 CMES, vol.118, no.2, pp.253-274, 2019

A Graph-Based Reinforcement Learning Method with Converged State Exploration and Exploitation

Han Li[1], Tianding Chen[2,*], Hualiang Teng[3] and Yingtao Jiang[4]

Abstract: In any classical value-based reinforcement learning method, an agent, despite of its continuous interactions with the environment, is yet unable to quickly generate a complete and independent description of the entire environment, leaving the learning method to struggle with a difficult dilemma of choosing between the two tasks, namely exploration and exploitation. This problem becomes more pronounced when the agent has to deal with a dynamic environment, of which the configuration and/or parameters are constantly changing. In this paper, this problem is approached by first mapping a reinforcement learning scheme to a directed graph, and the set that contains all the states already explored shall continue to be exploited in the context of such a graph. We have proved that the two tasks of exploration and exploitation eventually converge in the decision-making process, and thus, there is no need to face the exploration vs. exploitation tradeoff as all the existing reinforcement learning methods do. Rather this observation indicates that a reinforcement learning scheme is essentially the same as searching for the shortest path in a dynamic environment, which is readily tackled by a modified Floyd-Warshall algorithm as proposed in the paper. The experimental results have confirmed that the proposed graph-based reinforcement learning algorithm has significantly higher performance than both standard Q-learning algorithm and improved Q-learning algorithm in solving mazes, rendering it an algorithm of choice in applications involving dynamic environments.

Keywords: Reinforcement learning, graph, exploration and exploitation, maze.

1 Introduction

Reinforcement Learning (RL) has found its great use in a lot of practical applications, ranging from problems in mobile robot [Mataric (1997); Smart and Kaelbling (2002);

[1] College of Mathematics, Physics and Electronic Information Engineering, Wenzhou University, Wenzhou, Zhejiang, 325035, China.

[2] School of Physics and Information Engineering, Minnan Normal University, Zhangzhou, Fujian, 363000, China.

[3] Department of Civil and Environmental Engineering, Howard R. Hughes College of Engineering, University of Nevada, Las Vegas, NV, 454015, USA.

[4] Department of Electrical and Computer Engineering, Howard R. Hughes College of Engineering, University of Nevada, Las Vegas, NV, 454015, USA.

[*] Corresponding Author: Tianding Chen. Email: chentianding@163.com.

Huang, Cao and Guo (2005)], adaptive control [Sutton, Barto and Williams (1992); Lewis, Varbie and Vamvoudakis (2012); Lewis and Varbie (2009)], AI-backed chess playing [Silver, Hubert, Schrittwieser et al. (2017); Silver, Schrittwieser, Simonyan et al. (2017); Silver, Huang, Maddison et al. (2016)], among many others. The idea behind reinforcement learning, as illustrated in Fig. 1, is that an agent learns from the environment by interacting with it and receives positive or negative rewards for performing calculated actions, and the cycle is repeated. The key issue of the whole process is to learn a way of controlling the system so as to maximize the total award.

When the agent begins to sense and learn a completely or partially unknown environment, it involves in two distinct tasks: exploration which attempts to collect as much information about the environment as possible, and exploitation which attempts to receive positive rewards as quickly as possible.

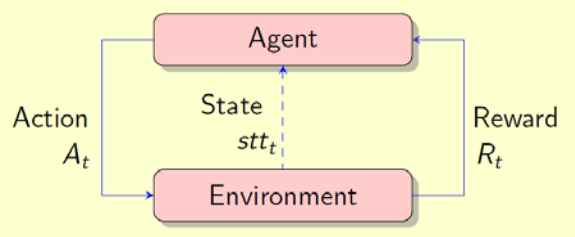

Figure 1: In reinforcement learning, the agent observes the environment, takes an action to interact with the environment, updates its own state and receives reward

There is a dilemma of choosing between the two tasks of exploration and exploitation, though. Too much exploration will adversely influence the efficiency and convergence of the learning algorithm, while putting too much emphasis on exploitation will increase the possibility of falling into a locally optimal solution. The existing RL algorithms all attempt to balance out these two tasks in their learning cycles (Fig. 1), but these is no guarantee that the best result can always be obtained.

Besides the exploration and exploitation dilemma, the RL algorithms have to employ value distributions that inexplicitly assume that environment is static (i.e., no change), or it changes very slowly and/or insignificantly. However, in many real applications, the environment rarely stays unchanged. More than likely, the environment that can be described in terms of states (Fig. 1) changes over the course of exploration. In this case, value distribution has nothing to do with the problem at hand, and all the information obtained from the previous exploration efforts become less, or totally irrelevant.

To effectively solve the aforementioned problems in reinforcement learning, we herein present a new algorithm based on the partitioning of the states set and search of the shortest path in a directed graph that represents a RL method. We have formally proved and experimentally verified that both exploration and exploitation in reinforcement learning actually converge at the end of the decision-making process, and thus, the learning process does not need to face the exploration/exploration dilemma as other existing reinforcement learning methods would do. This observation indicates that a reinforcement learning scheme is essentially the same as searching for the shortest path in

a dynamic environment, which is readily tackled by a modified Floyd-Warshall algorithm as proposed in the paper. The experiment that applies the proposed algorithm to solve mazes confirm better performance of the new algorithm, particularly its effectiveness in addressing issues pertaining to a dynamic environment.

2 Preliminaries and background

In this section, we will first survey the basic structure of reinforcement learning (RL) algorithms, particularly value-oriented method of RL, and formally define the exploration *vs.* exploitation tradeoff in RL. In the literature, RL is shown to be mapped to various graph representations, and these methods are briefly described in the section as well. With graph representations, RL can benefit from rich results in graph algorithms, and we thereby finish this section by reviewing algorithms that search for the shortest path in a graph, as they are related to this paper.

2.1 Value-oriented method for the exploration-exploitation tradeoff in RL

Most RL problems can be formalized using Markov Decision Processes (MDPs), and there are a few key elements in RL as defined below.

1. Agent: An agent takes actions.
2. Environment: The physical world through which the agent operates.
3. State: A state is a concrete and immediate situation in which the agent finds itself. In this paper, we denote stt_i as the state of the agent at time instance i, and set S contains all the states that the agent can operate on. That is, $stt_i \in S$.
4. Action: agents choose among a list of possible actions. Denote a_i as the action that agent might perform at time instance i. $Actions$ is defined as the set of all possible moves of the agent can make, i.e., $a_i \in Actions$.
5. Reward: A reward is the feedback that is used to measure the success or failure of an agent's action. Here a reward at time instance t is defined as R_t. Actions may affect both the immediate reward and, through the next situation, all the subsequent rewards [Sutton and Barto (2017)]
6. Exploitation: a task that makes the best decision given all the current information.
7. Exploration: a task that gathers more information to be used for making the best decision in the future.
8. An episode: the behavior process cycle of the agent from the beginning of the exploration to the beginning of the next exploration. When the interaction between the agent and the environment breaks naturally into subsequences, which are referred as episodes. Each episode ends in a special state called the terminal state, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states [Sutton and Barto (2017)].

In RL, the exploration-exploitation tradeoff refers to a decision making process that chooses between exploration and exploitation. Value-oriented RL methods have to deal with such exploration-exploitation tradeoff through the value distribution as defined by the value function or a probability that decides the chain of actions that lead to the target

state all the way from the start state through a series of awards. A decision chain refers to a series of decision-making steps taken by an agent.

In order to strike a balance between exploration and exploitation, there are two main decision methods that can be followed, the $\epsilon-$ greedy and softmax.

In $\epsilon-$ greedy method, the action is selected by, one has

$$a_{stt} = \begin{cases} a^*_{stt} & with\ probabiltiy\ 1-\epsilon \\ random\ action & with\ probability\ \epsilon \end{cases} \quad (1)$$

where a^*_{stt} is the action in which of the value function assumes the highest value:

$$a^*_{stt} = \text{argmax}_{a \in Actions}\, Q(stt, a) \quad (2)$$

where $Q(stt, a)$ is action-value function which evaluates each possible action while in the current state. One drawback of $\epsilon-$ greedy action selection is that when it explores, all the possible actions are given the equal opportunity, as indicated in Eq. (1). In a simple term, this method is as likely to choose the worst-appearing action as it is to choose the next-to-best action. This gives rise to the so-called softmax method that can vary the action probabilities through a graded function below,

$$\pi(a|stt) = \frac{e^{\frac{Q(stt,a)}{\tau}}}{\sum_{a' \in Actions} e^{\frac{Q(stt,a')}{\tau}}} \quad (3)$$

where $\pi(a|stt)$ is the probability policy to choose an action from the specific state stt, and τ is a "computational" temperature, and is an action-value function that evaluates each possible action in the current state.

The problem of value-oriented method is due to its weak ability to eliminate exploration blindness resulting from a large number of repeated explored states introduced by the value distribution structure. The stochastic factors that are added to help the search process jump out of the loops and balance exploration and exploitation actually come at the expense of more blindness of exploration.

2.2 RL over graphs

A RL can be represented as a directed graph, G<V, E>, where a vertex, $v_i \in V(G)$, corresponds to a state stt_i in reinforcement learning and an edge, $e_{ij} \in E(G)$, connects two vertices (two states $stt_i\ and\ stt_j$) with a decision action $a_i$in reinforcement learning. In this graph, a path can be regarded as decision sequences in reinforcement learning.

In the literature, many RL methods are related to their graph representations. In PartiGame Algorithm [Moore (1994)], the environment of RL is divided into cells modeled by kd-tree, and in each cell, the actions available consist of aiming at the neighbor cells [Kaelbling, Littman, Moore (1996)]. In Dayan et al. [Dayan and Hinton (1993)], speedup of reinforcement learning is achieved by creating a Q-learning managerial hierarchy in which high-level managers learn how to set tasks for their lower level managers. The hierarchical Q-learning algorithm in Dietterich [Dietterich (1998)] proves its convergence and shows experimentally that it can learn much faster than ordinary "flat" Q-learning. None of these methods, however, can solve the root problem concerning the dilemma of exploration and exploitation.

2.3 Floyd-Warshall algorithm

Denote $SSA(G, e_i, e_j)$ as a shortest path search algorithm that is applied to *G from vertice* e_i *to vertice* e_j that represents a RL. The classical shortest path algorithms like Dijkstra [Dijkstra (1959)] and A* [Hart, Nilsson, Raphael (1968)] are single starting point algorithms for the path-finding. The Floyd-Warshall algorithm [Floyd (1962)] (FW), which is pursued to use in this study, provides the shortest path between any two vertices in specified graph and it is found to be adaptive to the change of the graph.

In the standard Floyd-Warshall algorithm, two matrices (DIST and NEXT) are used to express the information of all the shortest path in the graph. The matrix DIST records the shortest path length between two vertices. The matrix NEXT contains a name of the intermediate vertex through which the two vertices are connected through the shortest path. Because of the optimal substructure property of the shortest path, no matter how many intermediate vertices the shortest path passes through, simply recording one of the intermediate vertices is sufficient to express the entire shortest path.

3 Convergence of exploration and exploitation

In this section, we first define the completely explored graph, which serves as the foundation for a graph-based iterative framework for reinforcement learning. Under this framework, the acquired knowledge from the RL's exploration task can be recorded by the graph, and the shortest path search can then be conducted to determine the next decision chain. This new approach is able to well track the graph changes that are caused by exploration and sometimes by the changing environment. In this section, we shall prove that with this framework involving the shortest path search, the exploration actually converges to exploitation. In a simple word, exploration will find the shortest path to reach the same reward as exploitation does.

3.1 Completely explored graph

Definition 1 A Completely Explored State is a state of which all its possible successor states have already been explored. If one of a state's successor states has been explored and at least one of its successor states has not yet been explored, the state is called a Partially Explored State.

Definition 2 If the vertex set V in connected graph G<V,E> includes the start states of episodes and all these states have been completely explored, and the edge set E represents all the actions that need to be taken to connect all the different states, graph G is called a Completely Explored Graph (CEG).

Fig. 2 shows an example of a CEG where each state(stt_{xx}) is linked with up to 4 possible actions: a_0, a_1, a_2, a_3. There are some explored edges which are omitted for simplicity, such as action a_1 for (stt_{20}) and $(stt_{01}) with\ a_0$; they point to nonexistent state transitions.

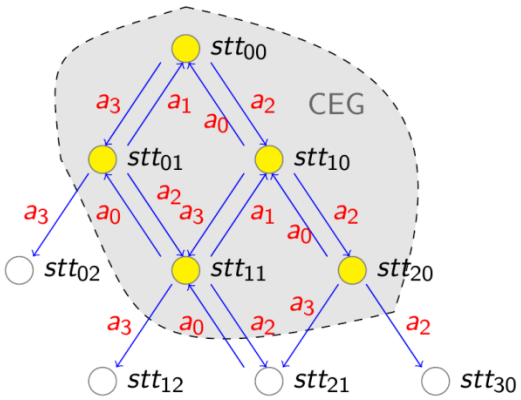

Figure 2: An example of a completely explored graph. Nodes represent states and directed edges between nodes represent actions. The shadowed area that includes all the State nodes (colored yellow) and all the associated directed edges represents the CEG. The unfilled nodes outside the shadowed area represent incompletely explored states, even though they connect to the CEG

Suppose the complete actions set $Actions = \{a_i: a_0, \ldots, a_n\}$ is known. State stt_i is a completely explored state if any reachable next state of stt_i, denoted as stt_{i+1} ,by taking a possible action $a_i \in Actions$, is traversed. Let $GU<V, E>$ represent a graph that contains all the traversed states, including both the completely and the partially explored states. We can define *the predecessor state set* $Set_{predecessor}(stt)$ *as*

$Set_{predecessor}(stt) = \{s: (s \in SE) \wedge (\{s, stt\} \in E(GU))\}$ and the successor state set $Set_{successor}(stt)$ is defined as

$Set_{successor}(stt) = \{s: s \in SE \wedge \{stt, s\} \in E(GU)\}$ where SE denotes the set V(CEG).

If we denote an environment feedback function by *Env*, then for a given action a_k, the next state s_{i+1}can be determined as $s_{i+1} = Env(s_i, a_k)$.

3.2 Exploration converges to exploitation

From CEG, we can prove that exploration converges to exploitation. Here stt_{rwd} is denoted as a reward state.

Lemma 1. Suppose that exploration of each episode starts with state stt_0, and ends in reward state $\mathrm{stt_{rwd}}$ after passing some intermediate states through a series of episodes. Of all the possible episodes, one can see $stt_{rwd} \notin SE$.

Proof: Once the agent has landed in state $\mathrm{stt_{rwd}}$, the episode ends, so there will be no further exploration originated from $\mathrm{stt_{rwd}}$. That is, the graph is a completely explored graph and the states are completely explored states, according to definitions 1 and 2. Thus $\mathrm{stt_{rwd}}$ cannot be a member of the SE set. (End of proof)

Define the envelope set of set SE as $SEE = \{stt_i: (stt_i SE) \wedge (Set_{successor}(stt_i) \not\subset SE)\}$

Corollary 1. For $stt_i \in SEE$ consists of members in SE, there exists at least one of the successor states of $\mathrm{stt_i}$ does not belong to SE.

Proof: It comes directly from the definition of SEE. *(End of proof)*

Corollary 2. If $\mathrm{stt_i} \in (\mathrm{Set_{predecessor}}(\mathrm{stt_{rwd}}) \cap \mathrm{SE})$ is in an exploring episode, then irrespective of the explore strategies adopted in the future, $\mathrm{stt_i}$ will always be part of SEE.

Proof: From lemma 1, one can see that if $stt_i \in SE$ has a successor state stt_{rwd} and it is impossible for stt_{rwd} to be a member of *SE*. By definition of SEE it is known that stt_i is always a member of SEE. *(End of proof)*

Definition 3. If SSA(CEG) plans a decision chain that eventually reaches stt_{rwd}, and it does not produce any change in either V(CEG) or E(CEG), this condition is referred as Exploration Convergence.

Corollary 3. Once the exploration converges, the planned decision chain from SSA(CEG) remains the same in the next exploration episode.

Proof: At the beginning of each episode, a CEG is constructed as needed to explore the new states. If the CEG keeps unchanged at the end of the current episode, the next episode will produce the exactly same path. At this point, the algorithm converges as it satisfies the condition set by Definition 3. *(End of proof)*

Theorem 1. Assume there are a finite number of states and a SSA(CEG) is able to find the shortest path in CEG, exploration becomes finding a path from the start state $\mathrm{stt_0}$ to the $\mathrm{stt_{rwd}}$. In other words, exploration converges to exploitation.

Proof:

i. During exploration, state stt_0 can reach the SEE through SSA(CEG) . That is, one needs to find the shortest path, path *k*, among all the paths, such that:

$$k = \underset{j}{\operatorname{argmin}}\{L(stt_0, stt_j) : stt_j \in SEE)\} \tag{4}$$

where $L(stt_i, stt_j)$ is the length of the shortest path between state stt_i and state stt_j. If $stt_k \in Set_{predecessor}(stt_{rwd})$, then $(stt_0, \dots, s_{ttk}, \dots, stt_{rwd})$ from SSA(CEG) algorithm marks the shortest path from stt_0 to stt_{rwd}. In this case, the conditions concerning exploration convergence (defined in Definition 3) are met, and the exploration converges to the exploitation.

ii. If $stt_k \notin Set_{predecessor}(stt_{rwd})$, CEG continues to evolve as exploration progresses.

iii. As exploration continues, new members are added into *SEE* and they replace the old ones, extending the shortest path, and according to Corollary 2, any new member $stt_i \in (Set_{predecessor}(stt_{rwd}) \cap SE)$ will always be part of *SEE*.

iv. When exploration ends, stt_k that satisfies Eq. (4) will eventually meet the condition: $stt_k \in Set_{predecessor}(stt_{rwd})$.

v. The agent is bounded to pass the state stt_k associated with the shortest path in the $Set_{predecessor}(stt_{rwd}$. If not, there would have a different $stt_{ks} \in Set_{predecessor}(stt_{rwd})$ from stt_k that otherwise makes $L(stt_0, stt_{ks}) < L(stt_0, stt_k)$. If $stt_{ks} \in SEE$, it's impossible for *SSA(CEG)* to choose stt_k as a state in the shortest path. If $stt_{ks} \notin SEE$, there must be a state $stt_m \in SEE$ in

stt_{ks}'s predecessor chain that makes $L(stt_0, stt_m) < L(stt_0, stt_k)$. The algorithm does not converge during this episode.

vi. Putting all things together, one can see that exploration by *SSA(CEG)* must converge to the shortest path from start state stt_0 to stt_{rwd}.

As indicated in corollary 3, once the algorithm has found the shortest path from stt_0 to stt_{rwd}, the path will be repeated with no change in the following episodes. In this case, the exploration is readily to be halted. (End of proof).

4 Algorithm implementation

Based on Theorem 1 described in the previous section, we propose a framework for RL that does not need to concern about the dilemma of exploration and exploitation. There are two major components in the framework, namely *CEG* and Incompletely explored states, and there are two iterative steps as illustrated in Fig. 3:

i. Based on the current *CEG*, an action decision, in the form of a single decision or a chain of multiple decisions, will be made to guide the next exploration.

ii. Update the *CEG* with the new knowledge acquired from the latest exploration. In a static or nearly static environment, exploration will help continue to grow *CEG*, while in a changing environment, *CEG* members can be added or deleted according to the exploration result. Note that when the CEG is updated, nodes or edges can be added or deleted from the graph. In a static environment, as the exploration progresses, the number of nodes and edges tends to increase, while in a dynamic environment, the number of nodes and edges may increase or decrease.

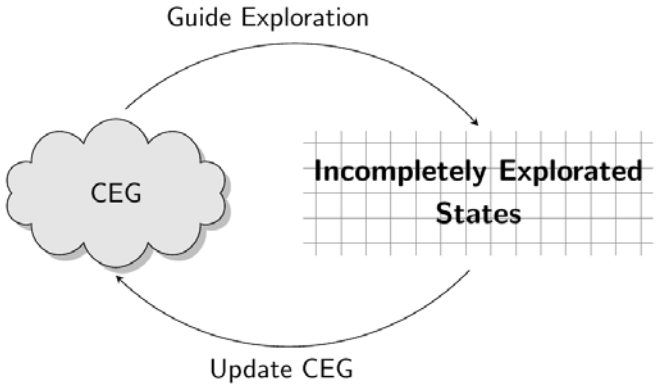

Figure 3: Graph iterative frame

4.1 Shortest path search in dynamic environment

The standard Floyd-Warshall Algorithm calculates matrices DIST and NEXT in batch divided by the length of short path (the number of relay vertices here) for each vertex pair. For constantly changing of vertices and graph structure that engages in exploration all the time, a more efficient method is needed and thus proposed in this section.

During exploration in reinforcement learning, the completely explored states are discovered in sequence, and subsequently, they are added to the CEG, after which the

corresponding edges are also added. In addition, if the agent wants to adapt to the dynamic environment, the removal of vertex must also be taken into account. In this section, a modified algorithm Floyd-Warshall algorithm (SFW) is presented, which is able to search for the shortest path in a graph that represents a dynamic environment. In a simple term, we present SSA(*CEG*) for SFW.

In SFW, each time when a new vertex is added, it is not only to add the shortest path associated with the new vertex directly tied to the two matrices as defined in Floyd-Warshall, but also to compare the length of the new path introduced by the new vertex against that of the shortest path obtained from the prior iteration. These operations may result in the update of the two matrices. The major steps of SFW are summarized follow.

If current state *stt* is to be added to set *SE*, do the following steps:

i. Add and initialize a new row in matrices DIST and NEXT.
ii. Add and initialize a new column in matrices DIST and NEXT.
iii. Update the new column by computing the shortest paths from all the vertices to this new vertex.
iv. Update the new row by computing the shortest paths from the new vertex to all the other vertices.
v. Update matrices DIST and NEXT by comparing the length of each vertices pair between the old shortest path recorded in the matrices and that of the new paths with the new vertex added.

Denote $L_{set}(stt_i, Set_{predecessor}(stt))$ as the length of the shortest path from arbitrary state stt_i to $Set_{predecessor}(stt)$ as defined in Section 2:

$$L_{set}(stt_i, Set_{predecessor}(stt)) = \min_j\{L(stt_i, stt_j): stt_j \in Set_{predecessor}(stt)\} \quad (5)$$

Matrices DIST and NEXT are updated by performing the following operations:

$$DIST(stt_i, stt) = L_{set}(stt_i, Set_{predecessor}(stt)) + 1 \quad (6)$$

$$NEXT(stt_i, stt) = stt_{j_{min}} \quad (7)$$

where j_{min} is the the result of $\min_j\{L(stt_i, stt_j): stt_j \in Set_{predecessor}(stt)\}$ as given in Eq. (5).

In the same token, one can update the stt's predecessor states set $Set_{predecessor}(stt)$ with stt's successor states set $Set_{successor}(stt)$. That is,

$$L_{set}(Set_{successor}(stt), stt_i) = \min_j\{L(stt_j, stt_i): stt_j \in Set_{successor}(stt)\} \quad (8)$$

$$DIST(stt, stt_i) = L_{set}(Set_{successor}(stt), stt_i) + 1 \quad (9)$$

$$NEXT(stt, stt_i) = stt_{j_{min}} \quad (10)$$

Correspondingly, matrix DIST in this case can be updated by

$$DIST(stt_i, stt_j) \leftarrow min\{DIST(stt_i, stt_j), DIST(stt_i, stt) + DIST(stt, stt_j)\} \quad (11)$$

If a path that passes through state stt is shorter than the previously obtained shortest path, then matrix NEXT is updated by:

$$NEXT(stt_i, stt_j) = stt \quad (12)$$

4.2 Guided exploration

As proved in Section 3, exploration finally converges to the shortest path that connects with the target state. Since exploration and exploitation basically produce the same result, our algorithm only needs to consider one single task, exploration.

The steps of how to guide exploration is listed in Tab. 1.

Table 1: Exploration algorithm

01	set the current state to stt, if $\exists a_i \in Actions \Rightarrow stt_{rwd} \equiv Env(stt, a_i)$:
02	return $\{a_i\}$
03	if $stt \notin SE$:
04	$As_{uxpl}(stt) = \{a: Env(stt, a) isunexplored, a \in Actions\}$
05	if $As_{uxpl}(stt) \neq \emptyset$:
06	choose $\{a_k\}$ randomly from $As_{uxpl}(stt)$
07	else:
08	choose $\{a_k\}$ randomly from Actions
09	else:
10	if $stt \in$ SEE:
11	$As_{outside}(stt) = \{a: Env(stt, a) \notin SE, a \in Actions\}$
12	choose $\{a_k\}$ randomly from $As_{outside}(stt)$
13	else:
14	get the nearest edge state $stten$ by SFW from current state stt
15	build the shortest actions chain As_{chain} by SFW from stt to $stten$
16	get $As_{outside}(stten)$
17	choose a_k randomly from $As_{outside}(stten)$
18	$As_{chain} \leftarrow As_{chain} \cup \{a_k\}$
19	return As_{chain}
20	return $\{a_k\}$

There are several major steps in the algorithm listed in Tab. 1:

Step 1: If stt is a neighbor of stt_{rwd}, the agent can take action a_i directly, which transitions the state to stt_{rwd}.

Step 2: If stt is not a member of *SE*, a decision is randomly made by calling $As_{uxpl}(stt)$.

Step 3: If stt is an edge state of *SE*, a decision is randomly made by calling $As_{outside}(stt)$.

Step 4: If stt is a member of SE but not a member of SEE, the shortest path is obtained using SFW. This path represents the decision chain by which the agent can exit SE in the most efficient way.

4.3 Update of the CEG

Before exploration starts, *SE* is empty, the agent has no *a priori* knowledge of the environment. Denote stt_0 as the start state of each exploration episode. Once exploration begins, from the initial state stt_0, for each successor state stt_i, an action a_k is selected from the actions set according to SFW, after which the agent moves to the next state stt_{i+1} by taking action a_k obtained from the feedback of the environment.

Exploration gets repeated. Whenever a new state is found, it will be added to the graph, *GU*, immediately. When the current state is completely explored, it will be added to set *SE*, sometimes to the *SEE* simultaneously. This algorithm is listed in Tab. 2. One can see that when a new completely explored state, corresponding to a vertex in the graph can be added to the CEG, it must generate some action decision reflected as edge changes in the graph. The new SEE by definition can be readily derived from the updated CEG.

Table 2: Algorithm: update CEG

01	Initialize: $SE = \emptyset$
02	Repeat:
03	Agent takes action a_k in state stt_i.
04	Get next state stt_{i+1} from environment.
05	if $(stt_i, stt_{i+1}) \notin E(GU)$:
06	$E(GU) \leftarrow E(GU) \cup \{(stt_i, stt_{i+1})\}$
07	if $\{(stt_i, stt_j): stt_j = Env(stt_i, a_k), a_k \in Actions\} \subset E(GU)$:
08	if $stt_i \notin SE$:
09	$SE \leftarrow SE \cup \{stt_i\}$
10	update CEG
11	update SEE

4.4 Notes on the proposed algorithm

If the current state of the agent is in *SE*, the shortest path to the boundary of the explored region is selected, as seen from the algorithm listed in Tab. 1. As far as the completely explored states are concerned, our approach is able to traverse all of them as opposed to explore them repeatedly. In the classical Q-learning algorithm, there are such a large number of states that have to be traversed repeatedly. This subtle difference makes our algorithm more computationally efficient, as evidenced by the experimental results reported in the next section.

Compared to value-oriented algorithms, experiences in our approach obtained from exploration history are recorded in *GU* and *SE* rather than derived from the value distribution. The establishment of two matrices in SFW relies on *GU* and *SE*, while its structure contains the shortest path of all the pairs of states in SE. Therefore, in the case of changing environment, the modest modification of *GU* and *SE* and the updates of the two matrices will make the proposed algorithm more adaptive to the new, changing environment. This feature can be clearly seen from the result reported in 5.3.

5 Experimental results

The new graph-based algorithm detailed in Section 4 has been applied to solve a maze. Maze solving has been widely adopted for the testing of reinforcement learning algorithms. The agent in the experiment can be seen as a ground robot roaming in a maze, and it can always sense its current position (state) as it moves around. At the beginning of experiment, the agent knows nothing about the maze, and it needs to find the reward(target) position and complete its journey by passing through a path from a specified start position.

5.1 Setup of experiment

The maze has a size of 16 rows by 16 columns for a total of 256 blocks. There are 4 types of blocks, namely target, trap, obstacle and ordinary pass. The fixed start position is treated as a normal pass block. The agent gets a reward of 1 when it reaches the target, but if the agent falls into a trap, it will get a reward of -1. Both conditions will lead to the finish of current episode, and the agent will have to return to the start position and restart its exploration. Note that the agent can keep the exploration information from all the previous episodes. There are 975 mazes in the experiment, and they differ from each other in terms of the locations of the obstacle blocks. In our experiment, there are 46 obstacles for each of the 975 mazes.

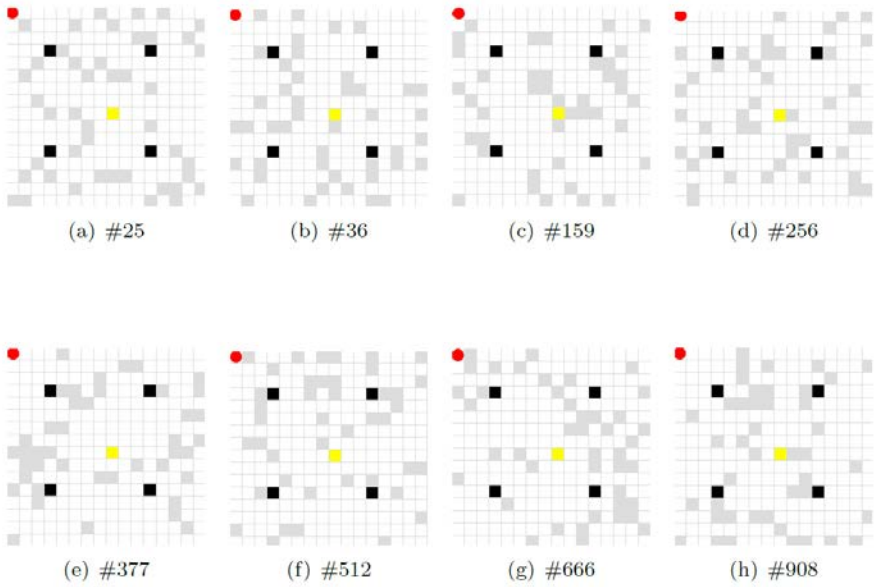

Figure 4: Maze settings

For each maze, the initial position of the agent is at the upper left corner (1,1), and the target position is set to be (9,9). There are 4 fixed traps, located at (4,4), (12,4), (4,12) and (12,12). Tab. 3 summarizes the main characteristics of the maze. Fig. 4 illustrates a sample of mazes, and their respective reference numbers are 25, 36, 159, 256, 377, 512, 666 and 908. In these mazes, the red circle represents the agent, gray blocks represent

obstacles, black blocks represent the traps, yellow block represents the target, and the rest are normal pass blocks.

Table 3: Maze design parameter

Parameter	Value
Map Size	16×16
Map Amount	975
Target Amount	1
Trap Amount	4
Rate of Obstacles	0.18
Total number of Episodes	100

The proposed algorithm, referred as SFW, is compared against the classical Q-learning algorithm (ql) and an improved Q-learning algorithm (qlm).Tab. 4 tabulates the main parameter values for ql. The main improvement of qlm over ql is that qlm can remember the locations of the obstacles and traps found during exploration, and avoid them during the subsequent explorations. Even if the next action is randomly selected based on some probability, qlm can filter out the obstacles and traps.

Table 4: Q-learning parameter

Parameter	Value
Learning Rate(α)	0.01
Discount (γ)	0.9
$\epsilon-$ greedy	0.9

5.2 Performance comparison with Q-learning algorithms

5.2.1 Single maze comparison

All three algorithms are compared in terms of number of steps per episode when they are applied to solve all the mazes, and the results from mazes 25 and 908 are plotted in Fig. 5 and Fig. 6, respectively. In solving both mazes, SFW is found to converge more quickly than the other two algorithms, and it requires less number of steps during the exploratory process. As expected, qlm's performance is better than that of classical ql.

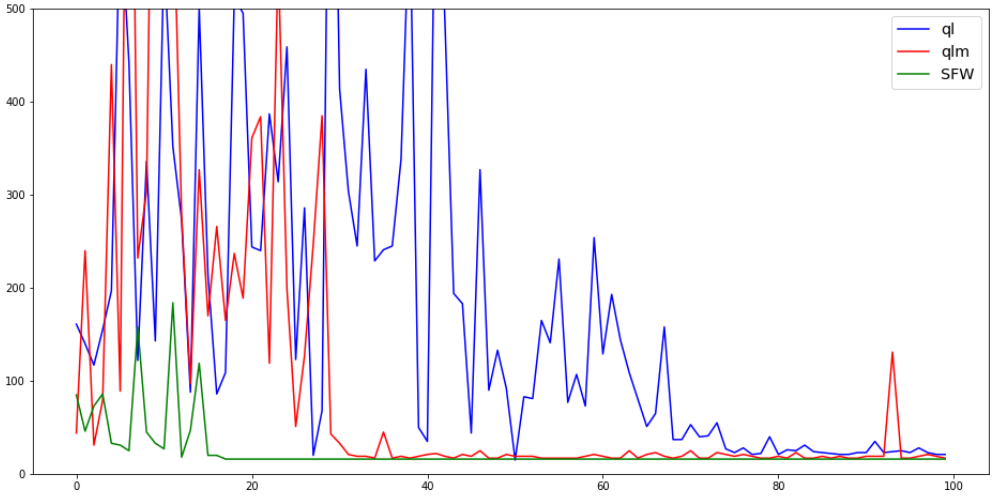

Figure 5: The steps amount in the #25 maze per episode comparison. The X axis is the episode number, and the Y axis represents the number of steps in each episode

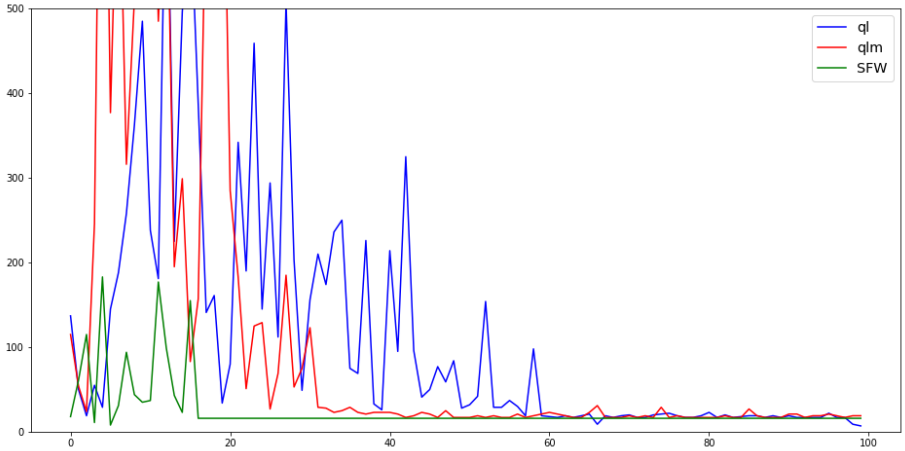

Figure 6: The number of steps in the #908 maze per episode

All three algorithms are compared in terms of average exploration efficiency of each step in every episode when they are applied to solve all the mazes, and again, the results from mazes 25 and 908 are plotted in Fig. 7 and Fig. 8, respectively. One can see that SFW is more efficient in exploration and converges faster than the other two algorithms.

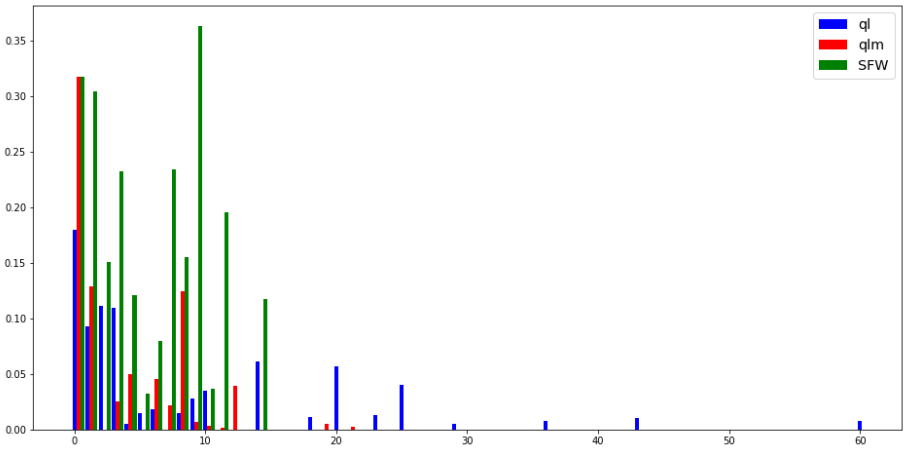

Figure 7: Average exploration efficiency while solving maze 25

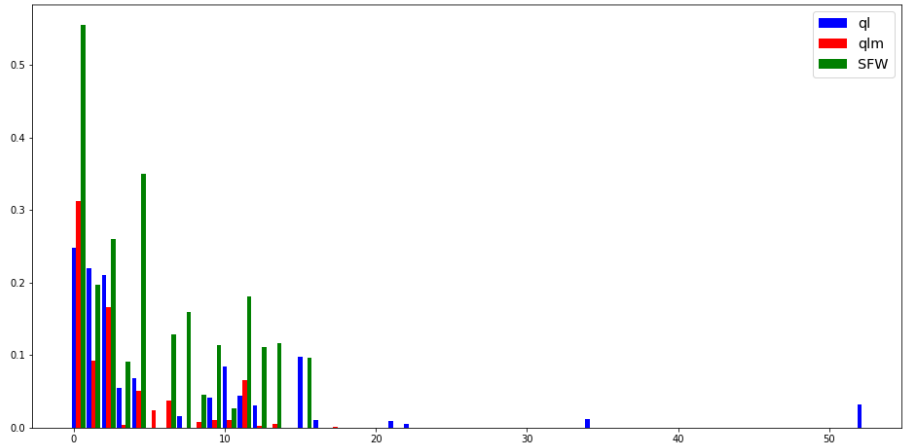

Figure 8: Average explore efficiency while solving maze 908

5.2.2 Statistical performance comparisons for all mazes

The experiments in this section include all 975 mazes. The X axis of each figure corresponds to the maze number.

The comparison of convergence speed of every maze is shown in the Fig. 9 and Fig. 10, where ql and qlm are compared with SFW separately. The Y axis of each figure is the number of episodes when the agent for the first time arrives at convergence. In both figures, one can see that the proposed algorithm converge more quickly than the other two algorithms, especially true when there are a large number of episodes. Actually, the SFW alogirthm converges after no more than 20 episodes, while the other two algorithms need as many as 100+ episodes.

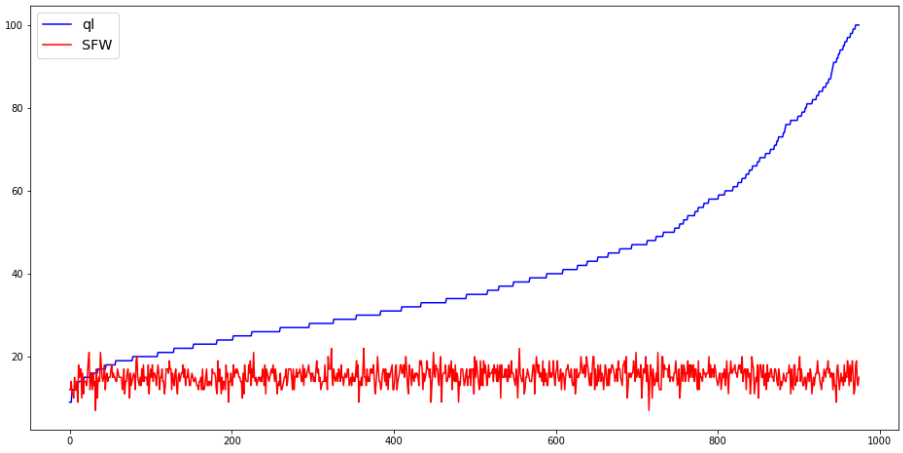

Figure 9: Convergence speed comparison: ql and SFW

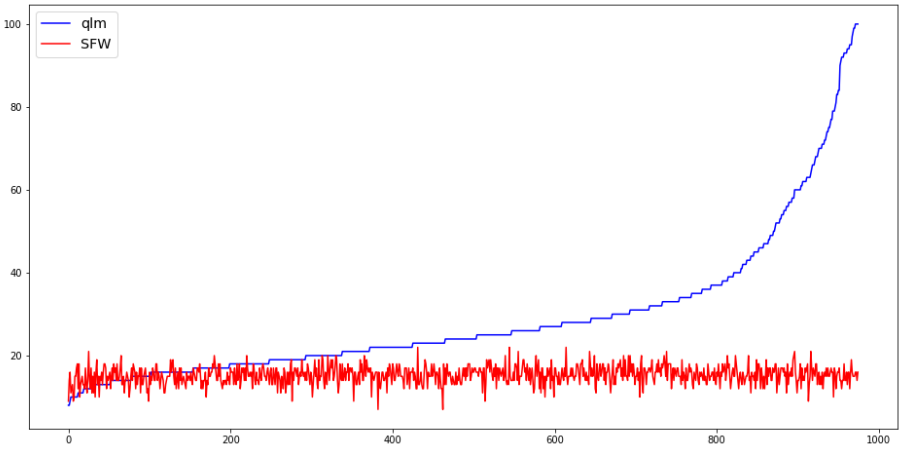

Figure 10: Convergence speed comparison: qlm and SFW

The lengths of the finally discovered paths by are reported in Fig. 11 and Fig. 12. One can see that the paths found by SFW have shorter lengths, in the range of 15 to 20, while the paths found by the other two algorithms are much longer. In quite a few cases, the paths found by ql and qlm are twice longer than those found by SFW.

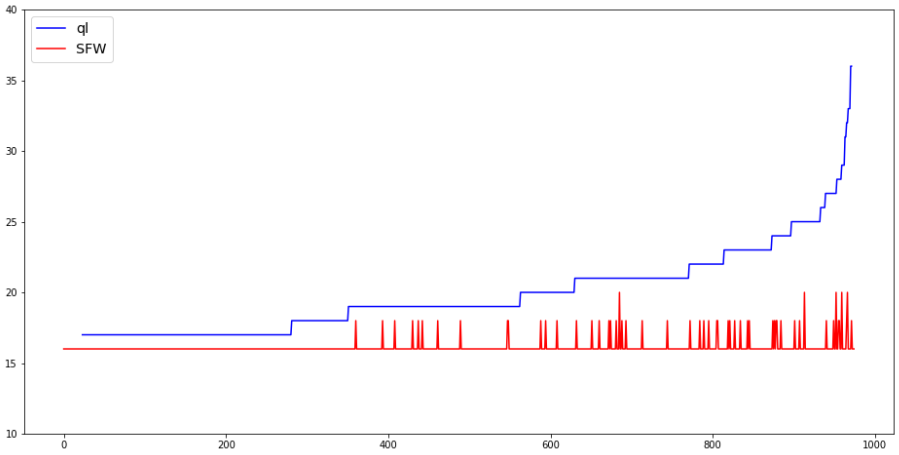

Figure 11: Steps length of convergence comparison: ql and SFW

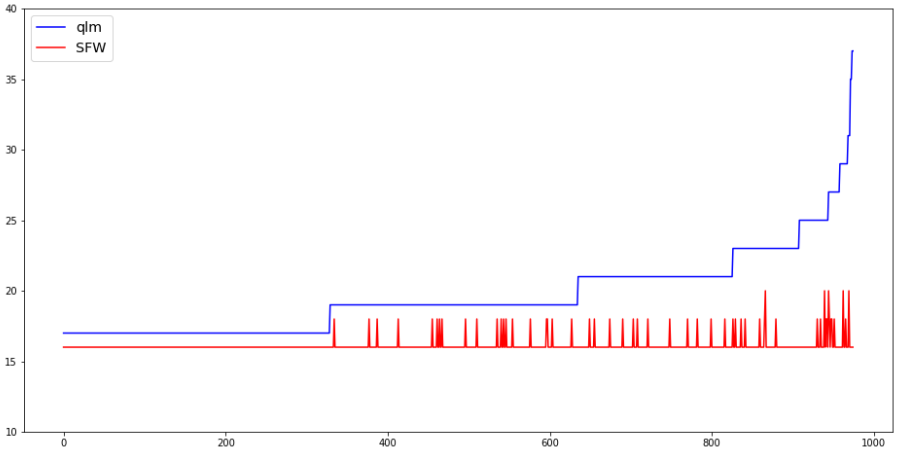

Figure 12: Steps length of convergence steps comparison: qlm and SFW

The exploration efficiency obtained from solving every maze is shown in Fig. 13. One can see SFW outperforms qlm in this regard, and both algorithms are significantly better than ql. The X axis represents the maze number. The Y axis is the ratio of the total number of explored states to the total number of steps when the agent for the first time arrives at convergence.

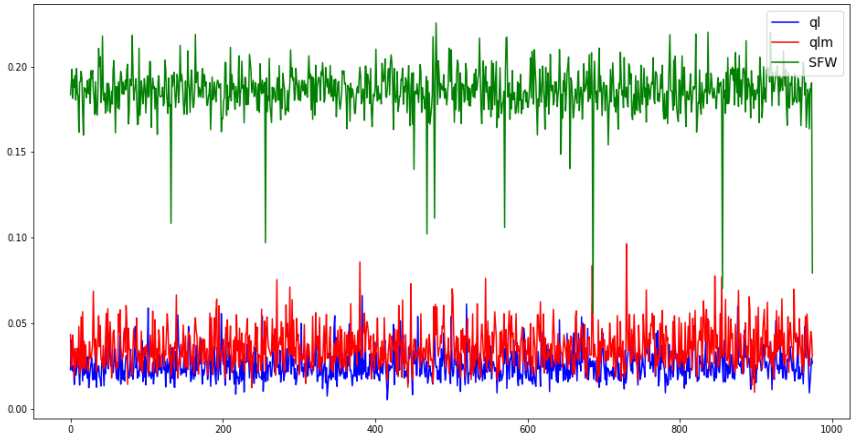

Figure 13: Explore efficiency comparison

5.3 The maze in the dynamic environment

Changes of environment are categorized as obstacle change and target position change. We will in this subsection examine how these changes can affect the performance of the three algorithms.

5.3.1 Obstacle change

Take Maze #8 as an example, the changes of the obstacles are tabulated in Tab. 5.

Table 5: Dynamic obstacle, Maze #8

Change episode	Change state	Convergence episode #	The corresponding figure
0	Init	16	Fig.14(a) and 14(c)
18	(2,8)	18	Fig. 14(d)
24	(3,6), (3,7)	25	Fig. 14(e) and 14(f)
29	(9,8), (10,8)	34	Fig. 14(g) and 14(h)

Fig. 14 shows the snapshot of exploration, convergence, environment change and adaptation. The maze has undergone three major changes that occur to the locations of the obstacles in the experiments. The green squares in Fig. 14 represent the members of *SE*, and the purple squares represent dynamically increased obstacles that are located within the current convergent path.

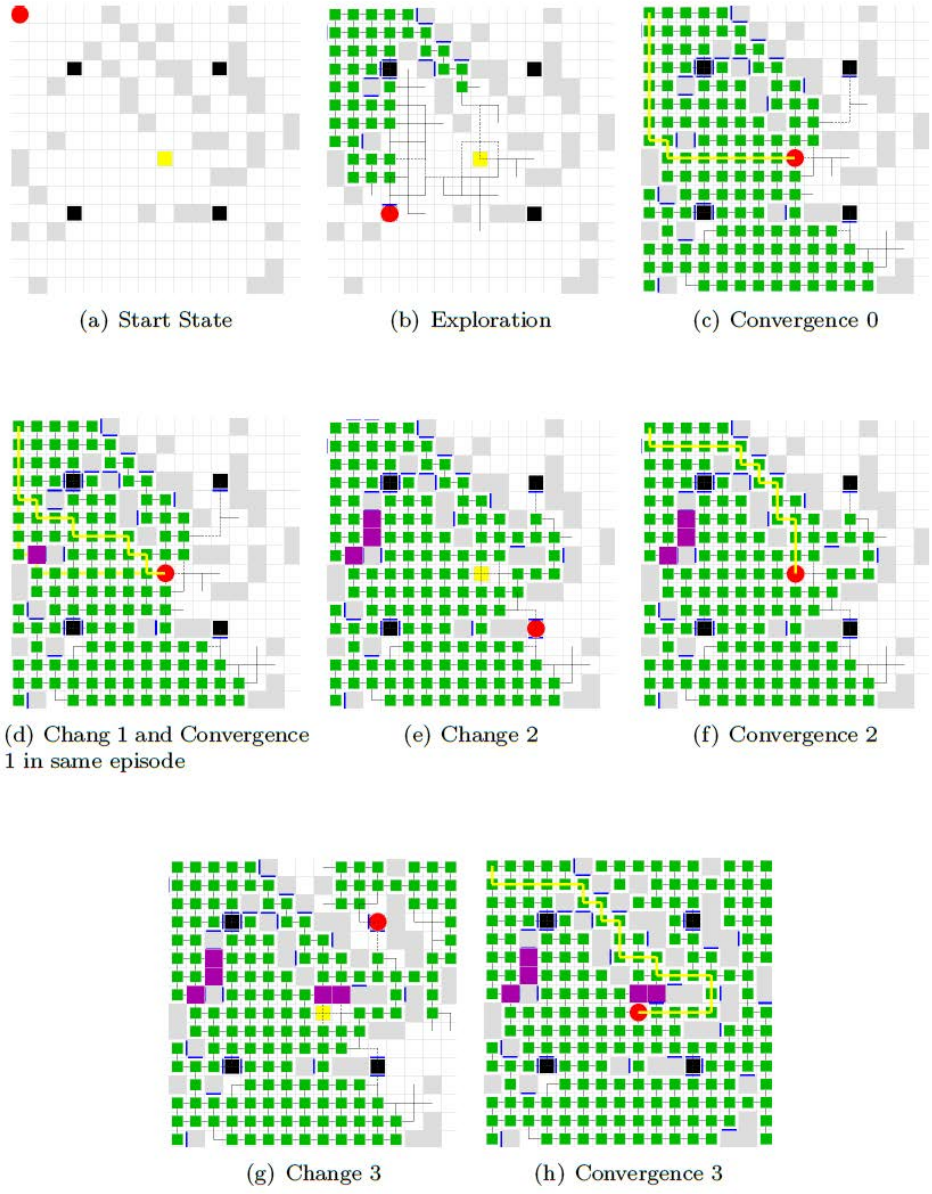

Figure 14: Dynamic obstacles, Maze #8

5.3.2 Changes of target positions

Tab. 6 summarizes the changes that occur to maze 243. Other mazes have gone through similar changes. One can see that the target position is changed once, relocated from the center of the maze to its lower left corner.

Table 6: Dynamic target, Maze #243

Change episode	Change state	Convergence episode #	The corresponding figure
0	Init	14	Fig. 15(a) and 15(c)
18	(9,9) ⇒ (1,16)	21	Fig. 15(d) and 15(e)

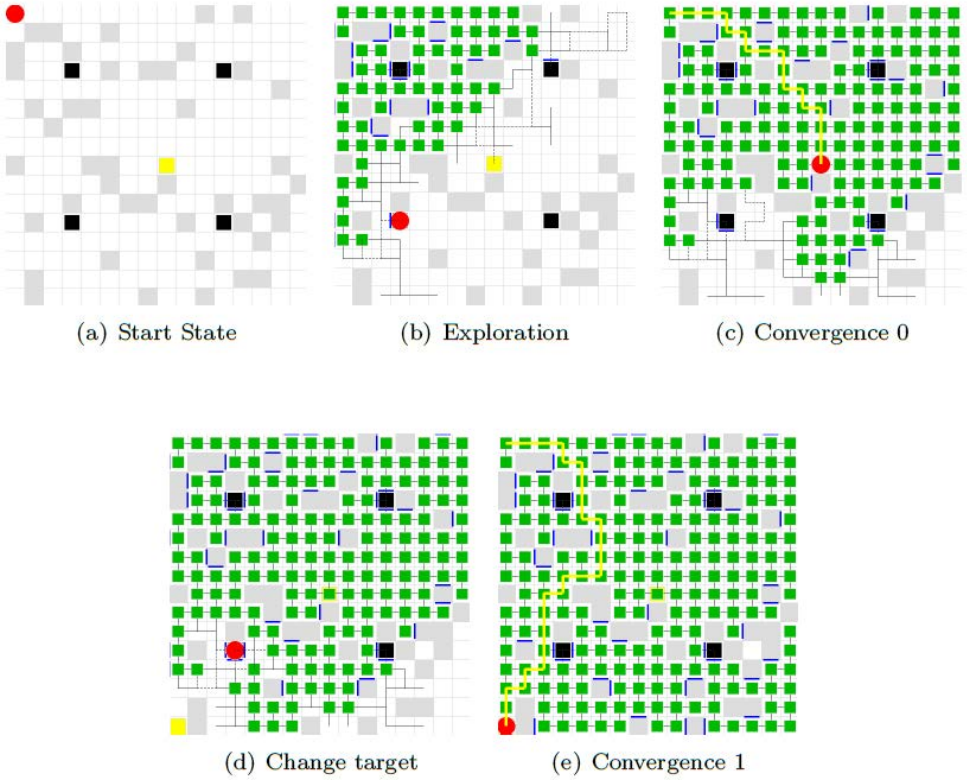

Figure 15: Dynamic target, Maze #243

The results of exploration, convergence due to target position changes and re-convergence are shown in Fig. 15. In episode 0, the reward is claimed at block (9, 9) (Fig. 15(a)). The exploration converges in episode 14 (Fig. 15(c)). In episode 18 where move the reward is moved to the block (1, 16) (Fig. 15(d)), the agent finds the right path to the target, after three episodes (Fig. 15(e)).

5.4 Computation efficiency

All three algorithms are compared for their respective computation efficiency under the same computation platform. The hardware used in the experiments has a Intel(R) Core(TM) i5-3210M CPU running at 2.50 GHz, and a RAM size of 8 GB. The operating

system is Ubuntu 64bits. The tools used to test CPU time and memory occupation are line_profiler and memory_profiler, respectively.

The average CPU time reported in Tab. 7 is the average time of solving all 975 mazes. The basic memory usage in Tab. 7 refers to the stable memory usage collected from solving select 62 mazes. One can see that SFW requires more memory space than the other two algorithms; the memory usage for both ql and qlm is comparable. The peak memory usage of SFW is also higher than that of ql or qlm.

Table 7: Algorithm complexity comparison

Parameter	ql	qlm	SFW
Average CPU Time (s)	0.5382	0.8472	4.9307
Basic Memory usage (MB)	60.434	60.352	70.148
Peak Memory usage (MB)	61.024	61.500	74.135

6 Conclusion

In this paper, a new graph-based method was presented for reinforcement learning. Unlike classical Q-learning algorithm and improved Q-learning algorithm, the proposed algorithm does not struggle with the exploration vs. exploitation tradeoff, as it was proved that the two tasks of exploration and exploitation actually converge in the decision-making process. As so, the proposed graph-based algorithm finds the shortest path during exploration, which gives higher efficiency and faster convergence than the Q-learning algorithm and its variant. Another big advantage of the proposed algorithm is that it can be applied to the dynamic environment where the value-oriented algorithm fails to work. The efficiency and convergence performance of the proposed algorithm comes at a cost of increased computational complexity. Future study will be focused to confine the computational complexity and particularly memory usage.

Acknowledgements: This research work is supported by Fujian Province Nature Science Foundation under Grant No.2018J01553.

The authors would like to thank the U.S.DOT Tier 1 University Transportation Center on Improving Rail Transportation Infrastructure Sustainability and Durability.

References

Dayan, P.; Hinton, G. E. (1993): Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, pp. 271-278.

Dietterich, T. G. (1998): The maxq method for hierarchical reinforcement learning. *ICML*, vol. 98, pp. 118-126.

Dijkstra, E. W. (1959): A note on two problems in connexion with graphs. *Numerische Mathematik*, vol. 1, no. 1, pp. 269-271.

Floyd, R. W. (1962): Algorithm 97: shortest path. *Communications of the ACM*, vol. 5, no. 6, pp. 345.

Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968): A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100-107.

Huang, B. Q.; Cao, G. Y.; Guo, M. (2005): Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance. *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, vol. 1, pp. 85-89.

Kaelbling, L. P.; Littman, M. L.; Moore, A. W. (1996): Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, vol. 4, pp. 237-285.

Lewis, F. L.; Varbie, D.; Vamvoudakis, K. G. (2012): Reinforcement learning and feedback control: Using natural decision methods to design optimal adaptive controllers. *IEEE Control Systems*, vol. 32, no. 6, pp. 76-105.

Lewis, F. L.; Varbie, D. (2009): Reinforcement learning and adaptive dynamic programming for feedback control. *IEEE Circuits and Systems Magazine*, vol. 9, no. 3, pp. 32-50.

Mataric, M. J. (1997): *Reinforcement learning in the multi-robot domain.* Robot Colonies Springer Press, pp. 73-83.

Moore, A. W. (1994): The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Advances in Neural Information Processing Systems*, pp. 711-718.

Smart, W. D.; Kaelbling, L. P. (2002): Effective reinforcement learning for mobile robots. *IEEE International Conference on Robotics and Automation*, vol. 4, pp. 3404-3410.

Sutton, R. S.; Barto, A. G.; Williams, R. J. (1992): Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems*, vol. 12, no. 2, pp. 19-22.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M. et al. (2017): Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv*, preprintarXiv:1712.01815.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A. et al. (2017): Mastering the game of go without human knowledge. *Nature*, vol. 550, no. 7676, pp. 354-359.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L. et al. (2016): Mastering the game of go with deep neural networks and tree search. *Nature*, vol. 529, no. 7587, pp. 484-489.

Sutton, R. S.; Barto, A. G. (2017): *Reinforcement Learning: An Introduction.* MIT Press.