# Reinforcement Learning for Solving the Pursuit-Evasion Problems

Chieh-Yang Huang, Shi-Hao Liu, Vince Chiang, and Edward Yang
School of Computing, Informatics, Decision Systems Engineering.
Arizona State University, Tempe, AZ, USA
Email: {chiehyang.huang, shihao.liu, vchiang, eyang12}@asu.edu

***Abstract*—In this paper, we consider the worst case of Pursuit-Evasion Problems. In this case, there are unknown numbers of evaders and those evaders can move arbitrary fast. In order to solve it, this problem is formulated as a clear-region problem, where the clear region would be recontaminated if there is a available path between the clear region and the contaminated region. Though applying brute-force search algorithm is the simplest way to solve it, the state space could be too large to solve. Therefore, Ramaithitima et al. proposed an idea of building an abstract state space in order to reduce the state space. However, building the abstract state space still takes a lots of time and effort. As a result, we proposed a reinforcement learning approach for finding a sequence of movement to clear the region. By learning the utility of the path, experiments show that our approach could solve the clear-region problem.**

## I. Introduction

In this paper, we focused on the worst case of the pursuit-evasion problem. In this case, an unknown number of evaders is presented in the given environment. All of the evaders could move arbitrary fast and the movement is unpredictable. Our goal in this problem is to have a policy that can make sure the pursuers can capture all of the evaders. Therefore, to address this problem, we model our environment by graph structure and then formulate it as a clear-region problem. As shown in Fig. 1, the red region is contaminated but once the pursuer checks the region, the region turn in to a clear region (green) which means there is no evaders. However, after the pursuer passes through the branch, the clear region gets recontaminated again since there is a path from the contaminated region to the clear region, and as our assumption, the evader could run arbitrary fast so the region we just cleared is now not clear anymore. As a result, in this case, we would need two pursuers to cooperate with each other. One pursuer stays at the branch top to protect the clear region and the other one keeps moving to clear the rest of the region.

The simplest way to solve this problem is applying a search algorithm such as DFS and BFS. However, in order to consider both the contaminated status and the positions of pursuers, the state space grows exponentially as the environment gets larger. As a result, Ramaithitima et al. [1] proposed an abstraction framework to partition the configuration space into sets of similar configurations, abstract state, to reduce search space. The idea is shown in Fig. 2a. As the figure shows, after building the abstract state space, we can apply search algorithm on this space to find a path that leads us to clear all the region. The last step is to transform this path back to the real state space. However, it turned out that building the abstract state requires to go through all of the real state space to determine, whether the abstract space corresponds to the real state space. Therefore, the irrational building time of the state space has motivate us to come up with a method that can prune extraneous state space instead of going through all of the possible states.

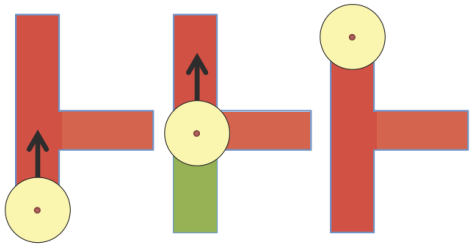


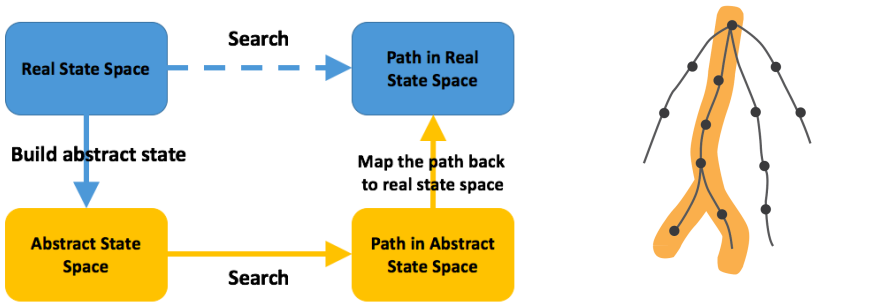
Fig. 1: The red color represents the contaminated region and the green color represents the clear region. Therefore, our target is to turn all the environment into green.

In this paper, we introduce an approach of using reinforcement learning to help us decide whether a state is desirable or not needed for further exploring. One advantage we can benefit from reinforcement learning is that it can automatically learn how to evaluate a state. Therefore, if we could set up a reasonable reward function, the reinforcement learning agent will be able to find a path to reach our goal, which is to clear up the region. The idea of our approach is illustrated in Fig. 2b.

The rest of the paper is organized as follows. In section II, we will briefly describe the related works of solving the pursuit-evasion problem and applying reinforcement learning on searching problem. In section III, we will introduce the benchmark algorithm. The benchmark result will be discussed in section IV. Then, in the section V and VI, we will explain the detail of our proposed approach and the result respectively. Last, the conclusion will be in the section VII.

## II. Related Works

The Pursuit-Evasion problem has been studied for decays. Researchers ranging from different fields focused on different aspects and provided their own approaches, including graph-based method [2, 1], visibility-based search [3], and probabilistic search [4].




(a) The work flow of the benchmark algorithm.

(b) How reinforcement learning search.

Fig. 2: The idea of the benchmark algorithm (a) and the idea of using reinforcement learning to solve this problem (b).

Parsons [2] proposed a problem to search a lost man in a dark cave. Though a party of searchers who is familiar with the cave is trying to find him, the lost man could move unpredictable and arbitrary fast and thus a special algorithm is needed to make sure we could find the lost man. Parsons [2] formulated this problem into a graph based problem and analyzed the number of searchers according the structure of the cave. However, in the real word, though we could find someway to transform a continues map into a graph based map structure, the graph is usually much more complicated and analyzing by hand is therefore infeasible.

Hollinger et al. [3] focused on the similar problem where a man is lost in a museum, office or supermarket and our goal is to find him. Comparing to Parsons [2]'s work, each room is treated as a node and once the room is occupied, it is cleared. That is, once we occupied a room, then we could conclude that the lost man is not in the same room. A Guaranteed Search with Spanning Trees (GSST) is proposed to solve this problem. However, the algorithm works only on a tree structure and thus the unstructured environment is still unsolvable.

In another case, Hespanha et al. [4] proposed a probabilistic method but what they were working on is a relaxed problem. In their setting, the evader could not move arbitrary fast, hence capturing the evader directly is possible. As a result, they proposed a greedy policy to control the pursuer. In their method, the pursuer will move toward the location where evader is most likely to be found. They also show that under some specific assumption, this policy could guarantee a finite steps solution.

Some other kinds of pursuer-evader problem are also well studied. One of the well known pursuer-evader problem is the Timed-Road Algorithm [5]. The basic problem is defined as measuring whether a single pursuer $P_0$ could eventually collide with a single evader $E_0$ or not. At the beginning, the Timed-Road Algorithm lists all the position state of $(P, E)$, and starts updating the time-stamp of each state. Finally we use the steps of each state to verify whether the pursuer $P_0$ could catch the evader $E_0$ at some certain state. The algorithm can be implemented with dynamic programming but with dimensions increasing, it will also increase the time-complexity.

Another kind of pursuer-evader problem is a game of cops and robbers [6]. Starting with a single cop and a single robber, we aim to find a graph where cops always wins meaning that cops can always collide with robbers. Later, we extend the problem to n-cops and n-robbers. A graph that cops always win is a tree which a cop go though a unique path and the robbers' area reduced monotonically and robbers' strategy will be stay at the original position (passive strategy) or they can move to adjacent position (active strategy). The algorithm servers a good purpose to find whether a graph can be cops-win graph and during the traveling of the cops, we can to remove the pitfalls and after certain amount of time, we reduce the graph into the single vertex in order to claim the winning of cops. The problem is that the robbers can stay at the same place which makes the problem more easily, and we want to handle more complicate problems that robbers will move extremely fast so that we need a good strategy to make sure we can catch all the robbers.

On the other hand, some researches were focusing on applying reinforcement learning to solve the search problem. Many works [7, 8, 9, 10] have shown that it is possible to solve this kind of problem using reinforcement learning.

In 2004, Şimşek and Barto [7] presented Relative Novelty Algorithm (RN) to identify useful temporal abstractions in reinforcement learning. The concept of RN can be used while partitioning the configuration state. Similarly, Şimşek et al. [8] using Local Graph Partitioning to identify useful sub-goal in reinforcement learning. Since we are proposing reinforcement learning method to solve the worse case Pursuit-Evasion Problems, we can introduce both concept of RN and Local Graph Partitioning while doing reinforcement learning in order to avoid unnecessary state exploration.

Moreover, Dai et al. [9] has been ongoing for unique combination of reinforcement learning and graph embedding to address three challenges Traveling Salesman Problem, Minimum Vertex Cover, Maximum Cut. By using a greedy policy that behaves like a meta-algorithm that incrementally constructs a solution [9], and the action is determined by the output of a graph embedding network capturing the current state of the solution. Remarkable part of this paper is the algorithm of using Q-learning for the graphs to be traverse. Learning value Q is stated as a utility across a set of m graphs

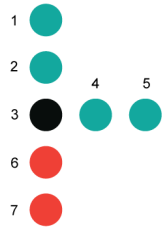



Fig. 3: A sample case of the real state. The two green regions are clear and the red region means it is contaminated . The black node represents the pursuer.

from distribution set D, each transition will result in state which is a series of graph based outcome, with the reward and cost defined according to different challenges, the policy is to learn the maximum reward according to the outcome states from the action given. As to solving the problem of a map which contains only the number of vertices but unclear of how the map is constructed, using reinforcement learning introduced in this paper can ensure the policy would give an outcome that has minimum cost but with a cost of maintaining a complex graph for traversing.

To view on another approach of pursuit-evasion problem, [10], the paper introduces a way of a new reinforcement learning strategy. Same as the pursuit-evasion problem, the methods for the pursuers not only includes capturing the prey, but also requires the pursuers to surround the prey in a 2D-graph. Hence we found something interesting in this paper as the pursuers have to determine whom have to take the positions North, South, East and West. By taking the position into a determining factor, after a finite amount of training the pursuers have learned that each position should be always taken by specific ones. But due to the restrictions in the paper, the graph utilized by the paper is simply a fully connected 2D array graph and the prey act as a easily predictable target since it can only move one step at a time. Therefore, while dealing with more complicated multiple branch connected graph, we have to put more instances into consideration, while determining how can the continuously changing graph be traversed due to the arbitrary fast prey, we take into account the pursuers could utilize a policy to determine the directions to take in order to prune the possibilities that could be generated.

## III. Benchmark Algorithm

In this section, we will first describe the real state and the problem definition. Knowing this could help us understand the idea of abstract easily. Then we will describe the benchmark algorithm in detail, including building the abstract state space, searching in abstract state space, and transforming the abstract state space path back to the real state space.

### A. Real State Space

The real state space contains a graph structure [1] representing the environment as shown in Fig. 3. To represent a real state, basically, we need two kinds of information, the positions of pursuers and the contaminated status of the map. Therefore, given the environment graph $G$, a real state could be represented by $(\mathbf{p}, \mathbf{d})$, where $\mathbf{p}$ is a vector of the pursuers' positions and $\mathbf{d}$ is a vector of the contaminated status of all nodes. Taking the state in Fig. 3 as an example, it could be represented by $(\{3\}, \{0, 0, 0, 0, 0, 1, 1\})$, where 3 is the position of the pursuer and the **0**s and **1**s in $\mathbf{d}$ means clear and contaminated respectively. Given the above definition, the problem can be defined as follows: As we can see, since the definition is quite comprehensive, it is actually very intuitive to apply search algorithm such as depth-first-search or breadth-first-search to solve this problem. However, it is also easy to notice that the state space is quite large and will grow exponentially especially when we need more pursuers to solve the problem. In Ramaithitima et al. [1]'s work, they provided a baseline called "baseline planner" but didn't explain the detail of it. Therefore, in our implementation, we utilize breadth-first-search and refer it as the **Brute Force** algorithm.

Problem Definition 1.

**Given:** A environment graph $G$ and an initial state $(\mathbf{p}_0, \mathbf{d}_0)$, where all $\mathbf{p}_0$ are initiated to 0 and $\mathbf{d}_0 = \{0, 1, 1, \cdots, 1\}$.

**Output:** A sequence of action that can lead us to a terminal state $(\mathbf{p}_t, \mathbf{d}_t)$ where $\mathbf{d}_t = \{0, 0, 0, \cdots, 0\}$.

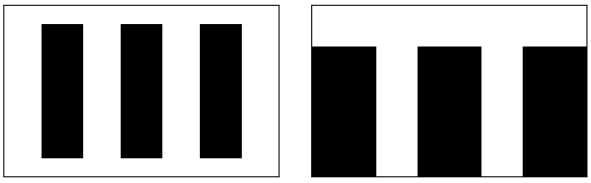
### B. Abstract State Space

The benchmark algorithm which aims to reduce the state space is dividing into two parts. The first part is **Partition Algorithm** which uses the configuration of the real state to build the abstract states. The configuration of the real state represents only the position of pursuers and the contaminated status is not included here. The abstract state could be regarded as a set containing one or more real states. The second part is running the **Search Algorithm** in the abstract state space. Since we have greatly reduced the state space by merging multiple real states into one single abstract state, this could be much easier and faster than doing search algorithm directly in the real state space.

For the **Partition Algorithm**, we are going to build the abstract state from the real state. Notice that in the partition step, we only consider the pursuers' positions and the contaminated status of the node is not in our consideration [2]. This could potentially reduce the search space. We start from the initialized state which all the pursuers stay at Node 0, and we will try to move the pursuers one by one to find the next possible real state (pursuers' position). During searching for next pursuers, we will also find the connected components for each real state. The connected components represent the regions that are separated by the pursuers. The author defines that two real states belong to the same abstract state if and only if there exists a bijection mapping between their connected components. By iterating all the configurations, we could then cluster these configurations into a abstract state space.

[1]To solve the problem in a continuous space, we can first apply a simple algorithm to transform it into a graph-based space and then utilize either the benchmark algorithm or our proposed method to solve it.

[2]The contaminated status will be added as a information when we are doing search algorithm later.



(a) Ladder with 4 steps. (b) Tree with 2 branches.

Fig. 4: Two kinds of maps. The white region is the road that pursuers and evaders could go. The black region is the wall.

After building the abstract state, we can now do the **Search Algorithm** by applying breadth-first-search on it to find a available path. The state here would be $(s, L)$, where $s$ is the abstract node and $L$ represents the contaminated connected components. To test if we reach the goal or not, we can simply check whether $L$ is an empty set or not. If $L$ is an empty set, this means there is no contaminated region so we successfully clear the environment. However, one challenge here is to propagate the contaminated state $L$. In our understanding, $L'$ would be clear if we could find a mapping from it to $L$. After finding a path on abstract state space, we need to transform it back to the original state space. This phase is called **Refinement**. Given a path on abstract space $\{S_1, S_2, ..., S_n\}$, our goal is to find a path between every neighbor states such as $(S_1, S_2)$ and $(S_k, S_{k+1})$ with the start pursuer configuration (real state) within the first abstract state such as $S_1$ and $S_k$ in the above example. In this problem, we need to do breadth-first-search within the first abstract state. For nodes that is not in the first abstract state, we just skip it. However, once we find a node in the target abstract state, the goal is achieved.

## IV. Benchmark Result

### A. Dataset

Since Ramaithitima et al. [1] didn't provide their data-set, we build the data-set by ourselves. There are two classical types of environment, ladder and tree, as shown in Fig. 4. For both of structures, the numbers and width of steps and branches could vary and thus give us several environment settings. This map contains way more loops and thus require much more pursuers to finish the task. Notice that we assume the pursuer have a disc sensor footprint with a radius of one unit. The unit is also used in determining the width of the steps or branches. For example, when a tree environment is build up with two branches, then each branch would be two units width which requires at least two pursuers to fill up.

After building the map, we need to represent the environment by graph notation as Ramaithitima et al. [1] illustrates. The first thing we need to consider is, the graph have to at least cover all of the point in the environment. The second property is optional, which is to reduce the number of graphs. The second is the classical minimal set cover problem and is proven to be a NP-complete problem. Hence, here we only focus on implementing the first property by a sampling algorithm. To do so, we first scatter graph center uniformly on the map and then keep those on the white region. Within the white region that is not covered by any graph node, we choose one point closest to the discarded graph centers as a new graph center. By doing so, we could eventually cover all the point in our environment. Besides, to decide whether a connection exists between two graph nodes or not, we examine if they have intersection first and then check if a transition path exists on the map. However, there are still some incorrect transition path. Though having more transition path is okay (it will be more difficult to solve, but the result of clearing region is guarantee), we manually remove it to form a more simple graph.

| Configuration | Brute Force | Benchmark |
|---|---|---|
| ladder, k=2, w=1 | 0.03 | 1.873 |
| tree, k=1, w=1 | 0.015 | 0.02 |
| tree, k=1, w=2 | 1.34 | 0.561 |
| tree, k=4, w=1 | 0.2 | 0.14 |

TABLE I: The execution times of Brute Force and Benchmark.

### B. Result and Comparison

In our experiment, we build two approaches, benchmark algorithm and brute force algorithm. As Ramaithitima et al. [1] reveals, it is very hard for brute force algorithm to find a path. In their data, the brute force algorithm takes 1082.77 seconds for a ladder environment with 2 steps and 14 vertices. However, in our brute force implementation, the result is actually much faster. On the other hand, our implementation of benchmark doesn't get much acceleration. Some execution time are shown in TABLE I for comparison. We also plot two result paths in Fig. 5. As we can see, both the brute force algorithm and the benchmark algorithm can reach the goal.

## V. Extensions with Reinforcement Learning

To apply reinforcement learning to solve this problem. First, we build the environment which contains the information of reward then build the agent with Q-learning [11]. In order to obtain a better result, we took the state representation into consideration as to find a new way to achieve the solution. In the following section, we will describe these in detail.

### A. State

In section III-A, we have described how to represent the state in an intuitive way. However, when more than one pursuer exist in the environment, we will need some redundant states to represent them. As Fig. 6 shows, if we have two pursuers in this environment, the left figure would be $(\{3, 3\}, \{0, 0, 0, 1, 1, 0, 0\})$ but the right figure could be represent by two states, $(\{3, 3\}, \{0, 0, 0, 0, 0, 1, 1\})$ and $(\{3, 4\}, \{0, 0, 0, 0, 0, 1, 1\})$. The redundant states grow exponentially as more pursuers are added into. Therefore, we proposed a new way to represent the state in order to remove these redundant states.

The state is now represented by **graph node status**. The number of the corresponding node represents different states of it. We use a **-1** to represent a contaminated node, a **0** to represent a clear node, and a positive number to represent the number of the pursuers. Let's take the same case provided in Fig. 6 as an example. The left figure is now represented as $\{0, 0, 2, 0, 0, -1, -1\}$, where the number 2 at the third element means there are two pursuers occupying the third

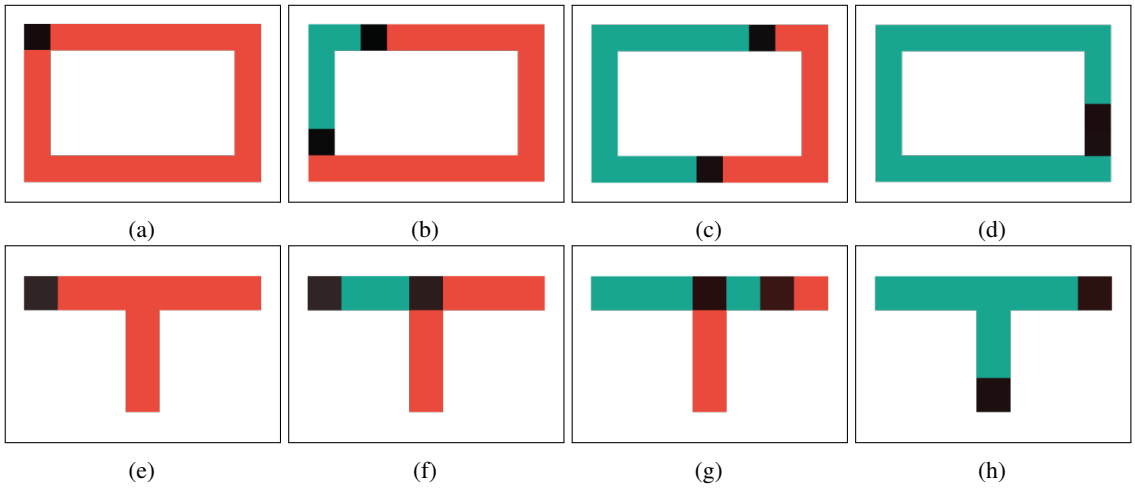



Fig. 5: Result path. (a)-(d) is the result of brute force algorithm. (e)-(h) is the result of benchmark algorithm.

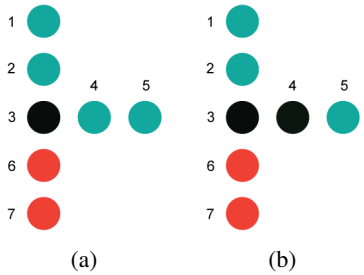



Fig. 6: Example of the redundant state representation.

node. On the other hand, the right figure is now represented as $\{0, 0, 1, 1, 0, -1, -1\}$ and there is only one way to represent it so no redundant states will exist when using our graph node status to represent the state. By applying the graph node status as our state, the definition of the terminal state also need to be modified and is now a state $s_t$ where no $-1$ exists.

### B. Environment and Reward

Building the environment is one of the most important thing when applying reinforcement learning, especially when we want the agent to learn solve a new kind of problem. Therefore, we designed three kinds of terminal reward, including the fixed value reward, the dynamic value reward, and the boundary reward, in order to encourage our agent to reach the goal. In additional to the terminal reward, we also introduced a step reward which will be given to the agent every steps.

Besides, one important thing in our framework is that we set up a step limit $n$ for the agent. If the agent can not find the goal state within $n$ steps, then we will treat it as a fail. A negative reward will be given to the agent as a penalty. This is useful since usually the reason of not finding the goal is that the agent fail to protect the clear region. After losing lots of the clean area, the best way is to start from the beginning.

*1) The Step Reward:* The step rewards is related to the number of the clear region and is defined as follows:

$$R_{step} = node_{clear} - cost_{step} \tag{1}$$

where, $node_{clear}$ is the number of the clear region and $cost_{step}$ is a fixed value for step cost. The idea of the step cost is to encourage the agent to find the shortest path and reach the goal directly instead of doing a lot of useless movements first and then reaching the goal. Remember that the agent will receive $R_{step}$ every steps so taking $nodes_{clear}$ into account can guild the agent to clear more region directly.

*2) The Fixed Value Terminal Reward:* For the terminal reward, let's start from the first one, the fixed value reward. In the fixed value reward, when the agent successfully clears all the region, we will give it a fixed positive value as a reward. On the other hand, when the agent fails to clear all the region, we will give it a fixed negative value as a penalty. This reward function could be written as follows:

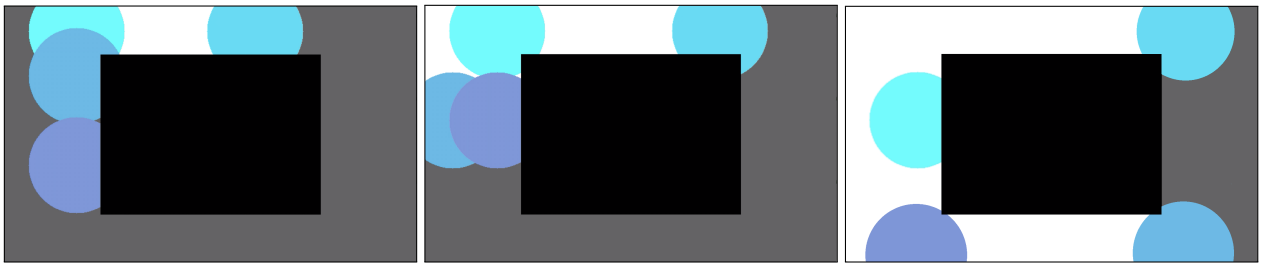
$$R_f = \begin{cases} +150 & , if\ reaching\ goal \\ -150 & , otherwise \end{cases} \tag{2}$$

*3) The Dynamic Terminal Reward:* The second one is the dynamic reward. In the fixed value reward, we treat all of the fail terminal state evenly, which means that all of the fail terminal state get the same penalty. However, as shown in the Fig. 7a and Fig. 7b, both of them are fail states but it is intuitive to understand that Fig. 7b is better than Fig. 7a since it has more clear region. Therefore, in order to encourage the agent to expand the clear region, we proposed the dynamic reward which takes the clear region into consideration. The dynamic reward is defined as follow:

$$R_d = \begin{cases} 20 \cdot node_{clear} & , if\ reaching\ goal \\ -1000 + 10 \cdot node_{clear} & , otherwise \end{cases} \tag{3}$$

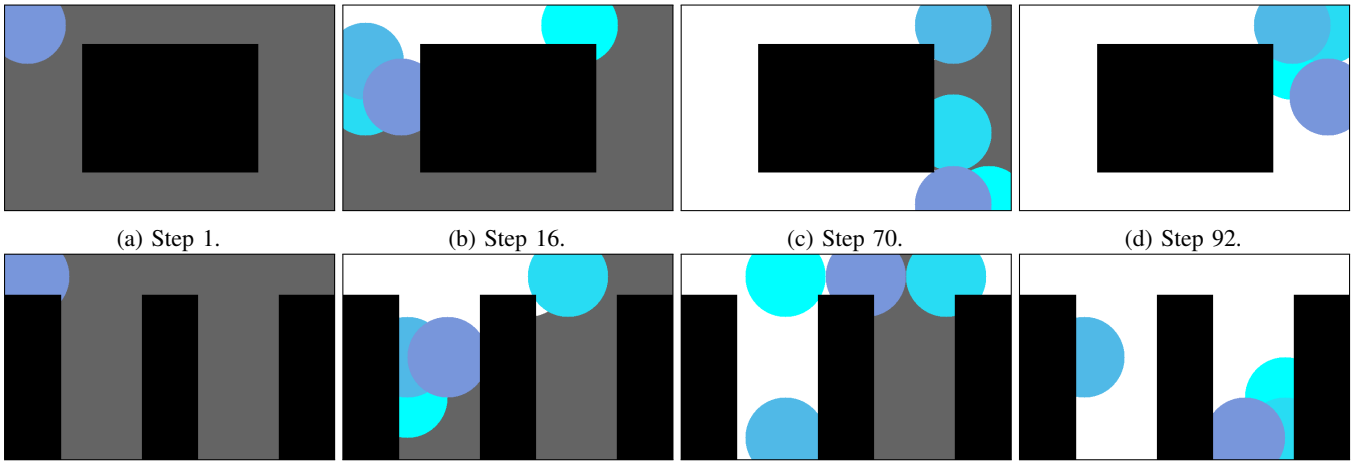
The positive reward $20 \cdot node_{clear}$ is actually a fixed value but will change according to the environment. For a more complex environment which contains more nodes, the positive reward will also be larger, which is also reasonable.

*4) The Boundary Terminal Reward:* The last one is the boundary reward. As the Fig. 7c shows, it is actually very common to see some pursuers staying in a clear region and



(a) (b) (c)

Fig. 7: (a), (b) show the idea why we need the dynamic reward and (c) shows the idea why we need the boundary reward.



(a) Step 1. (b) Step 16. (c) Step 70. (d) Step 92.

(e) Step 1. (f) Step 17. (g) Step 43. (h) Step 68.

Fig. 8: Result path of the q-learning agent. (a)-(d) is the result in the ladder map with k and w both equal to 2. (e)-(h) is the result in the tree map with k and w both equal to 2. The agents are trained for 30,000 and 50,000 episodes respectively.

being useless. However, in Fig. 7c, if we want to clear the region, we need all of the four pursuers work together in the contaminated region or say in the frontier. Therefore, we proposed the boundary reward which gives some extra points to the agent when pursuers are on the boundary. A boundary pursuer is defined as a pursuer who has both clear neighbors and contaminated neighbors. By this definition, both pursuers staying in purely contaminated region and purely clear region are not counted as a boundary pursuer. After defining the boundary pursuer, we can now modify the dynamic reward to equip with the boundary reward as:

$$R_b = \begin{cases} 20 \cdot node_{clear} & ,if\ reaching\ goal \\ -1000 + 10 \cdot node_{clear} & \\ \quad +100 \cdot pursuer_{boundary} & ,otherwise \end{cases} \tag{4}$$

The weight of $pursuer_{boundary}$ is very high since in our case only few pursuers are used (less than 5).

### C. Q-Learning Agent and Exploration Strategy

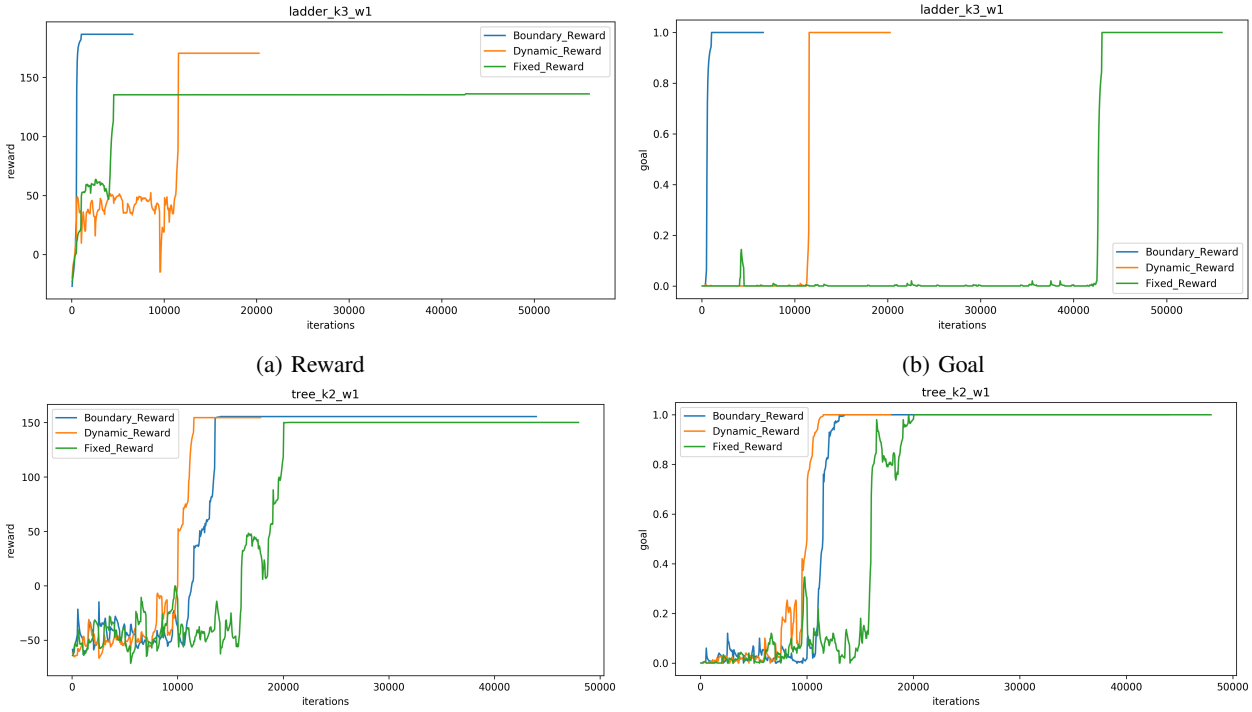
We implemented the agent with the Q-learning algorithm, but equipped with different exploration strategy. Exploration strategy is a very important thing in reinforcement learning. If the exploration strategy is poor, then it would be hard for the agent to explore the environment and find the goal. Therefore, we proposed three kinds of exploration strategies [12], the purely random exploration, the Boltzmann-distributed exploration, and the counter-based exploration. Remember that the exploration is happened when we are going to choose an action to do. Therefore, what we need to do is to define a policy to select the action given an available action set.

*1) The purely random exploration:* The first one is the purely random exploration, which is the simplest one and is adopted most of the time. The idea is to randomly select one action from the available action set with probability $\epsilon$. For the rest of the probability $1 - \epsilon$, the agent will select the action with the highest q value. Also, notice that the exploration probability $\epsilon$ will decay as the training goes on.

*2) The Boltzmann-distributed exploration:* The Boltzmann-distributed exploration considers the utility of the actions and assign the probability to the actions according their utility. The distribution is given by:

$$P(a) = \frac{\exp U(a)}{\sum_{a' \in actions} \exp U(a')} \tag{5}$$

where $U(a)$ is the utility of the action $a$. As we can see, in this distribution, if one of the action have a high utility, then it will usually dominate the distribution. However, if




(a) Reward

(b) Goal

(c) Reward

(d) Goal

Fig. 9: Comparison between different reward functions. (a), (b) are tested on the map ladder_k3_w1. (c), (d) are tested on the map tree_k2_w1. The **reward** means the average reward the agent can collect in the training phase. The **goal** means the average rate of the agent successfully reaching the goal in the training phase.

the utilities are not stable and have roughly the same value, then the distribution will be even. The benefits of using the Boltzmann-distributed exploration is that after learning for a while, the agent will not do exploration when it is familiar with the states. However, when entering to a new states, the agent will start to explore.

*3) The counter-based exploration:* The last one is the counter-based exploration. In this exploration strategy, we further consider how many times the states have been explored. If the state has not been explored for enough times, then we will give it more chance. However, if the state has been explored many times, then we will reduce the chance that we explore it again. The distribution could be describe as follows:

$$P(a) = \alpha \cdot U(a) + \frac{\beta}{c(s'|s,a)} \tag{6}$$

where $U(a)$ is the utility of the action $a$, $c(s)$ is a counter function that keeps the times the agent explores $s$, and $\alpha, \beta$ are the weighting coefficient. As the equation illustrated, if the agent has already explored $s'$ a lots of times, then the second term will be small, but if the agent explores it only for a few times, then the second term will be large. By using this distribution, the agent would like to explore the states that it didn't enter before.

Notice that though the Boltzmann-distributed exploration and the counter-based exploration can adjust itself between exploration and exploitation, we still use an $\epsilon$ to decide whether the agent needs to do the exploration or not. For the rest $1 - \epsilon$, we just let the agent choose the action with the highest utility, which leads to the exploitation.

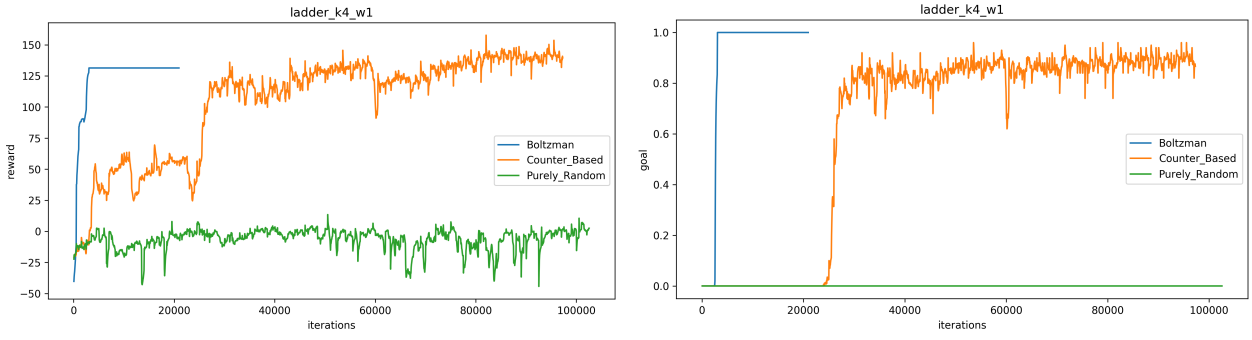
## VI. Extensions Experiment and Result

In this section, we describe the result of our extension with the q-learning agent. We also compare the difference between the three reward functions and the three exploration strategy.

### A. Experiment Setup

We use the same maps that were built in the benchmark stage. Besides, the code is written in python with the help of OpenCV [13] to display the movement. The three experiments are described as follows.

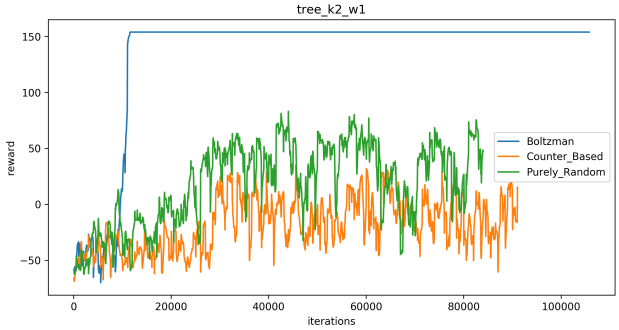
1) The results of q-learning agent, showing our best results to clear the map. We use the boundary reward and the Boltzmann-distributed exploration strategy in this experiment.
2) The comparison between different reward functions. In this experiment, we use the Boltzmann-distributed exploration strategy with three different reward functions to see the agent's behavior.
3) The comparison between different exploration strategies. In this experiment, the reward function we used is the boundary reward. By applying different exploration strategies, we can see the difference between them.



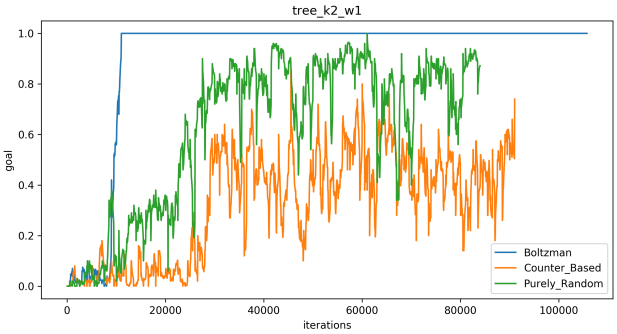

(a) Reward




(b) Goal




(c) Reward




(d) Goal

Fig. 10: Comparison between different exploration strategies. (a), (b) are tested on the map ladder_k4_w1. (c), (d) are tested on the map tree_k2_w1. The **reward** means the average reward the agent can collect in the training phase. The **goal** means the average rate of the agent successfully reaching the goal in the training phase.

Besides, the step limit $n$ is 120 in all of the cases. The $\epsilon$ for the purely random exploration is 0.3 and decays half for every $10,000$ episodes until 0.05. The $\epsilon$ for the Boltzmann-distributed exploration and the counter-based exploration is always 0.3. The weighting term $\alpha$ and $\beta$ in the counter-based exploration are 0.5 and 10 respectively.

### *B. Result and Analysis*

In this section, we will show the result with some figures and give some analysis to explain the possible reasons.

*1) Result Path of Our Extensions:* As the Fig. 8 shows, our agent can really solve the problem by cleaning up all the contaminated region. These two maps with the path width $w = 2$ are actually very difficult to solve because it relies much on the cooperation between the pursuers. Also, in the tree structure, as shown in the Fig. 8 (e)-(h), it is much more difficult because after the pursuers clean the left branch, they have to go all the way back to the top and use the only path to enter to the right branch. As a result, In the tree structure, the agent needs more training episodes to finish the task.

*2) Comparison between Reward Functions:* As shown in the Fig. 10, both dynamic reward and the boundary reward outperform the fixed reward. The **goal** figure means the rate of the agent achieving the goal. Thus, when the goal rate is 1, it means that the agent can always clear up the region. Although the behaviors of dynamic reward and the boundary reward are quite similar in this setting, the boundary reward could help us to solve a more complex map such as the map with the path width $w = 2$ illustrated above. Besides, in the Fig. 10 (a), though the fixed reward setting reaches to a high reward very soon, it doesn't reach the goal actually. If we look very carefully at the green line around the $40,000$ episodes, there is actually a small jump which reveals the fix reward setting successfully clear all the region.

*3) Comparison between Exploration Strategy:* In this experiment, we can see, in Fig. 10, the Boltzmann-distributed exploration simply outperforms the other two strategies. The counter-based exploration agent performs better in the ladder structure than in the tree structure. The reason could be the influence of the structure since in the tree structure, there is only one single path to reach to another branch. If the agent doesn't enter to another branch, it will keep exploring a space where the goal does not exists.

## VII. Conclusion

Our experiments show that the proposed extension approach works well to clear the region and is able to improve itself if we train it for more episodes. Moreover, our experiments also suggest that different reward functions and exploration strategies will greatly influence the agent's performance. Therefore, designing a good reward function and exploration strategy is inevitable if we want to further improve the result and test on a more complex map.

## REFERENCES


[1] R. Ramaithitima, S. Srivastava, S. Bhattacharya, A. Speranzon, and V. Kumar, "Hierarchical strategy synthesis for pursuit-evasion problems." in *ECAI*, 2016, pp. 1370–1378.

[2] T. D. Parsons, "Pursuit-evasion in a graph," in *Theory and applications of graphs*. Springer, 1978, pp. 426–441.

[3] G. Hollinger, A. Kehagias, and S. Singh, "Gsst: Anytime guaranteed search," *Autonomous Robots*, vol. 29, no. 1, pp. 99–118, 2010.

[4] J. P. Hespanha, H. J. Kim, and S. Sastry, "Multiple-agent probabilistic pursuit-evasion games," in *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, vol. 3. IEEE, 1999, pp. 2432–2437.

[5] V. Isler, D. Sun, and S. Sastry, "Roadmap based pursuit-evasion and collision avoidance." in *Robotics: Science and Systems*, vol. 1, 2005, pp. 257–264.

[6] M. Aigner and M. Fromme, "A game of cops and robbers," *Discrete Applied Mathematics*, vol. 8, no. 1, pp. 1–12, 1984.

[7] Ö. Şimşek and A. G. Barto, "Using relative novelty to identify useful temporal abstractions in reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 95.

[8] Ö. 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*. ACM, 2005, pp. 816–823.

[9] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," *arXiv preprint arXiv:1704.01665*, 2017.

[10] N. Ono and K. Fukumoto, "Multi-agent reinforcement learning: A modular approach," in *Second International Conference on Multiagent Systems*, 1996, pp. 252–258.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[12] R. McFarlane, "A survey of exploration strategies in reinforcement learning," *McGill University, http://www. cs. mcgill. ca/ cs526/roger. pdf, accessed: April*, 2018.

[13] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.