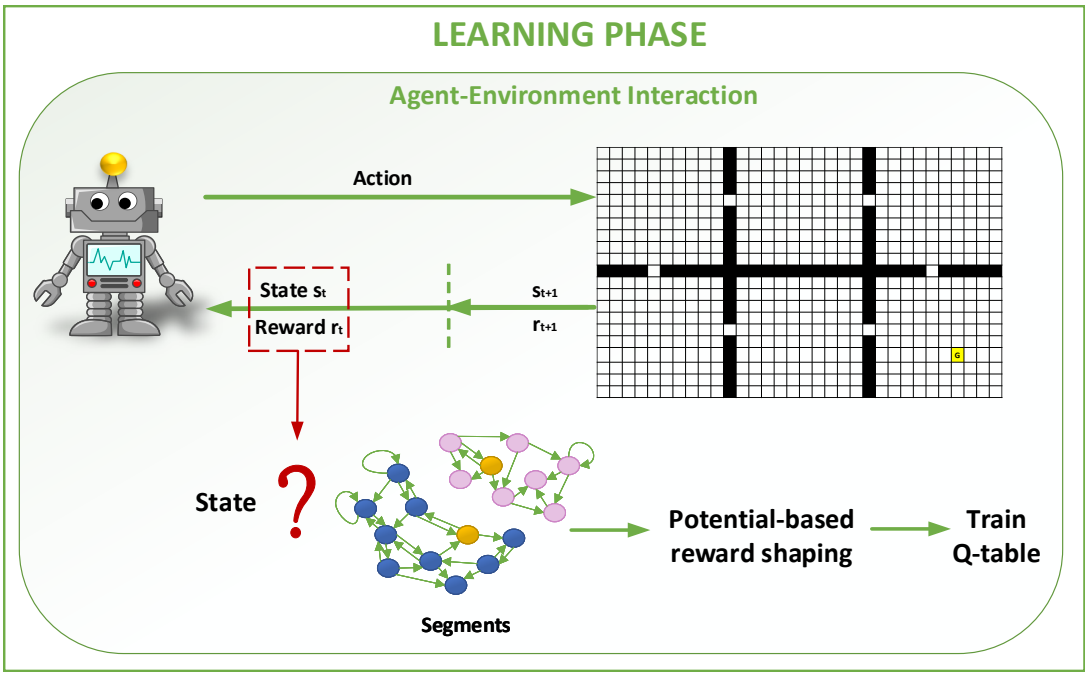


Figure 5.4: A schematic representation for the learning phase.

5.2.2.1 Value of the Segments

The *value* of a segment i denoted by v_{seg_i}, $i \in segments$, is defined as the expected return starting from segment i and computed as the average Q-value of all possible states in segment i for all possible actions with

$$v_{seg_i} = \frac{\sum_{s \in i} \frac{\sum_{a \in \mathcal{A}} Q(s,a)}{|\mathcal{A}|}}{\mid i \mid}, \tag{5.1}$$

where $Q(s,a) \doteq \mathbb{E}\left[G_t \mid s_t = s, a_t = a\right]$ shows the expected return starting from state s and taking action a, $\mid \mathcal{A} \mid$ represents the size of the action-space and $\mid i \mid$ denotes the size of the segment i i.e. the number of states that belong to segment i.

Algorithm 15: `getSegmentValues`

Input : $segments$, $segment\_values$, Q_{SEG}

Output: $segment\_values$

for *each segment* i *in* $segments$ **do**
  $sum = 0$
  for *each state* s *in* $segments$ **do**
    $sum = sum + \frac{\sum_{a \in \mathcal{A}} Q_{SEG}(s,a)}{|\mathcal{A}|}$
  end for
  $segment\_values[i] = \frac{sum}{|segments[i]|}$
end for
return $segment\_values$

If any segment is found by the completion of random walk procedure, the agent computes the segment values with `getSegmentValues` procedure as given in the Algorithm 16 using the rule (5.1). Since Q-values for each state-action pair are initialized to zero at the beginning of the first episode, initially segment values also become zero. However, as the Q-values are updated during the learning process, segment values should be changed accordingly. By doing this, the agent will be able to determine which segment is good or bad. Therefore, *Q-Segmenter* updates the segment values at the end of each episode by `updateSegmentValues` method. As explained in more detail in the Algorithm 16, this method updates the segment values with the update rule

$$v_{seg_i}^{n+1} \leftarrow v_{seg_i}^{n} + \alpha(\gamma v_{seg_i}^{n+1} - v_{seg_i}^{n}), \tag{5.2}$$

where $\alpha \in (0, 1]$ is the learning rate, $\gamma \in (0, 1]$ is the discount factor and $n \in [0, M]$ denotes the episode number.

5.2.2.2 Potential-based Reward Shaping Using Values of the Segments

Following the computation of segment values, *Q-Segmenter* starts interacting with the environment by applying Q-Learning algorithm. However, it shapes the reward signals received from the environment with potential-based reward shaping based on state-space segmentation and uses shaped rewards to update its Q-value estimates for

Algorithm 16: `updateSegmentValues`

Input : $segments, segment\_values, \alpha, \gamma$

Output: $segment\_values$

```
new_segment_values ← getSegmentValues()
for each segment i in segments do
    q_seg_update = γ(new_segment_values[i]) − segment_values[i]
    segment_values[i] = segment_values[i] + α(q_seg_update)
end for
return segment_values
```

state-action pairs.

The *potential* of a state s denoted with the function $\Phi(s), \Phi : \mathcal{S} \rightarrow \mathbb{R}$ is defined by

$$\Phi(s) = \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i}, \tag{5.3}$$

where $\mathbb{1}_{s \in i}$ is the indicator function that takes value of 1 if state s is in segment i, 0 otherwise and v_{seg_i} represents the value of the segment i. A state can be an element of only one segment. From this, we can conclude that the potential of a state shows the value of the segment to which the state belongs.

Let, we define the potential-based reward shaping function $F, F : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$ in terms of the difference of the potentials of states s and s' for transition $s \rightarrow s'$ as

$$\begin{aligned} F(s, s') &= \gamma\Phi(s') - \Phi(s) \\ &= \gamma \sum_{i \in segments} \mathbb{1}_{s' \in i} v_{seg_i} - \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i} \\ &= \gamma v_{seg_k} - v_{seg_j}, \end{aligned} \tag{5.4}$$

where $s' \in$ segment k, $s \in$ segment j and $k, j \in segments$.

Theorem 2 *Let MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$ and transformed MDP M' is defined as $M' = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}', \gamma \rangle$ where $\mathcal{R}' = \mathcal{R} + F$ and shaping reward function $F : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$ is defined as in (5.4). Potential-based reward shaping function F defined in (5.4)*

preserves the policy invariance i.e. every optimal policy in M' is also an optimal policy in M (and vice versa).

Proof. Optimal Q-function for the original MDP M denoted as Q^*_M satisfies the Bellman optimality equation:

$$Q^*_M(s,a) = \mathbb{E}_{s'}\left[\mathcal{R}\left(s,a\right) + \gamma \max_{a' \in A} Q^*_M\left(s',a'\right)\right]. \tag{5.5}$$

When we subtract $\Phi(s)$ from both sides

$$\begin{aligned} Q^*_M(s,a) - \Phi(s) = \mathbb{E}_{s'}[&\mathcal{R}\left(s,a\right) + \gamma\Phi\left(s'\right) - \Phi(s) \\ &+\gamma \max_{a' \in A}\left(Q^*_M\left(s',a'\right) - \Phi\left(s'\right)\right)\Bigg]. \end{aligned} \tag{5.6}$$

Since $\Phi(s) = \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i}$, we get

$$\begin{aligned} Q^*_M(s,a) - \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i} = \mathbb{E}_{s'}[&\mathcal{R}\left(s,a\right) \\ &+\gamma \sum_{i \in segments} \mathbb{1}_{s' \in i} v_{seg_i} - \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i} \\ &+\gamma \max_{a' \in A}\left(Q^*_M\left(s',a'\right) - \sum_{i \in segments} \mathbb{1}_{s' \in i} v_{seg_i}\right)\Bigg]. \end{aligned} \tag{5.7}$$

If we define $Q_{M'}(s,a) = Q^*_M(s,a) - \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i}$ and substitute $F(s,s') = \gamma \sum_{i \in segments} \mathbb{1}_{s' \in i} v_{seg_i} - \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i}$ back to (5.7), we obtain

$$\begin{aligned} Q_{M'}(s,a) &= \mathbb{E}_{s'}\left[\mathcal{R}\left(s,a\right) + F\left(s,s'\right) + \gamma \max_{a' \in A} Q_{M'}\left(s',a'\right)\right] \\ &= \mathbb{E}_{s'}\left[\mathcal{R}'\left(s,a\right) + \gamma \max_{a' \in A} Q_{M'}\left(s',a'\right)\right]. \end{aligned} \tag{5.8}$$

The equation obtained in (5.8) is the Bellman optimality equation applied for transformed MDP M'. Thus, $Q_{M'}$ must be optimal state-action value function since it satisfies the Bellman optimality equation. Therefore,

$$Q^*_{M'}(s,a) = Q_{M'}(s,a) = Q^*_M(s,a) - \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i}. \tag{5.9}$$

Following 5.9, the optimal policy for transformed MDP M' satisfies the following

$$\begin{aligned} \pi^*_{M'}(s) &\in \arg\max_{a \in A} Q^*_{M'}(s,a) \\ &= \arg\max_{a \in A} Q^*_M(s,a) - \sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i} \\ &= \arg\max_{a \in A} Q^*_M(s,a). \end{aligned} \tag{5.10}$$

□

Since the effect of actions are eliminated in the segment values by averaging the Q-values over all possible actions as shown in the Equation (5.1), the optimal policy for M' is not affected by the additional $-\sum_{i \in segments} \mathbb{1}_{s \in i} v_{seg_i}$ term. Hence, the optimal policy for M' becomes also optimal for M and policy invariance is preserved with the policy-based reward shaping function F defined in the Equation (5.4). Similarly, we can prove that the optimal policy for M will also be optimal in M' by considering the reward shaping function $-F$.

The shaped reward signal for the transition of *Q-Segmenter* agent time step t becomes

$$\tilde{r_t} = r_t + F(s_t, s_{t+1}). \tag{5.11}$$

Depending on the transition of the agent, the shaped reward can take positive or negative values. If the agent transitions to a state with greater segment value than the current one, then F will behave as a bonus for the agent. On the other hand, the agent will receive a punishment for visiting a state with a lower segment value. Even though the agent traverses in the same segment, F will take negative value due to the discount factor. This is especially helpful in sparse reward environments since the agent receives frequent feedback while being in the same segment and it may encourage agent to complete the task as fast as possible.

5.3 Computational Experiments

This section presents the results of computational experiments for the evaluation of learning performance of the proposed method. In Section 5.3.1, problem domains used in the experiments are explained. The parameter settings in the experiments are provided in the Section 5.3.2. Finally, experiment results and evaluation of the method is given in the 5.3.3.

5.3.1 Sample Problem Domains

For performance evaluation of the proposed method, we preferred to use sparse-reward problem benchmarks as diagnostic tasks. Hence, two well-known versions of GridWorld navigation domains which are `Six-Rooms` and `Zigzag Four-Rooms` were suitable settings for the experiments. As introduced in the Section 4.4.1 of Chapter 4, the RL agent needs to find the goal state labeled as G in an environment having several rooms connected by hallways as sketched in the (a) and (c) of Figure 4.3. In each episode of learning, the agent starts from an arbitrary state in the upper-left room and by taking actions among action set *north, south, east*, and *west*, it observes the consequences of its actions at the very end of the episode either by receiving $+10$ reward for reaching the goal state or 0 for all other cases.

Furthermore, we also experimented on a more complex version of `Six-Rooms` domain called `Locked Shortcut Six-Rooms`. In this problem, the south-west door is locked until the state with the key visited ss illustrated in the Figure 5.5. The state is defined as a tuple $\langle x, y, 0/1 \rangle$ that shows the x & y-coordinates of the agent's location and whether agent has the key i.e. visited the state with key or not. Thus, the size of the state-space is $32 \times 21 \times 2$. Similar to the `Six-Rooms`, action space consists of *north, south, east*, and *west*. In each episode of learning, the agent always starts from the south-west corner of the grid and tries to reach the goal state denoted with G. This is also a sparse reward environment as the agent receives a reward of $+10$ for reaching the goal state or 0 for all other cases in an episode.

The delayed-feedback nature of these problem domains makes them highly suitable for evaluation of our proposed method in terms of learning speed. We carried out the

experiments using the same sizes of state-space and action-space as provided in the Table 4.1.

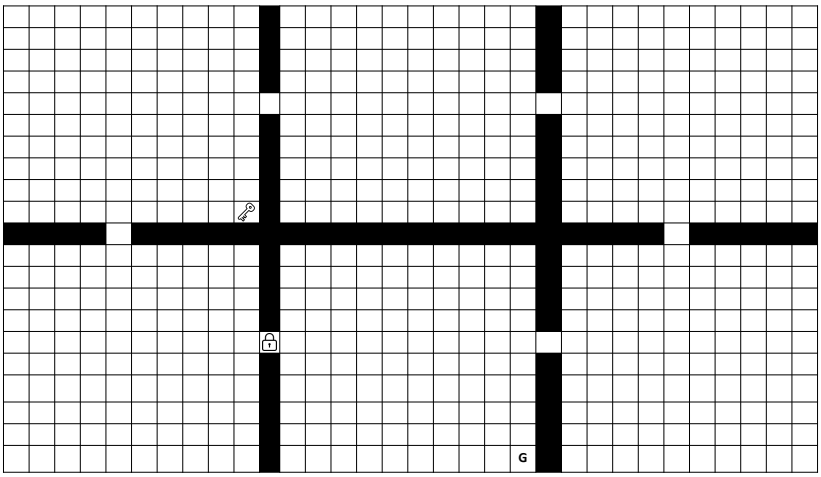


Figure 5.5: `Locked Shortcut Six-Rooms` domain.

5.3.2 Experiment Settings

The parameters are set as given in the Table 5.1 for the experimental study. We used two different $steps$ limit for an episode in the problem domains. We set the parameter to 1000 and 2000 for `Six-Rooms`, 2000 and 5000 for `Zigzag Four-Rooms`, and 3000 and 5000 for `Locked Shortcut Six-Rooms` environments. By doing so, we aim to observe the impact of having a limited amount of interaction time on the agent's learning and whether our proposed method speeds up learning in such a case or not.

We performed the experiments using workstation having Intel® Core™ i7 3.10 GHz processors and 16 GB RAM with 64-bit Microsoft Windows 10 operating system. We replicate each experiment for 50 times and present the average learning performances in the next section.

5.3.3 Experiment Results and Discussion

We determined Q-Learning algorithm as the benchmark since SegQ-Cut method is originally introduced to accelerate Q-Learning algorithm [26]. Furthermore, we com-

Table 5.1: Parameter settings for the experiments.

Parameter	**Value**
Learning rate α	0.3
Discount rate γ	0.9
Exploration probability ε	linearly decaying from 0.1 to 0.05
Number of episodes M	1000
Steps limit $steps$	domain-dependent
Number of episodes for random walk M_{cut}	25
Cut quality threshold c_q	1000

pared the learning performances in terms of measures such as average number of steps taken to reach the goal state, average reward per step, and average elapsed time required for learning. Average reward per step is computed by dividing total reward in an episode by the episode length. As we run each experiment for 50 times, the average performance of all experiments are reported in the Table 5.2. In addition, since convergence rate reflects the learning speed of the approaches in the RL literature, we also displayed the average reward and average number of steps measures on figures from 5.9 to 5.11. Each figure shows the average learning performance of 50 runs.

To illustrate the identified state space segments, we provide three examples that shows segments and cut set on the transition graph after random walk phase for 25 episodes is completed for each domain. As can be seen from Figures 5.6 to 5.8, *Segmenter* component is able to identify each room as a separate segment. However, in some cases, it aggregates several rooms and gather them in a single segment as depicted in Figure 5.6. These are only the examples from single run, hence we observed other combinations of segments (at least two segments) in the rest of the experiments. For instance, it is observable from Figure 5.8 that Q-Segmenter could not identify the each room as a separate segment.

As can be noted from Table 5.2, *Q-Segmenter* outperforms Q-Learning in all domains regarding the number of steps to reach the goal state and average reward measures. It also converges to better results in all domains. This indicates that although *Q-Segmenter* is able to identify at least two segments in each domain, PBRS mechanism contributes significantly to the improve agent's learning. In terms of elapsed time, there is no clear dominant method since in some domains Q-Segmenter beats

Q-Learning, and vice versa. However, when the time for learning phase is considered, we can conclude that there is no substantial difference between learning times. Finally, when we compare the learning speeds from Figures 5.6 to 5.8, we notice that the learning is much faster than Q-Learning. This shows us, *Q-Segmenter* learns much earlier than 1000 episodes are completed. If we also account the observation that *Q-Segmenter* is able to learn much faster than Q-Learning, then the training time for *Q-Segmenter* is not need to be that longer. Thus, the actual elapsed time for learning will be smaller. As a result, our proposed method indeed accelerates the learning in problem domains with sparse explicit reward structure.

Table 5.2: Overall performance comparison of the proposed method *Q-Segmenter* with benchmarks.

Problem	**Method**	**Average Steps (stdev)**		**Average Reward (stdev)**		**Average**	**Average Elapsed Time (sec) for**	
		over all episodes	**of the last episode**	**over all episodes**	**of the last episode**	**Elapsed Time sec (stdev)**	**Random Walk Phase (stdev)**	**Learning Phase (stdev)**
Six-rooms	Q-Segmenter	**696.33 (166.83)**	**444.04 (458.39)**	**0.06 (0.04)**	**0.13 (0.11)**	121.26 (45.17)	20.58 (9.09)	100.67 (45.83)
(1000 steps)	Q-Learning	984.99 (17.94)	929.40 (228.14)	0.00 (0.00)	0.01 (0.05)	**93.71 (7.15)**	-	-
Six-rooms	Q-Segmenter	**198.59 (391.22)**	**43.70 (5.19)**	**0.18 (0.08)**	**0.23 (0.03)**	**67.45 (6.70)**	33.47 (1.95)	33.98 (6.09)
(2000 steps)	Q-Learning	741.70 (800.56)	46.94 (5.20)	0.13 (0.09)	0.22 (0.02)	70.48 (14.89)	-	-
Zigzag-four-rooms	Q-Segmenter	**1808.44 (89.51)**	**1489.18 (806.83)**	**0.01 (0.00)**	**0.03 (0.05)**	196.59 (42.78)	6.94 (3.96)	189.64 (46.54)
(2000 steps)	Q-Learning	1924.65 (104.13)	1562.38 (755.86)	0.00 (0.00)	0.02 (0.05)	**186.77 (12.23)**	-	-
Zigzag-four-rooms	Q-Segmenter	**186.45 (473.57)**	**74.00 (6.39)**	**0.11 (0.03)**	**0.14 (0.01)**	**60.93 (4.44)**	30.34 (0.84)	30.58 (4.57)
(5000 steps)	Q-Learning	1169.62 (1742.68)	75.74 (5.40)	0.09 (0.06)	0.13 (0.01)	116.71 (18.78)	-	-
Locked Shortcut Six-rooms	Q-Segmenter	**384.69(669.94)**	**54.00(2.26)**	**0.14(0.06)**	**0.19(0.01)**	130.08 (27.19)	53.66 (4.29)	76.42 (28.05)
(3000 steps)	Q-Learning	1257.07(1301.60)	56.00(3.55)	0.10(0.08)	0.18(0.01)	**116.20 (19.80)**	-	-
Locked Shortcut Six-rooms	Q-Segmenter	**308.39(672.78)**	**56.30(3.27)**	**0.14(0.06)**	**0.18(0.01)**	180.84 (15.43)	66.19 (2.38)	114.64 (15.04)
(5000 steps)	Q-Learning	840.56(1566.00)	56.70(4.30)	0.01(0.01)	0.02(0.00)	**84.58 (12.40)**	-	-

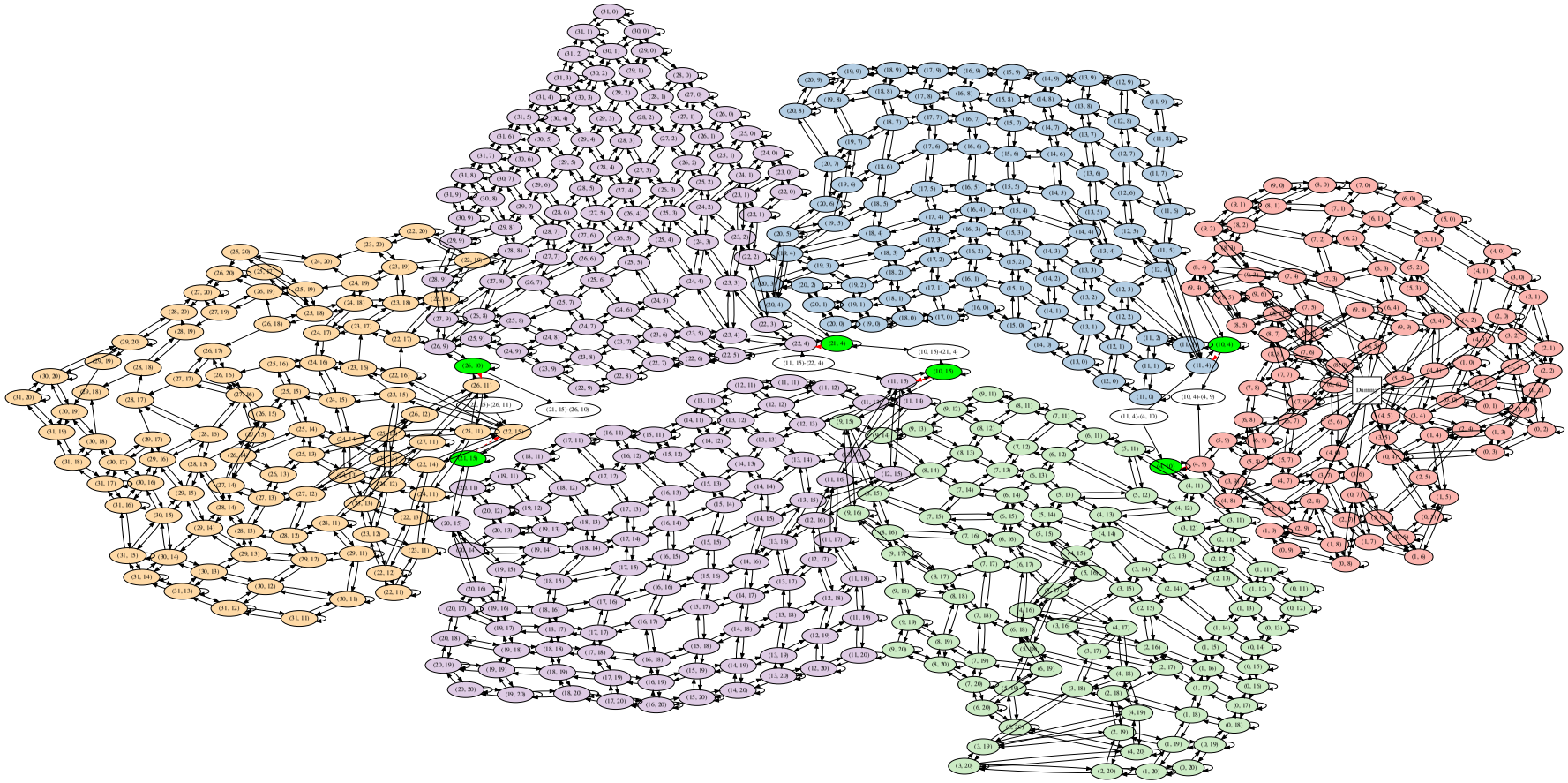


Figure 5.6: Segments and cuts on the graph after random walk phase for 25 episodes is completed in `Six-Rooms GridWorld` domain.

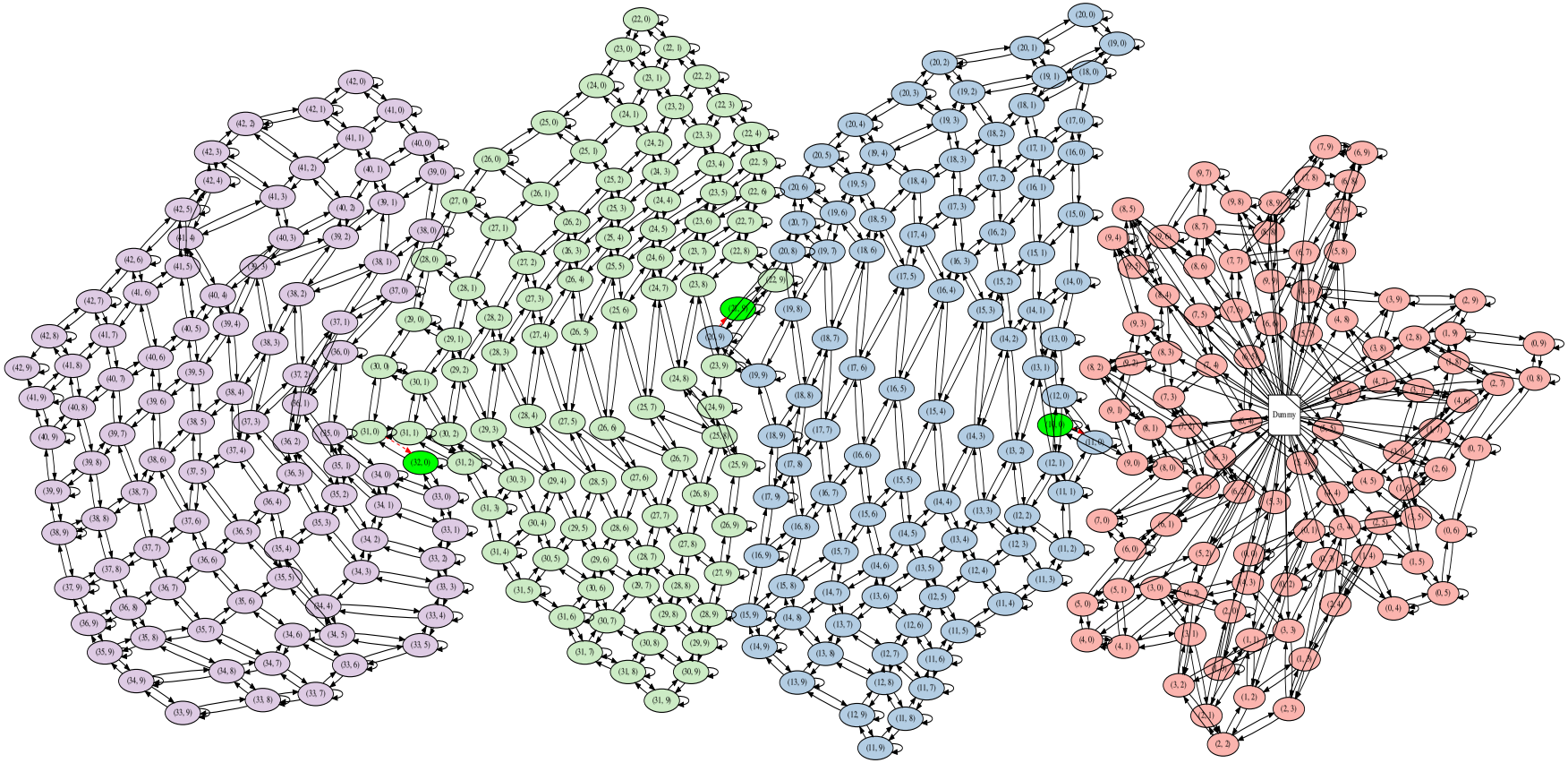


Figure 5.7: Segments and cuts on the graph after random walk phase for 25 episodes is completed in `Zigzag Four-Rooms GridWorld` domain.

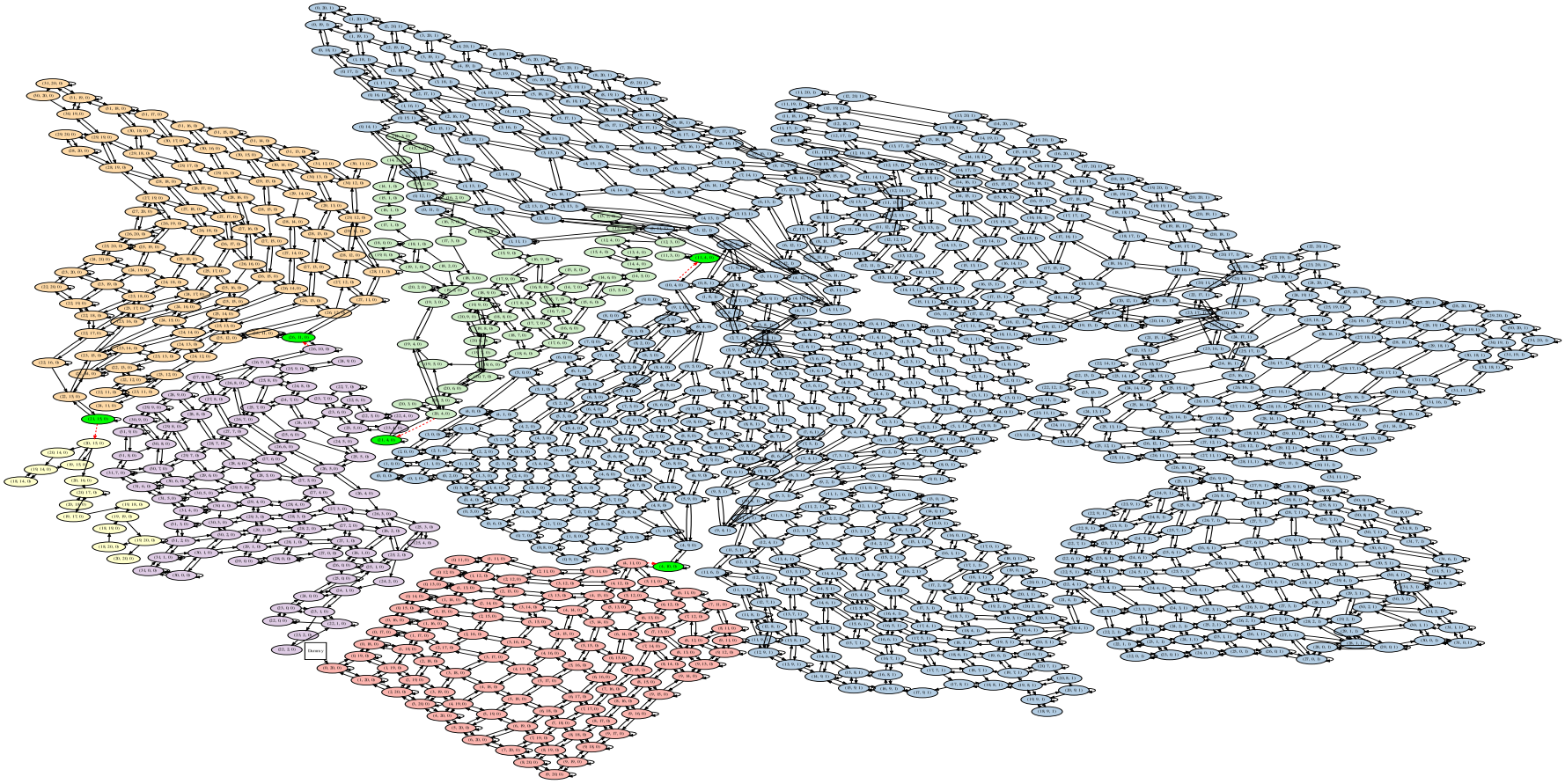
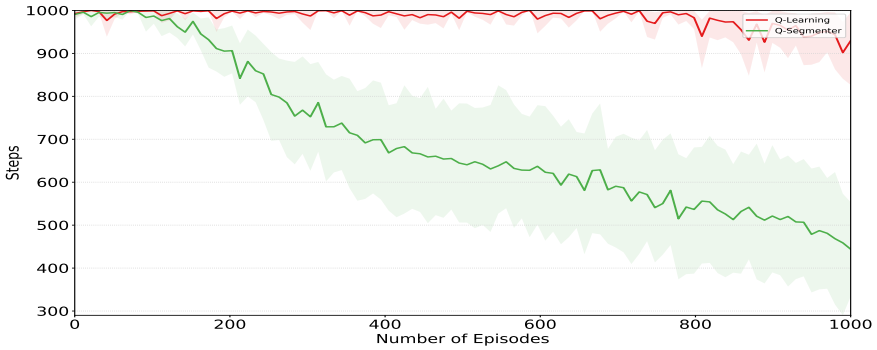
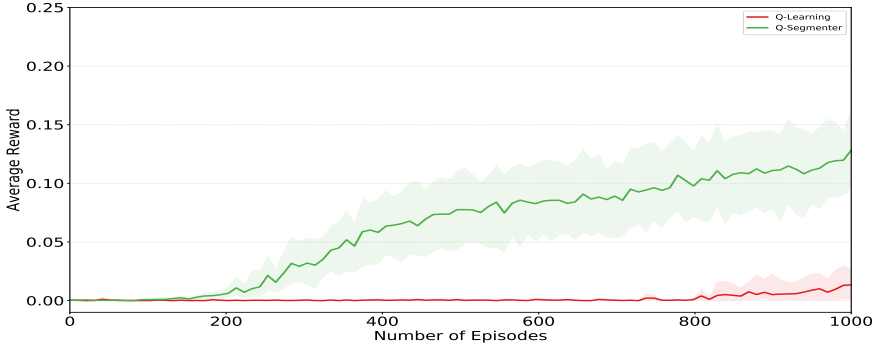


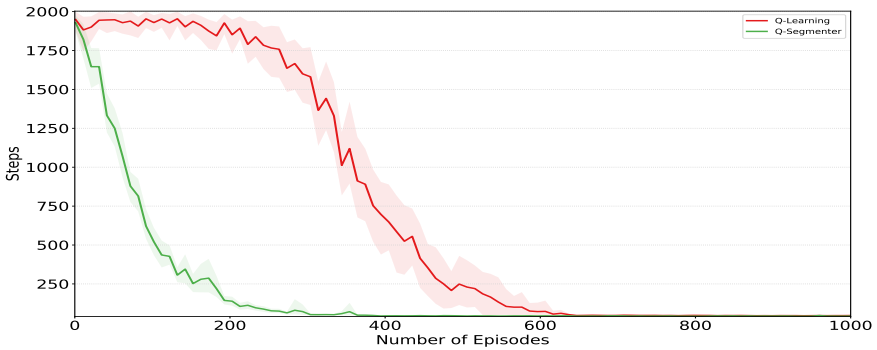
Figure 5.8: Segments and cuts on the graph after random walk phase for 25 episodes is completed in `Locked Shortcut Six-Rooms` domain.



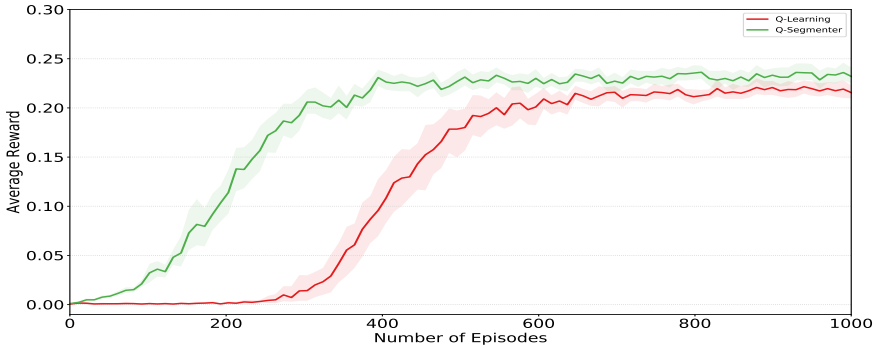
(a) Average steps to reach the goal state under 1000 steps limit.



(b) Average reward per episode under 1000 steps limit.

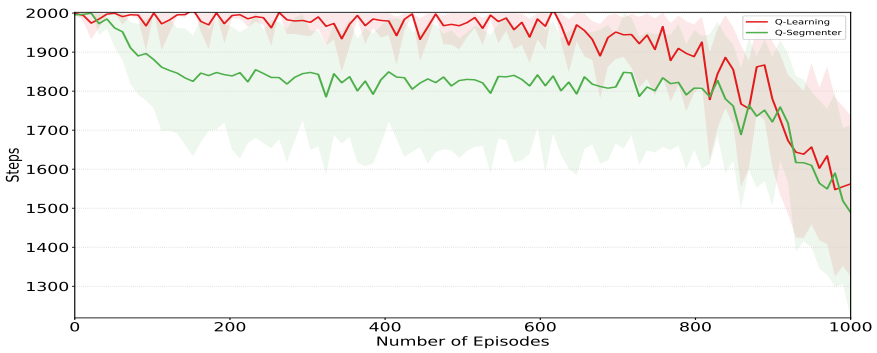


(c) Average steps to reach the goal state under 2000 steps limit.

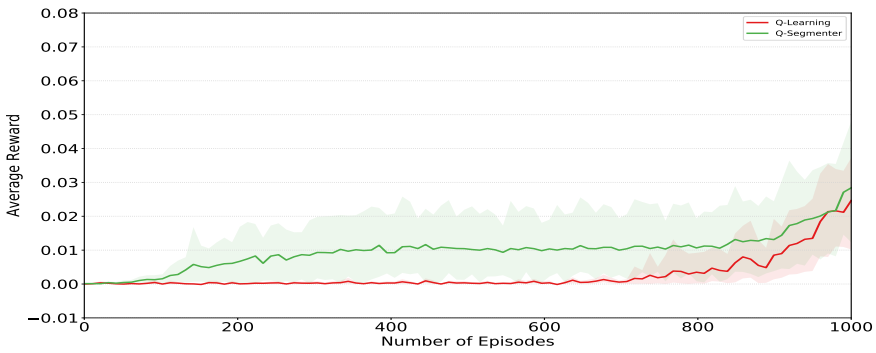


(d) Average reward per episode under 2000 steps limit.

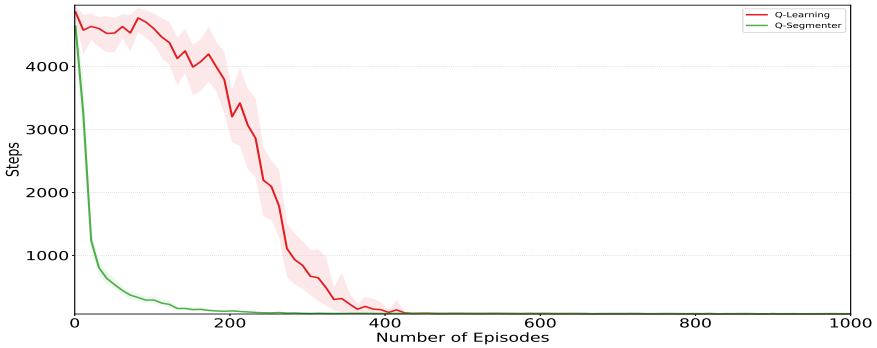
Figure 5.9: Learning performances of proposed method *Q-Segmenter* and Q-Learning for `Six-Rooms GridWorld` domain under 1000 and 2000 steps limit.



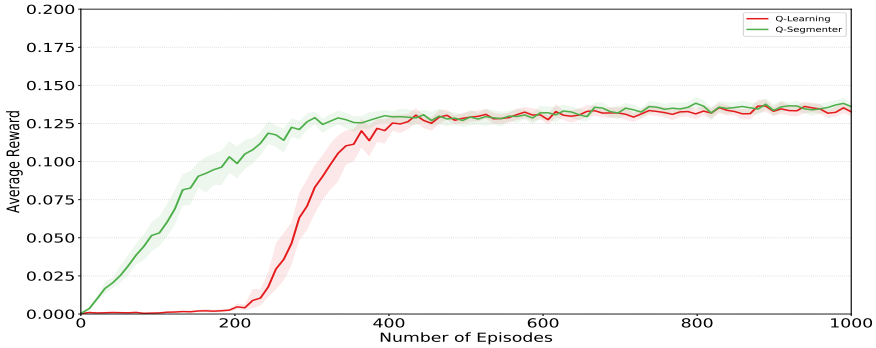
(a) Average steps to reach the goal state under 2000 steps limit.



(b) Average reward per episode under 2000 steps limit.

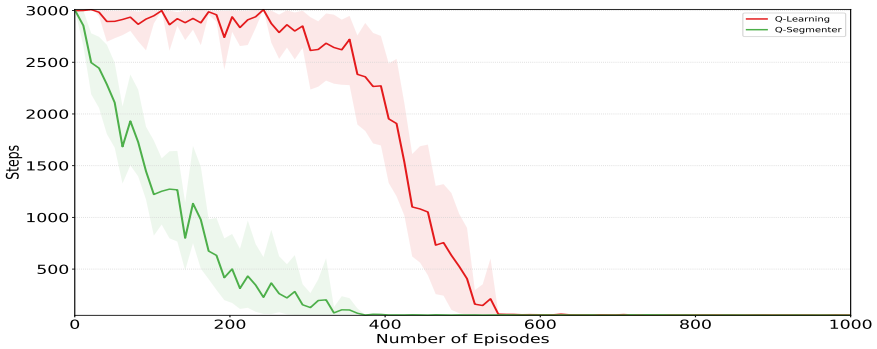


(c) Average steps to reach the goal state under 5000 steps limit.

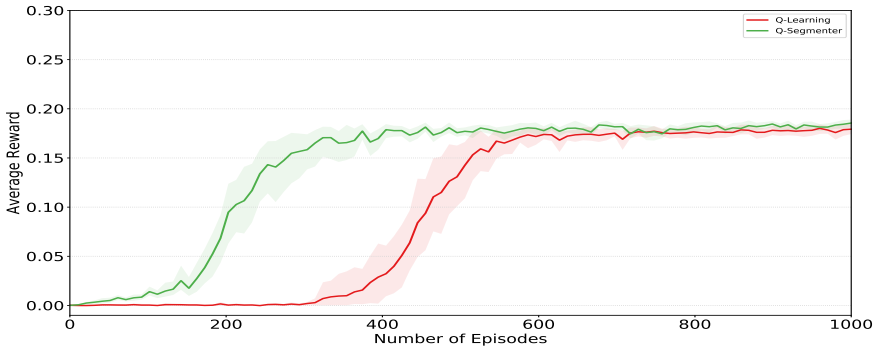


(d) Average reward per episode under 5000 steps limit.

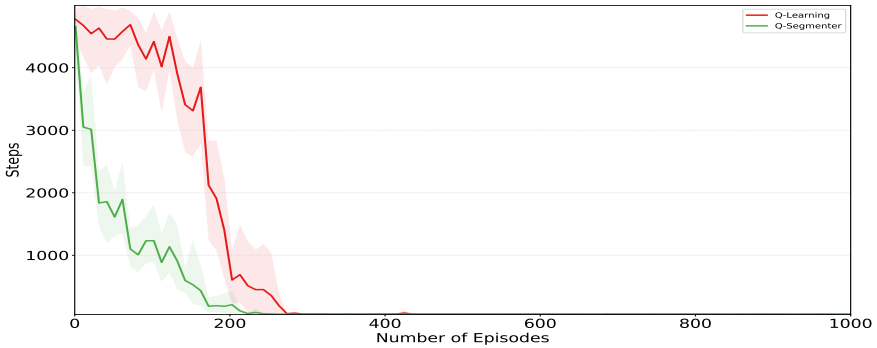
Figure 5.10: Learning performances of proposed method *Q-Segmenter* and Q-Learning for `Zigzag Four-Rooms GridWorld` domain under 1000 and 5000 steps limit.



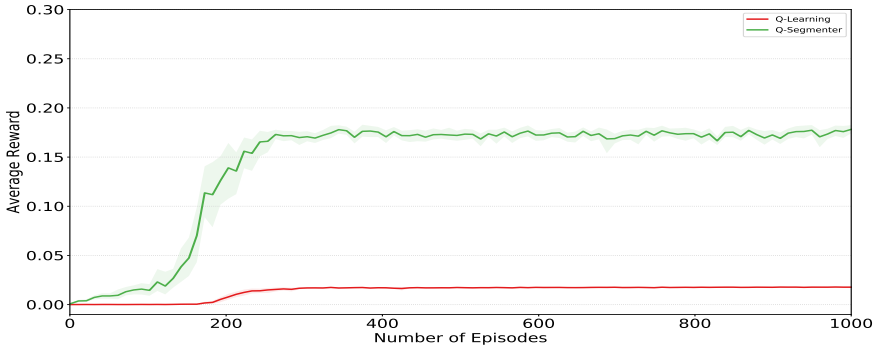
(a) Average steps to reach the goal state under 3000 steps limit.



(b) Average reward per episode under 3000 steps limit.



(c) Average steps to reach the goal state under 5000 steps limit.



(d) Average reward per episode under 5000 steps limit.

Figure 5.11: Learning performances of proposed method *Q-Segmenter* and Q-Learning for `Locked Shortcut Six-Rooms GridWorld` domain under 3000 and 5000 steps limit.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis focuses on learning efficiency problem in Reinforcement Learning (RL) tasks. For an RL agent to master a task efficiently, it should improve its behavior using the extracted information about its environment through strategic exploration. Hence, efficient exploration is an important step that the agent should reach on the path of efficient learning.

Furthermore, the required time for learning is a significant factor that impacts the learning efficiency of an RL agent. In tasks with sparse explicit reward structure, the RL agent struggles to extract the policy which leads to completing the task successfully due to the lack of immediate feedback on its behavior.

In this thesis, we propose two methods with a reward shaping mechanism to attack efficient exploration and speeding up learning problems in the RL settings. We introduce a framework called *population-based repulsive reward shaping mechanism using eligibility traces* for efficient exploration problem. This framework helps to explore the state-space in a coordinated manner with a population of sub-agents that employ eligibility traces. The coordinator called *RRS-Agent* shapes the environmental reward signals of the sub-agent population and benefits from the experiences of them in its learning process. Our computational study on various well-known RL problem domains showed that the framework achieves coordinated exploration with an improvement in the state-space coverage and learning performance. Furthermore, the framework brings a new perspective to RL literature by consolidating the eligibility traces of multiple-agents to create coordination.

Second, we propose *potential-based reward shaping using state-space segmentation*

with the extended segmented Q-Cut algorithm to accelerate learning. In this method, rather than using a reward shaping mechanism to extract a useful information about the environment, we apply potential-based reward shaping depending on extracted information about the environment in the learning process of the agent. That means, the reward shaping is performed using the state-space segment information. Our analyses on sparse-reward problem settings showed that the proposed method accelerates the learning of the agent while maintaining policy invariance and without the need to prolong the computation time.

In future work, the first proposed method can be applied with function approximation algorithms to evaluate the performance in a wide range of tasks including visual benchmark problems in the RL literature. For the second method, one can consider using the *Segmenter* component periodically rather than identifying segments only once at the end of the random walk phase. However, in this direction, the question of how to combine previously identified segment information with the new ones should be carefully thought out. Moreover, the cut quality threshold can be designed with a more systematic approach instead of hand-crafted as it significantly affects the quality of identified segments. Another research direction can be combining both proposed methods in a single framework. That is, using the population of sub-agents in the random walk phase of the second method such that the state-space segment information is extracted from the experiences of the sub-agent population. Through repulsive reward shaping, the complete transition graph can be achieved in a faster way and extracted segment information can be more adequate.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.

[2] S. Amin, M. Gomrokchi, H. Satija, H. van Hoof, and D. Precup. A survey of exploration methods in reinforcement learning. *CoRR*, abs/2109.00157, 2021.

[3] A. G. Barto, S. J. Bradtke, and S. P. Singh. Real-time learning and control using asynchronous dynamic programming. Technical report, University of Massachusetts at Amherst, Department of Computer and Information Science, August 1991.

[4] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying count-based exploration and intrinsic motivation. NIPS'16, page 1479–1487, Red Hook, NY, USA, 2016. Curran Associates Inc.

[5] R. Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[6] R. I. Brafman and M. Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *J. Mach. Learn. Res.*, 3(null):213–231, mar 2003.

[7] J. S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In F. F. Soulié and J. Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[8] P. V. C. Caironi and M. Dorigo. Training and delayed reinforcements in q-learning agents. *Int. J. Intell. Syst.*, 12:695–724, 1997.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[10] O. Şimşek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd International Conference on Machine Learning*, ICML '05, page 816–823, New York, NY, USA, 2005. Association for Computing Machinery.

[11] W. Dabney, G. Ostrovski, and A. Barreto. Temporally-extended ϵ-greedy exploration. *ArXiv*, abs/2006.01782, 2021.

[12] P. Dayan and T. Sejnowski. Exploration bonuses and dual control. In *Machine Learning*, pages 5–22, 1996.

[13] A. Demir, E. Çilden, and F. Polat. Landmark based reward shaping in reinforcement learning with hidden states. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, page 1922–1924, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems.

[14] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, may 1964.

[15] M. Grzes. Improving exploration in reinforcement learning through domain knowledge and parameter analysis. University of York, March 2010.

[16] M. Grzes and D. Kudenko. Plan-based reward shaping for reinforcement learning. volume 31, pages 10–22, 10 2008.

[17] A. Harutyunyan, S. Devlin, P. Vrancx, and A. Nowe. Expressing arbitrary reward functions as potential-based advice. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29(1), Feb. 2015.

[18] A. M. Hinz. The tower of hanoi. *Enseign. Math*, 35(2):289–321, 1989.

[19] S. Iqbal and F. Sha. Coordinated Exploration via Intrinsic Rewards for Multi-Agent Reinforcement Learning. *arXiv:1905.12127 [cs, stat]*, May 2021. arXiv: 1905.12127.

[20] S. Khadka, S. Majumdar, T. Nassar, Z. Dwiel, E. Tumer, S. Miret, Y. Liu, and K. Tumer. Collaborative evolutionary reinforcement learning. 2019.

[21] J. Z. Kolter and A. Y. Ng. Near-bayesian exploration in polynomial time. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 513–520, New York, NY, USA, 2009. Association for Computing Machinery.

[22] A. D. Laud. *Theory and application of reward shaping in reinforcement learning*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[23] A. Mahajan, T. Rashid, M. Samvelyan, and S. Whiteson. Maven: Multi-agent variational exploration. 2019.

[24] O. Marom and B. Rosman. Belief reward shaping in reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[25] M. J. Mataric. Reward functions for accelerated learning. In *In Proceedings of the Eleventh International Conference on Machine Learning*, pages 181–189. Morgan Kaufmann, 1994.

[26] I. Menache, S. Mannor, and N. Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning*, ECML '02, page 295–306, Berlin, Heidelberg, 2002. Springer-Verlag.

[27] I. Menache, S. Mannor, and N. Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *ECML*, 2002.

[28] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann, 1999.

[29] T. Okudo and S. Yamada. Subgoal-based reward shaping to improve efficiency in reinforcement learning. *IEEE Access*, 9:97557–97568, 2021.

[30] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2778–2787. JMLR.org, 2017.

[31] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley amp; Sons, Inc., USA, 1st edition, 1994.

[32] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

[33] J. Schmidhuber. Curious model-building control systems. In *In Proc. International Joint Conference on Neural Networks, Singapore*, pages 1458–1463. IEEE, 1991.

[34] J. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, page 222–227, Cambridge, MA, USA, 1991. MIT Press.

[35] S. Singh, R. Sutton, and P. Kaelbling. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22, 11 1995.

[36] R. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 08 1988.

[37] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *In Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[38] R. S. Sutton. Integrated modeling and control based on reinforcement learning and dynamic programming. In *Proceedings of the 3rd International Conference on Neural Information Processing Systems*, NIPS'90, page 471–478, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[39] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[40] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1–2):181–211, aug 1999.

[41] M. Tokic and G. Palm. Value-difference based exploration: Adaptive control between epsilon-greedy and softmax. In J. Bach and S. Edelkamp, editors, *KI 2011: Advances in Artificial Intelligence*, pages 335–346, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[42] T. Wang, J. Wang, Y. Wu, and C. Zhang. Influence-based multi-agent exploration. *CoRR*, abs/1910.05512, 2019.

[43] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[44] M. Wiering. *Explorations in Efficient Reinforcement Learning*. PhD thesis, Universiteit van Amsterdam, 01 1999.

[45] Z. Yang, M. Preuss, and A. Plaat. Potential-based reward shaping in sokoban. *ArXiv*, abs/2109.05022, 2021.