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 evaders can move arbitrary fast. Therefore, 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. We proposed a reinforcement learning method for finding a sequence of movement to clear the region. In the current version of paper, we provide the literal review of the Pursuit-Evasion problem and reinforcement learning methods for some similar problems. Besides, the benchmark algorithm [1] is implemented for comparison.**

I. Introduction

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

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 a way to turn the continues map into the graph structure, the graph is much more complicated and is hard to analyze by hand.

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. 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 unsolvable.

In another case, Hespanha et al. [4] proposed a probabilistic method but what they were working 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 purser. In their method, the purser 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 purser-evader problem are also well studied. One of the well known Purser-Evader algorithms for solving basic problems is the Timed-Road Algorithm [5]. To define a basic problem, given a single Purser $P0$ and a single Evader $E0$, can $P0$ eventually collide with $E0$? At the beginning, the Timed-Road Algorithm lists all the position state of (P, E), and starts to update the time-stamp of each state, and finally we use the steps of each state to verify whether the Purser $P0$ can catch the Evader $E0$ at some certain state. The algorithm can be implemented with dynamic programming but with dimensions increasing, it will also increase our time-complexity.

Another interesting approach is simulated as a game of cops and robbers [6] where the start a single cop and a single robber and we want to find a graph which cops always wins that means cops can 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.

The problem we are implementing is the worst case of the pursuit-evasion problem experimented in [1]. 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. Therefore, to address this problem, we formulate it as a clear-region problem. As shown in Fig. 1, the red region are contaminated but once the purser checks the region, the region turn in to a clear region (green) which means there is no evader. However, after the purser passes through the branch, the clear region gets recontaminated again since there exists a path from the contaminated region to it, and as our assumption, the evader could run arbitrary fast so the region we just clear it up is now not clear anymore.

Our attempt to improve the pursuit-evasion problem solution

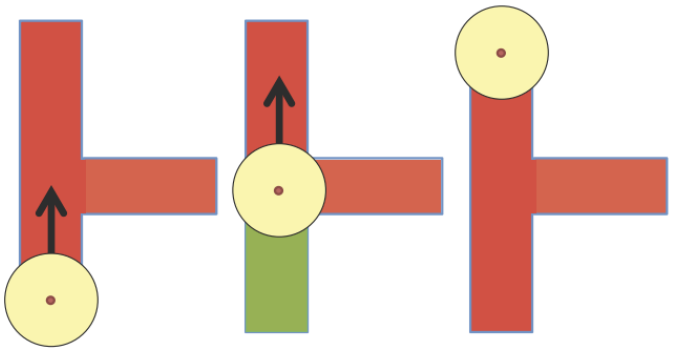


Fig. 1: Caption

through introducing Reinforcement Learning can be related to work done by Ramaithitima et al. [1], who purposed an abstraction framework to partition the configuration space into sets of similar configurations, abstract state, to reduce search space. However, the partition is off-line algorithm, which means all the partition are done before doing search using Breadth-first-search liked algorithm. As a result, it may lead to exploring enormous redundant configuration state.

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.

Dai et al. [9] Moreover, researches 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 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.

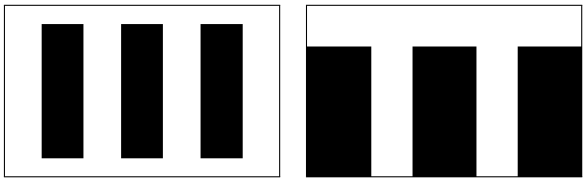
To view on another approach of pursuit-evasion problem, [10], the paper introduces a way of a new reinforcement learning strategy. Same as the pursuit-evasion problem, the methods for the pursuers not only includes capturing the prey, but also requires the pursuers to surround the prey in a 2D-graph.Hence 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.

II. Benchmark Algorithm

The benchmark algorithm is dividing into two parts. The first part is “Partition Algorithm” which use the real states to build the abstract states. The real state represents the position of pursers and the abstract state represent single or multiple real states. The second part is running the “Search Algorithm” on the abstract state and since we greatly reduce the number of nodes by merging multiple real states into single abstract state.

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 / clear status of the node is not in our consideration. 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 pursers one by one to find next possible real state (pursers’ position). During searching for next pursers, we will also find the connected components for each real state. The connected components represent the spaces that are separated by the pursuers. We define 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 of the configuration of pursuers, we could then cluster these configuration into a Abstract State.

After building the abstract state, we can now do 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, we need to find a path on the original 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)



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

Fig. 2: Different kind of maps. The white region means the road that purser and evader could go. The black region means the wall.

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.

Besides the benchmark algorithm, we also implemented the baseline planner mentioned in [1]. Since we do not have the detail of the planner. We basically do Breadth-first search on believe state X, consists of the configuration/position of all pursuers, denoted by p, and possible positions of the evaders, which will be referred as the contaminated regions denoted by c. X = (p, c)

III. 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. 2. 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 pursers to finish the task. Notice that we assume the purser 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 pursers 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	0.015	0.0029
ladder, k=4, w=1	0.032	
tree, k=1, w=1	1.34	0.0029
tree, k=1, w=2	3.991	0.242866
tree, k=3, w=2	922.44	132.047

TABLE I: The execution time for Brute Force and Benchmark algorithm.

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.

IV. Conclusion

Acknowledgment

The authors would like to thank...

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.