Maze solver with reinforcement learning using multiple agents

Vishisht Rao
Computer Science and Engineering
PES University
Bangalore, India
vishisht.rao@gmail.com

Rithvik G
Computer Science and Engineering
PES University
Bangalore, India
rithvik.ganeshhs@gmail.com

V R Badri Prasad
Department of
Computer Science
PES University
Bangalore, India

***Abstract*—Reinforcement learning is a subset of machine learning wherein agents have no prior knowledge of the environment but learn progressively as a result of rewards given to them when they perform an action that would lead them to complete a given task. Q-learning is an algorithm in reinforcement learning which employs a Q-Table to determine the total reward an agent may obtain by performing an action in any given state. This paper shows how the maze problem can be solved in a more efficient manner by employing multiple agents using the concept of threading. The agents cooperate with each other by updating values to a shared Q-Table.**

Keywords— Q-Learning, Maze, Bellman Equation, Agent

I. Introduction

Reinforcement learning is a subset of machine learning. It involves performing a certain action in order to maximise the reward for a given situation. It is used to find the optimum path that should be taken in order to achieve a goal for the given task. Reinforcement learning is different from other forms of machine learning in the sense that an agent does not know anything about the environment that it is put into, but learns by performing random actions in order to explore the environment further and eventually draws conclusions based on the reward that it receives in each scenario.

Applications of Reinforcement Learning:

- Robotics
- Games
- Chemistry
- Deep Learning

Q-learning is an algorithm in reinforcement learning that procures the optimum action to take from any given state. A policy defines what action an agent takes based on the state that it is in. Q-learning is termed as an off-policy algorithm as it does not require any policy, instead it performs random actions and learns from experience. Hence it can be said that Q-learning learns a policy that maximises the reward.

Q-learning stands for Quality-learning. Quality is defined as how good an action is in order to maximise the future reward.

Q-learning employs a Q-Table to represent the reward to be gained by selecting any given action from any given state. It is an nxn matrix, where n represents the number of states. The value Q[i][j] in the matrix represents the reward to be gained by transitioning from the state i to the state j. All the values in the Q-Table are initially set to zero as the agent has no knowledge of the environment. During each episode, the values are updated. After the training is complete, the Q-Table is used as a reference for the agent to traverse through in order to yield an optimum path.

II. Related Works

The authors of [1] explore how multiple agents can cooperate with each other without having to communicate with each other much, since communication between agents increases the complexity of the problem. This paper solves the dilemma problem using the concept of internal rewards (as compared to the conventional external rewards used in Q-Learning). This paper explains theoretically the idea behind setting internal rewards to assure the cooperation between the agents with minimal communication and provides a method to update the goal values within the agents under no communication.

The authors of [2] use the DFS algorithm to construct a map of the maze, such that once the map is constructed, a path between any two points can be determined. In order to speed up the construction of the map, two agents are used, and the information that they collect is exchanged between them when they collide into each other. The path that each agent traverses is stored in a stack, and the information stored in these stacks are compared and exchanged.

This paper provides an improvised depth-first search to establish a map of the given maze. Two agents are used to make the process of exploring the maze faster as compared to one agent. The results of the above procedure show that this is feasible.

In [3], the authors explore three Reinforcement Learning techniques. Discrete Q Learning, Dyna-CA and FRIQ-Learning. To check performance in varied conditions following four maze configurations are considered:
9x6 sized maze without obstacles.
9x6 sized maze with obstacles.
18x12 sized maze without obstacles.
18x12 sized maze with obstacles.

The authors in [3] conclude that FRIQ Learning outperforms all others under all conditions.
It explains how Q-Learning is an iterative algorithm that assumes an initial condition before the first update occurs to find a fixed point solution of a complex puzzle. The Q-Learning has discrete state, action and Q-function representation and hence provides a sluggish response towards decision making.

The authors in [4] use 4 searching algorithms namely DFS, BFS, Uniform Cost Search and A* search and compare them on the basis of performance, completeness and optimality. The author applies all these algorithms on pacman in small maze, medium maze and big maze. The authors then show the complexities of all these algorithms.
In the 4 algorithms 3 are uninformed search strategies and 1 is an informed search strategy among the three uninformed search strategies Uniform Cost Search gives the best time complexity. But uniform cost search is useful only when the path costs are different otherwise it reduces to breadth first search. If all the path costs are the same then breadth first search gives us the best time complexity. The informed search strategy: A* search is optimal if we use consistent path cost(in case of graph search).

III. Proposed Work

In order for an agent to solve a maze, the structure of the maze must first be clearly defined. The feasibility matrix(F) is a matrix of order nxn where n is the number of states(or number of cells) in the maze. The value in the ith row and jth column of the feasibility matrix is 1 if it is possible to move directly from the ith state to the jth state in the maze without passing through any intermediate states.

The reward matrix(R) and the Q-Table(Q) are matrices of order nxn where n is the number of states(or number of cells) in the maze. The value in the ith row and jth column of the reward matrix indicates the reward an agent gains by moving from the ith state to the jth state. The reward matrix is initialised to zero and the reward for the ith row and jth column is given as 10 if j is the goal state and ith row and jth column of the feasibility matrix is equal to 1. The Q-Table is where the maximum expected future reward for all actions at each state is calculated and stored using the Bellman Equation. The Q-Table is initialised to zero as the agent has no knowledge to begin with.

The agents are then allowed to train for a certain number of episodes which is defined by the user. This is where the agent is allowed to explore the maze based on certain parameters such as learning rate(which is the importance given to the knowledge the agent already has, if the learning rate is 0 then the agent learns nothing new and relies completely on its current knowledge) and discount factor(which determines how much the agent cares about rewards in the distant future relative to those in the immediate future, if the discount factor is 1 the agent cares only about future rewards). For each iteration in the agent's training, the agent chooses a random start state(currentState), calculates certain values based on the start state for the Bellman Equation and then applies the Bellman Equation to modify a value in the Q-Table. The Bellman Equation obtained from [5] is as follows:

$$Q[currentState][nextState] = (1 - learnRate)*Q[currentState][nextState] + learnRate*(R[currentState][nextState] + discountFactor*maxQ(nextState, nextnextState))$$

The "nextState" is defined as the state to which an agent in the current state can proceed directly to and the "nextnextState" is defined as the state to which an agent can directly proceed to from the "nextState". The maxQ function determines the highest value in the Q-Table in the "nextState" row among the "nextnextState" columns. The Bellman Equation can be decomposed into two parts, the first part gives the immediate reward and the second part gives the discounted future values. The "currentState" takes on the value of the "nextState" and the entire process in each iteration is repeated until the goal state is reached.

After the training has completed the agent starts from the start state(currentState), scans the Q-Table to determine the highest value in the "currentState" row and the column corresponding to this highest value is called the "nextState". The "currentState" assumes the value of the "nextState" and this process is continued till the goal state is reached. By recording the values of the "currentState" the shortest path between the two states can be noted.

In order to use multiple agents, they must simultaneously execute their train portions individually while updating a common Q-Table. Doing this would allow each agent to train for (1/k) number of episodes where k is the number of agents. This is done using the concept of threading.

Threading is the concept of splitting a process into multiple parts(threads) such that each thread can execute concurrently with the other threads in order to maximise utilisation of resources and thereby minimise the overall time of execution. In this implementation, pthreads in C language is used due to its simplicity as well as the overall efficiency of C language. Every Computer's processor has a certain number of cores, every core has a certain number of threads. Hence in order to maximise efficiency, all the threads available within a system can be used.

During an agent's training, in each iteration the agent randomly selects a start state and proceeds from there. The number of iterations is equal to the number of episodes. If "n" is the number of start states and "e" is the number of episodes, the average number of times a given state is randomly selected as the start state is (e/n). If the number of start states is reduced by a factor of "k" and the number of episodes is also reduced by a factor of "k" then the average number of times a given state is randomly selected as the start state is also (e/n). Hence it will hold true that if there are "k" agents, the number of states can be split into "k" parts with each agent randomly selecting a start state from one of the parts and each agent training (1/k) times the number of episodes, the resulting Q-Table values will be the

same as long as a common Q-Table is used in each agent's training.

To illustrate the above concept with an example, consider a maze consisting of 20 states. Assume that 10,000 episodes are run. Then if a random start state is selected in every episode, the average number of times a given state is selected as the start state is 500 (obtained by dividing 10,000 by 20). If 4 agents are used where the first agent selects only among the first 5 states as its start state, the second agent selects only among the next 5 states as its start state and so on, the average number of times a given state would be selected as the start state is 2000 (obtained by dividing 10,000 by 5). If we decrease the number of episodes of each agent to 2500 the average number of times a given state would be selected as the start state is 500 (obtained by dividing 2500 by 5).

IV. Pseudo code

Shown below is the algorithm for the train function of an agent. The time complexity of this algorithm is Ω(maxEpochs).

```
// Algorithm to train a Q-Learning Agent
/* Inputs:
        Q (Q-Table)
        nextStates[ ] (array with possible next states of each state)
*/
//Output: Q

for i=0 to maxEpochs do
        currentState = random state from set of all states
        while true do
                nextState = random possible next state from currentState
                maxQ = - infinity

                for j=0 to number of possible next states from nextState do
                        nextnextState = jth state from set of next states of
                nextState
                        q = Q[nextState][nextNextState]
                        if q > maxQ then maxQ = q

                Apply Bellman Equation for Q[currentState][nextState]

                currentState = nextState
                if currentState = goal then break
```

V. Experimentation

The above procedure has been applied on an Ubuntu 20.04 OS running on a system having 2 cores and 4 threads. The results are as shown below.

In the figures shown below the time taken for the programs to finish execution is given by the real time. The time taken for a single agent is 8.798s, the time taken for two agents is 5.062s, the time taken for four agents is 4.292s. There is a speed up of 1.74 between a single agent and two agents and there is a speedup of 2.05 between a single agent and four agents.

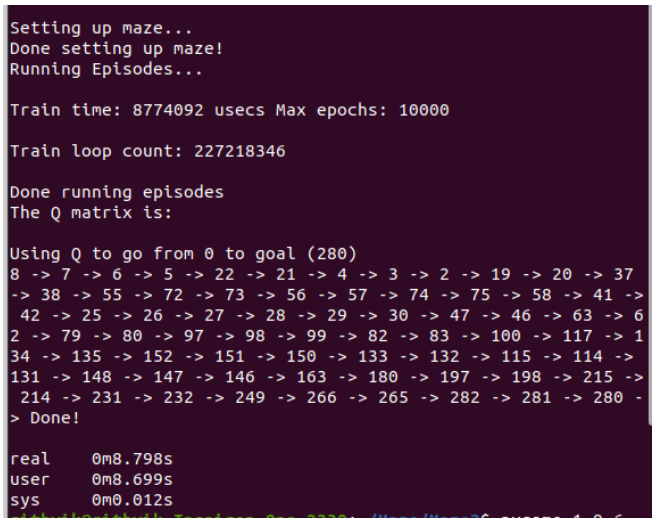

Fig. 1 The time taken for single agent to solve maze

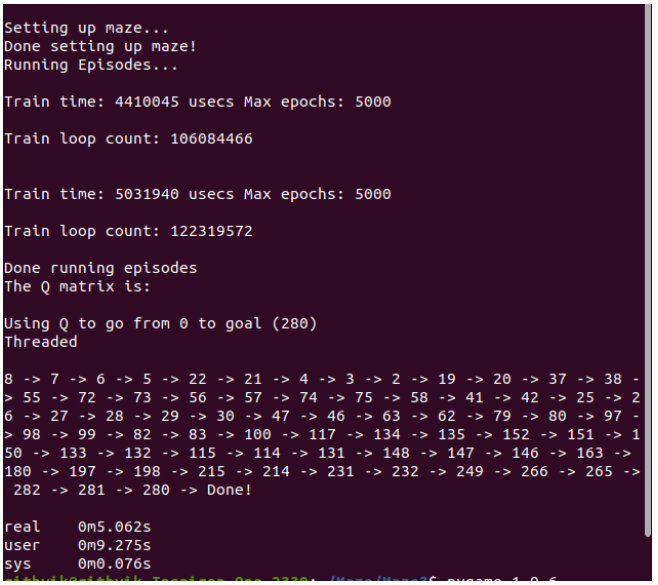

Fig. 2 The time taken for two agents to solve maze

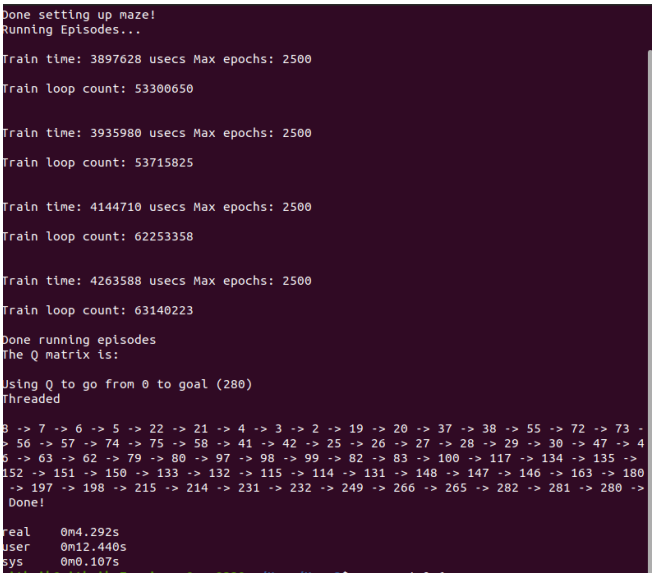

Fig. 3 The time taken for four agents to solve maze

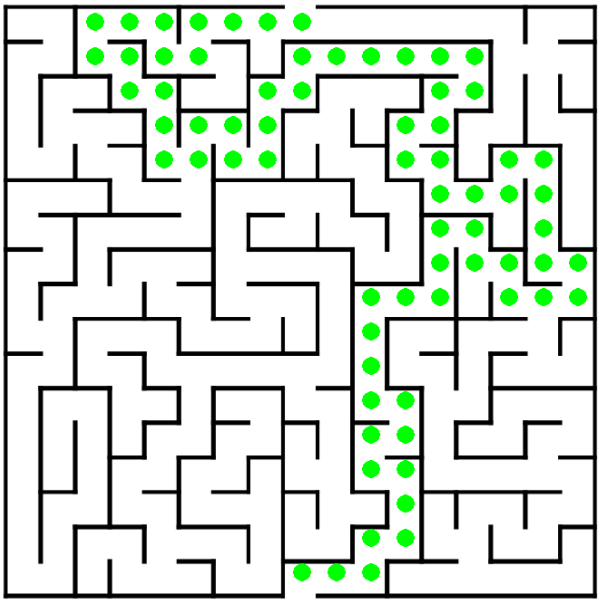

Fig. 4 Path found by agent(s) to solve the maze

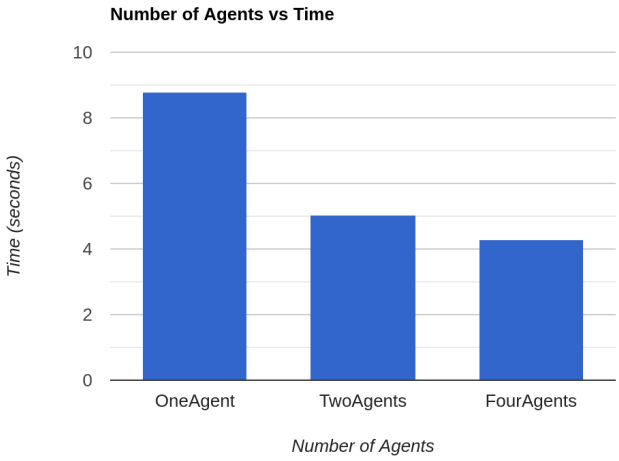

Fig. 5 Variation of time with number of agents

VI. Conclusion

The approach taken in this paper shows that solving the maze problem with multiple agents by using the concept of threading decreases the time required for the problem to be solved. As every agent randomly selects a start state from a subset of the total states, the number of episodes to be run is less. Since each agent represents a thread and the threads execute concurrently, each agent takes less time to complete its task as compared to a single agent solving the entire maze.

VII. Future works

In this paper the feasibility matrix is written manually, this process can be automated by using image processing techniques.

Optimization techniques using dynamic programming can be explored to get a more efficient solution.

VIII. References

[1] Uwano, F., Tatebe, N., Tajima, Y., Nakata, M., Kovacs, T., & Takadama, K. (2018). Multi-agent cooperation based on reinforcement learning with internal reward in maze problem. SICE Journal of Control, Measurement, and System Integration, 11(4), 321-330.

[2] Chen, Y. H., & Wu, C. M. (2020, September). An Improved Algorithm for Searching Maze Based on Depth-First Search. In 2020 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-Taiwan) (pp. 1-2). IEEE.

[3] https://www.ijettcs.org/Volume6Issue1/IJETTCS-2017-01-17-12.pdf

[4] Pal, J. B., Modak, S., & Chatterjee, D. (2019). DESIGNING OF SEARCH AGENTS USING PACMAN.

[5] https://visualstudiomagazine.com/articles/2018/10/18/q-learning-with-python.aspx

[6] https://en.wikipedia.org/wiki/Q-learning