Shortest path finding based on Q-Learning

Yuchen Jiang, *Postgraduate, SUSTech*

***Abstract*—Shortest path finding problem is widely appeared for engineering and has application in real life. To solve the shortest path problem in weighted directed graph, we explore multiple approaches, including dynamic programming, reinforcement learning (specifically Q-learning), and value function approximation. Reinforcement learning is widely used to solve problems in real life. In this report, we take use of reinforcement learning to solve shortest path problem in weighted directed graph. Q-learning method is applied and we also use value function approximation to solve the problem. Besides, dynamic programming is applied to the graph to compute the shortest path between any pair of start and end nodes. Furthermore, we try to train a model based on value function approximation such that the obtained solution can work even if you offer a different graph. The results of experiments show that we achieve good performance.**

***Index Terms*—Shortest path problem, Dynamic programming, Reinforcement learning, Q-learning**

I. Introduction

SHORTEST path problem is a general problem in engineering field, which is applied in robot navigation and path planning. In this report, we provide several methods to solve shortest path problem in weighted directed graph, including dynamic programming, reinforcement learning and value function approximation. We test these method in weighted directed graphs and compare their performance. Besides, we also try to build a model based on value function approximation such that the obtained solution can work even if you offer a different graph.

Dynamic programming [1] is a powerful algorithmic technique used to solve optimization problems by breaking them down into overlapping subproblems. It relies on the principle of "optimal substructure," which means that an optimal solution to a larger problem can be constructed from optimal solutions to its smaller subproblems. Dynamic programming often involves storing the solutions to subproblems in a table or an array, allowing for efficient retrieval and reuse. One well-known application of dynamic programming is the Floyd-Warshall algorithm [2], which is used to find the shortest paths between all pairs of vertices in a weighted graph. The algorithm iteratively updates the distances between vertices by considering intermediate vertices and gradually builds the shortest paths. By using dynamic programming principles, the Floyd-Warshall algorithm efficiently computes the shortest paths between all pairs of vertices in a graph with a time complexity of $O(V^3)$, where V is the number of vertices [2].

Reinforcement learning [3] is a branch of machine learning that focuses on how an agent can learn to make sequential decisions in an environment to maximize a cumulative reward [4]. It is inspired by the principles of behavioral psychology, where an agent interacts with an environment, takes actions, and receives feedback in the form of rewards or penalties. The goal of reinforcement learning is to find an optimal policy that guides the agent's decision-making process to achieve the maximum possible long-term reward. One popular algorithm in reinforcement learning is Q-learning [5]. Q-learning is a model-free, off-policy algorithm that aims to learn the optimal action-value function, called the Q-function. The Q-function represents the expected cumulative reward an agent can obtain by taking a particular action in a given state and following a specific policy thereafter. Q-learning iteratively updates the Q-values based on the rewards received during the agent's interactions with the environment. By repeatedly updating and refining the Q-values, the agent gradually learns the optimal policy for maximizing its long-term rewards. Q-learning is well-suited for problems with discrete states and actions, and it has been successfully applied to various domains, including game playing, robotics, and autonomous systems.

What's more, in reinforcement learning, value function approximation [6] is a technique used to estimate the value function in environments with large or continuous state spaces. Instead of maintaining a lookup table to store the value of each state or state-action pair, value function approximation aims to generalize and approximate the value function using a parametric function approximator, such as a neural network. By learning from a subset of observed states, the approximator can generalize its predictions to unseen states. Value function approximation allows reinforcement learning algorithms to scale to more complex environments by reducing the memory and computational requirements. The most commonly used approach for value function approximation is called function approximation with linear models. In this method, the value function is represented as a linear combination of features that capture relevant information about the state. These features can be handcrafted or automatically derived from the observed states. The coefficients of the linear combination are learned through techniques like gradient descent or least squares, which minimize the discrepancy between the predicted values and the true values obtained from the environment. Value function approximation provides a powerful framework for solving reinforcement learning problems in high-dimensional and continuous state spaces. By leveraging function approximation techniques, agents can effectively estimate the value function, make informed decisions, and learn optimal policies in more complex and realistic environments.

II. Problem formulation

Here we define our problem with several definitions.

Definition 1 (Weighted Directed Graph): We define the basic data structure of our question as weighted directed graph, in which nodes and edges exists. All edges are weighted with non-negative values and they have exact direction between nodes.

Definition 2 (Problem): The problem is to find shortest path solution to any pair of start and end nodes without re-doing the methods, which means only doing once for the whole graph.

III. Main results (Algorithms)

Here we show the algorithms of each method.

A. Floyd-Warshall Algorithm

Algorithm 1 Floyd-Warshall Algorithm

Input:Weighted directed Graph $\mathcal{G}$
Output:Global distance matrix $\mathcal{D}$
Initialize a distance matrix D
For $(from\_node, to\_node, weight) \in \mathcal{G}.edges$:
    $weight \rightarrow \mathcal{D}[from\_node][to\_node]$
Set diagonal elements with 0
Other elements are initialized with INFINITE
For $node\ k \in \mathcal{G}.nodes$
    For each pair $nodei$ and $nodej$
        If $\mathcal{D}[i][j] > \mathcal{D}[i][k] + \mathcal{D}[k][j]$ **then**
            Update $\mathcal{D}[i][j]$ with smaller value
Return $\mathcal{D}$

The time complexity of the Floyd-Warshall algorithm is $O(V^3)$, where V is the number of vertices in the graph. This makes it suitable for small to medium-sized graphs. However, the algorithm may become computationally expensive for very large graphs due to the cubic complexity.

B. Q-Learning

Algorithm 2 Q-learning Algorithm

Initialize:
Initialize Q-table Q with random values for all state-action pairs
Set learning rate α, discount factor γ, Set exploration rate ϵ, number of episodes N
for each episode in N **do**
    Initialize starting state s
    while episode is not finished **do**
        Choose action a based on exploration-exploitation trade-off
        Execute action a, observe new state s' and reward r
        Update Q-value for the previous state-action pair: $Q[s][a] \leftarrow (1 - \alpha) \cdot Q[s][a] + \alpha \cdot (r + \gamma \cdot \max_a Q[s'][a'])$
        Update current state s to s'
    end while
end for

We set episode number as 1500, learning rate as 0.2, discount factor as 0.9 and epsilon as 0.2 to get the optimal performance. The Q-learning algorithm starts by initializing a Q-table, which is a lookup table storing the estimated values (Q-values) for each state-action pair. The Q-values represent the expected cumulative reward an agent can obtain by taking a particular action in a given state and following a specific policy thereafter. The algorithm then sets hyperparameters such as the learning rate (α), discount factor (γ), exploration rate (ϵ), and the number of episodes (N).

In each episode, the algorithm follows an exploration-exploitation trade-off to choose an action. The action is executed, resulting in a new state and a reward. The Q-value for the previous state-action pair is updated using the Bellman equation, which incorporates the reward, the maximum Q-value for the next state (s'), and the discount factor (γ). The Q-value update is performed iteratively until the episode is finished.

After executing all the episodes, the Q-table is updated with better estimates of the Q-values, reflecting the learned optimal policy. The agent can then use the Q-table to make informed decisions and follow the optimal policy for maximizing long-term rewards.

C. Value Function Approximation

Algorithm 3 Value Function Approximation

Input:
Set observed states $\mathcal{S}$, actions $\mathcal{A}$, Learning rate α, Discount factor γ, Number of iterations N
Initialize:
Choose a function approximator Q
Randomly initialize weights $\mathbf{w}$ of Q
Training Loop:
for each iteration in N **do**
    Initialize an episode
    Observe initial state s
    Repeat until episode termination:
        Choose an action a based on exploration-exploitation trade-off
        Execute action a, observe next state s' and reward r
        Compute TD target: $y = r + \gamma \cdot \max_{a'} Q(s', a', \mathbf{w})$
        Compute estimated Q-value: $Q_{\text{estimated}} = Q(s, a, \mathbf{w})$
        Update weights: $\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot (y - Q_{\text{estimated}}) \cdot \nabla_{\mathbf{w}} Q(s, a, \mathbf{w})$
        Update current state s to s'
end for
Output: Approximated value function Q with weights $\mathbf{w}$

The algorithm for value function approximation starts by defining the input parameters, including the set of observed states $\mathcal{S}$, the set of actions $\mathcal{A}$, the learning rate α, the discount factor γ, and the number of iterations N. The algorithm then initializes a function approximator Q (such as a neural network) and randomly initializes the weights $\mathbf{w}$ of Q. It then enters the training loop, where it iteratively performs the following steps for each iteration

IV. Experiments(Simulations)

All experiments are shown in **'Final Project.ipynb'** file. The experiment environment is shown in 'mlai_fp.yaml' file. Here we do some analysis for it.

A. Efficiency

We mainly talk about cost time for this part. It's noticed that we record the cost time when solving the problem. Among all methods, dynamic programming(Floyd-Warshall Algorithm) is the fastest. Since Q-learning and VFA (Value Function Approximation) need to simulate and update through episodes many times, their performance are worse that Floyd-Warshall Algorithm. It's obvious because we are generally use algorithms instead of reinforcement learning to solve the shortest path problem.

B. Accuracy

Unfortunately, only Q-learning has the ability to compete against Floyd-Warshall Algorithm. When graph is difficult or node number increases, VFA will fail to find shortest path.

C. Others

Since we need to test these methods randomly, we will face different kinds of graph. The two kinds of graph which confused me are sparse graph and cycle graph, which both lead to INFINITE solution and give trouble to Q-learning and VFA. We set cycle detection in Q-learning and VFA (see notes in code) to make it successfully running.

Besides, we try to build up a model based on value function approximation such that the obtained solution can work even if you offer a different graph. However, there is no time to finish training it since it costed too much time to solve the first four questions.

V. Conclusion and future problems

In this project, we take use of three methods, which are dynamic programming, Q-learning and value function approximation in reinforcement learning. I try my best to solve the problem and analyze the experiment results. If possible, the project should be improved to meet a better performance. Although in my opinion, it's not suitable to solve shortest path in weighted directed graph by reinforcement learning, it inspires us about applications in real life. Sometimes we can not get global information of the environment and the environment may change (according to add and delete edge in WDGraph class in my code). Reinforcement learning then has a good performance for the application. All in all, I gain much about reinforcement learning and shortest path problem. I wish it would help me in the future.

References

[1] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[2] M. A. Djojo and K. Karyono, "Computational load analysis of dijkstra, a, and floyd-warshall algorithms in mesh network," in *2013 International Conference on Robotics, Biomimetics, Intelligent Computational Systems*. IEEE, 2013, pp. 104–108.

[3] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[4] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[5] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[6] G. Konidaris, S. Osentoski, and P. Thomas, "Value function approximation in reinforcement learning using the fourier basis," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 25, no. 1, 2011, pp. 380–385.