A Deterministic Improved Q-Learning for Path Planning of a Mobile Robot

ELEC-E8119 Robotics: Manipulation, Decision Making and Learning

19th December 2016
Course Project

Jaakko Mattila 426367
Eljas Hyyrynen 348380

Abstract

The effective path planning in a space with obstacles is a major requirement for autonomous mobile robots. Optimizing some aspects of the movements such as travel time or path length presumes path planning. There are several approaches to path planning, one of which is grid-based search in discretized configuration space.

Konar et al. [1] propose a specialized algorithm for generating a collision-free path in an environment with obstacles. They have made several improvements on the classical Q-learning algorithm [CQL] which under certain special assumptions increase performance on the path planning task. This improved Q-learning algorithm overcomes well-known search algorithms like A* and Dijkstra in the experiments the paper presents. They have also implemented the algorithm on a real robot which accounts on the applicability of their algorithm.

This report introduces the improved Q-learning algorithm [1] and implements it with some modifications. This implementation is tested in Matlab simulations and further compared to the original results of the research paper [1]. Through this extended experimentation, the capability of the original algorithm is fully realized in its strengths and limitations, allowing suggestion of improvements and further work on similar problem.

Approach

In path planning, reinforcement learning methods, such temporal difference, provide way to iteratively improve the action value function to determine the best route. One of such methods is Q-learning, which chooses the local best action at each visited state. The goal of Q-learning is to sum these local optimals to eventually reach the global optimal path. Therefore, Q-learning is not tied to following any policy as each state is evaluated independent of each other, utilizing off-policy learning. The following method would be considered classical Q-learning(CQL):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right].$$

[2]

The CQL employs a Q-table to store the Q-value for each action-state pair. In contrast to this, Konar et al. [1] combined the idea of classical Q-learning with the assumption of known distance from the current state to its next states and the goal state. Hence, this improved Q-learning(IQL) method only requires the storing of the Q-value with the best local action of each state. Therefore, the update time is reduced for the action-values of each state.

Furthermore, at each state, the Q-values are only updated once using the action values of the local neighbor states. To ensure a single sweep of updates at the states, the IQL introduces a boolean lock variable for each state. If the Q-value of a given state needs no updating, the state is locked, by identifying the best action for the current locked states. The neighboring states of a locked state can be updated with the Properties derived in the paper as seen in Figure 1. In conclusion, the Q-value is higher the closer the state is to the goal, which has the highest reward.

Property 2: If $L_p = 1$ and $d_{nG} < d_{pG}$ then $Q_n = Q_p / \gamma$ and set $L_n = 1$.

Figure 1: One of the four properties to determine whether present or neighbor value is updated, depending on which is locked. Then, updating the value of that open state depending on if it's closer or further than its neighbor [1].

We have implemented the improved Q-learning algorithm by Konar et al. in Matlab and provide some experiments that grant additional information on the limits of their algorithm. The experiments are presented in the next section of this report. The implementation consists of two steps: generating the Q-value table based on the pseudo code in the paper and path planning with the known Q-value table. Additionally, we present a visualisation of each test run.

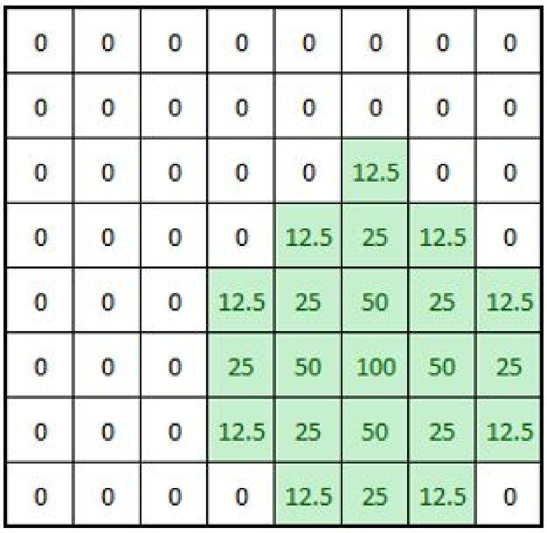
The original paper presents the Q-table and lock-up table L as column vectors. We have chosen to implement these vectors as matrices for practical purposes. This simplifies editing of the values for the programmer, visualization. Also, transformations are not required between column vector values and their representatives in Cartesian coordinates.

In the Q-table update step presented in the paper, step 2 of the pseudo code states: “Select and executes actions until the agent is at the goal state”. To our understanding, this step is composed mainly for the purpose of a physical robot reaching the goal. Our motivation was solely simulation rather than running the code with a real robot. Thus, we avoided the computational time -consuming random walking step in our software by setting the current state to the goal state in the initialization step.

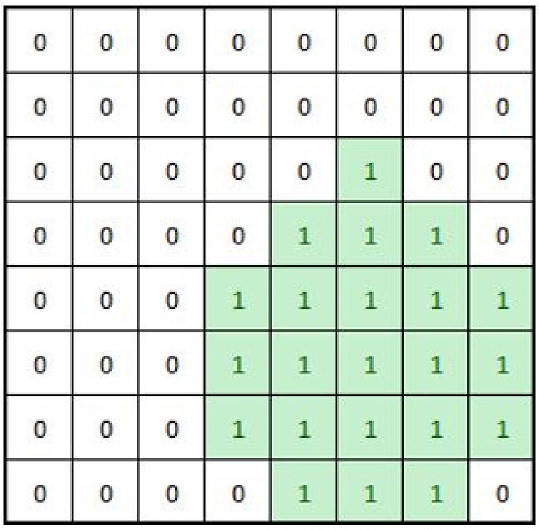
Similarly to the paper, our algorithm iterates the Q-table until all states are locked. However, our implementation differs in the order selection of Q-table value updates and lock-up states. In the paper, this has been stated ambiguously, hence, we chose the following implementation to solve the problem.

The approach of this report introduces a list of states for which all neighbors are not locked. First, this approach updates all neighbors of the first incomplete locked state and removes them from the list. Naturally, the initial starting state for this step, the goal state, is locked already. Second, the neighbors of the removed neighbors are added to the list. This step repeats until all of the neighbor states are locked and no more open neighbors can be found for any of the locked states.

On parallel with the locking, the values of the open states are updated with the values of those locked states. The values of these updates depends entirely on the initial or goal reward, and the discount factor used to reduce the reward with respect to distance from that goal state. The highest Q-value is with the goal state while it neighbors get once discounted reward and so on. As a result, we get a Q-table with elements that essentially inform the city block distance from the goal to each element as shown in Figure 2 below.



0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	12.5	0	0
0	0	0	0	12.5	25	12.5	0
0	0	0	12.5	25	50	25	12.5
0	0	0	25	50	100	50	25
0	0	0	12.5	25	50	25	12.5
0	0	0	0	12.5	25	12.5	0



0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	1	1	1	0
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	0

Figure 2: An example of a Q-table (left) and boolean lock table (right) during the execution of the algorithm. The state with value of 100 is the goal state, while the discount factor is 0.5. The Q-value of each cell is essentially goal reward time the discount factor to the power of distance from the goal state or $Q(G) = y^d$.

With the generated Q-table, the path planning algorithm can solve a path from the initial state to goal state by always choosing as a next state the neighboring state with the highest Q-value and without an obstacle. If there are multiple states that satisfy these condition, the one that requires least turns from the current pose is preferred. This is achieved by comparing the next action with the previous action. The next action closest to the previous action is preferred over otherwise equal alternatives.

Both the algorithm presented in this report and the approach by Konar et al. [1] assume the availability of the following information. First, they assume the knowledge about distance from current state to the goal from every reachable state. Secondly, they both perform in a grid world. This discretization limits our movement to rectilinear distance which is practical as the paper authors' experiments are run with a robot that is able to make only turns in sets of 90 degree. Finally, the algorithm assumes fully observable world. These assumptions limit the applicability of IQL. Nevertheless, they enable increased performance compared to more general algorithms such as A* and Dijkstra's algorithm in special cases.

Results

We have conducted several simulations to benchmark the algorithm implementation of the work presented by Konar et al [1]. We gathered data on several runs, most interesting of which are presented in this chapter. Our focus was on differences between the run times, path choices and limitations of our and the paper authors' algorithms. Even though our algorithm does not reach the performance of the paper authors' implementation, we discovered possibilities for future development of the improved Q-learning algorithm.

The first experiment, presented in Figure 3, is similar to experiment 3 by Konar et al [1]. Since our algorithm requires notably more computation time for a 20x20 grid, depicted in Figure 4, we have scaled down the grid into an essential 10x5 patch containing the critical obstacles, initial location and goal.

Our algorithm renders the Q-table in 640 milliseconds and finishes the path planning in additional 34 milliseconds. The resulting path corresponds exactly the experiment by Konar et al. [1] which was expected for the simple obstacle configuration. Even though their algorithm takes significantly longer to run - 9.34 seconds - we have not outperformed them since they render a Q-table over a notably larger configuration space.

This experiment illustrates that our implementation of the original algorithm can achieve similar results to the paper authors for significantly smaller configuration spaces with simple obstacle configurations.

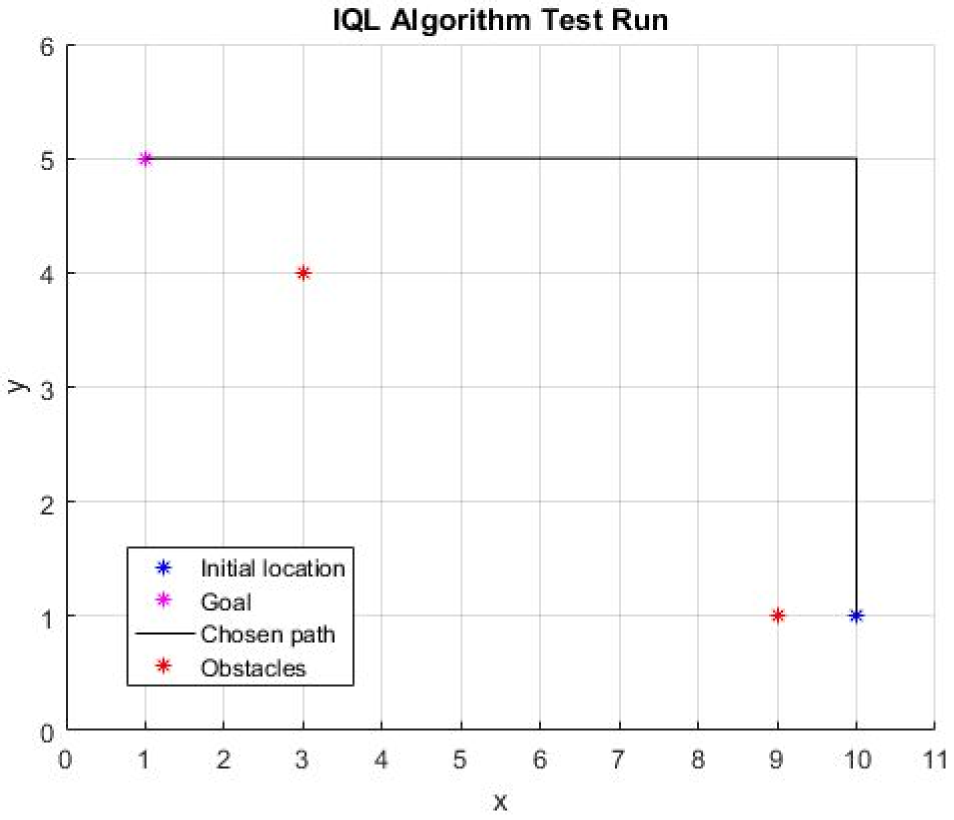


Figure 3: The performance of the implementation presented in this paper with similar experiment to Konar et al. [1].

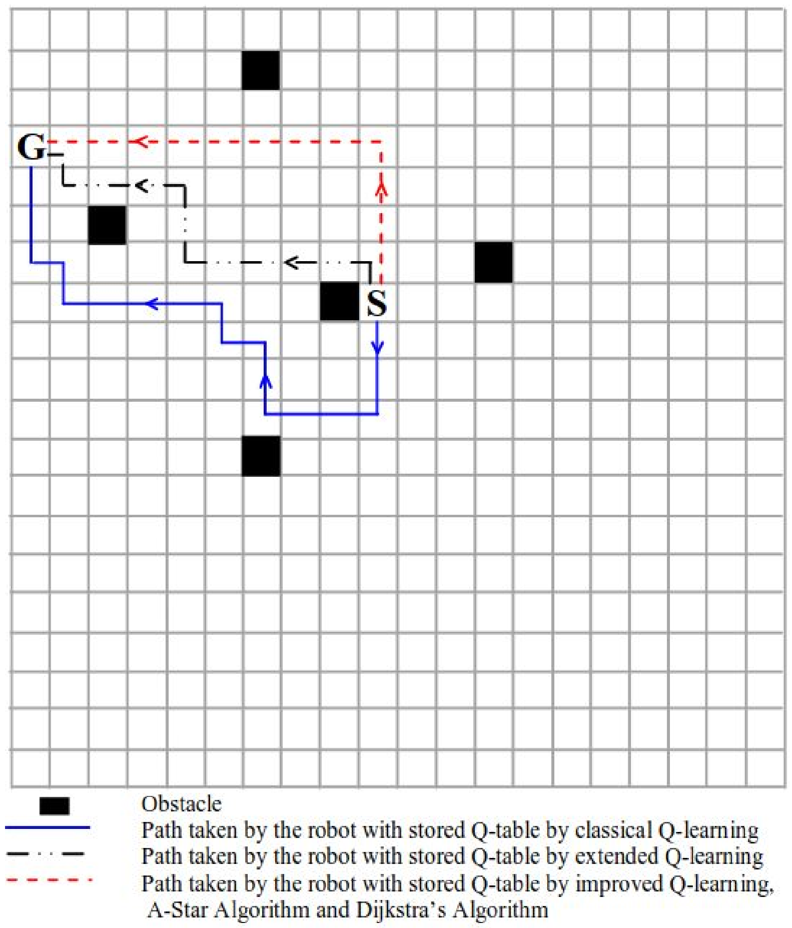

G
S
Obstacle
Path taken by the robot with stored Q-table by classical Q-learning
Path taken by the robot with stored Q-table by extended Q-learning
Path taken by the robot with stored Q-table by improved Q-learning, A-Star Algorithm and Dijkstra's Algorithm

Figure 4: The performance of improved Q-learning algorithm along with several other algorithms. [1]

In the second experiment (Figure 5), we have defined a maze-like configuration space. The Q-table is generated in 259 milliseconds and path planning is done in 42 milliseconds. The experiment reveals that the resulting path is not the shortest path since the neighboring states of the initial state both have equal Q-values. One of the reasons for these results may be that obstacles are ignored in the Q-table update step of the algorithm. Also, in the initial state the next state is chosen randomly. After iteration the available random decision, we could find the optimal path.

These kind of test cases are not presented in the experiments of Konar et al [1]. Our results may implicate that also their implementation requires experiments that include dead ends and turning back from unwanted locations. Those potential deadlocks are crucial to identify, if the algorithm can be used to reliably reach the goal state.

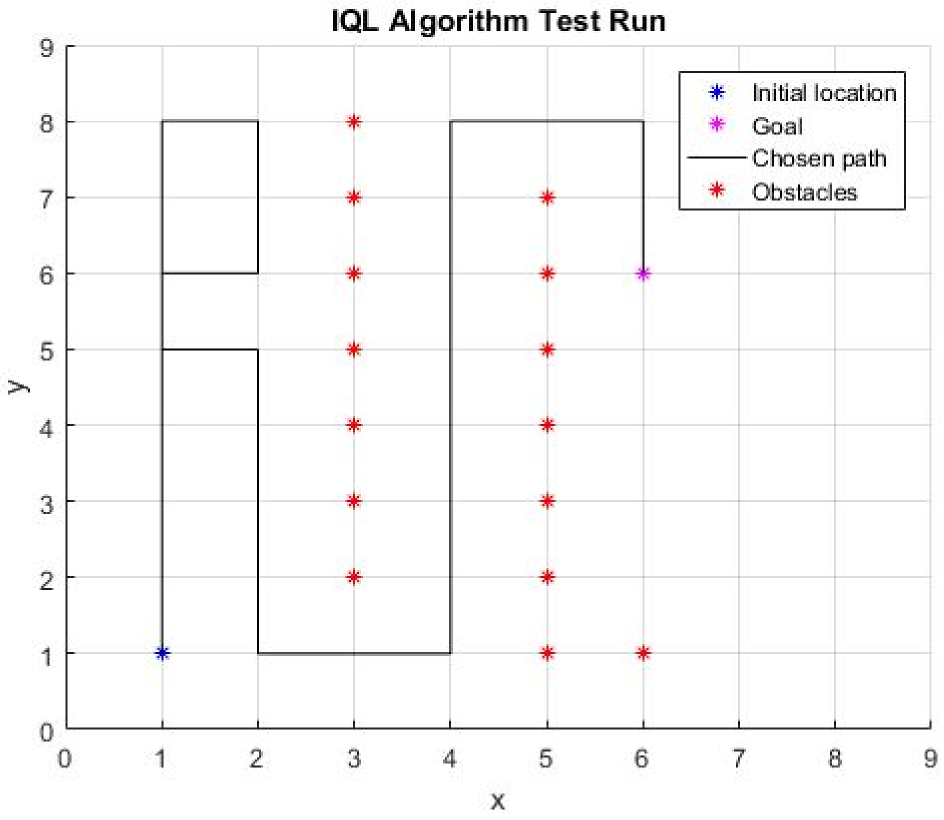


Figure 5: The algorithm chooses the direction after the initial state randomly. Afterwards, it prefers to move in the same direction. At y = 6, it chooses to go right, but has to make a loop as it encounters an obstacle on its path.

The third experiment (Figure 6) illustrates again how the algorithm may not initial choose the shortest path among alternatives. The 8x8 Q-table takes 250 milliseconds to generate and path planning step takes 35 milliseconds. This experiment highlights an important issue this algorithm can run into when preferring moving straight too much. To overcome this, the algorithm should have exploration within it to sometimes bypass the desire to keep on going straight. This could be achieved by updating the Q-values of that scenario of obstacles.

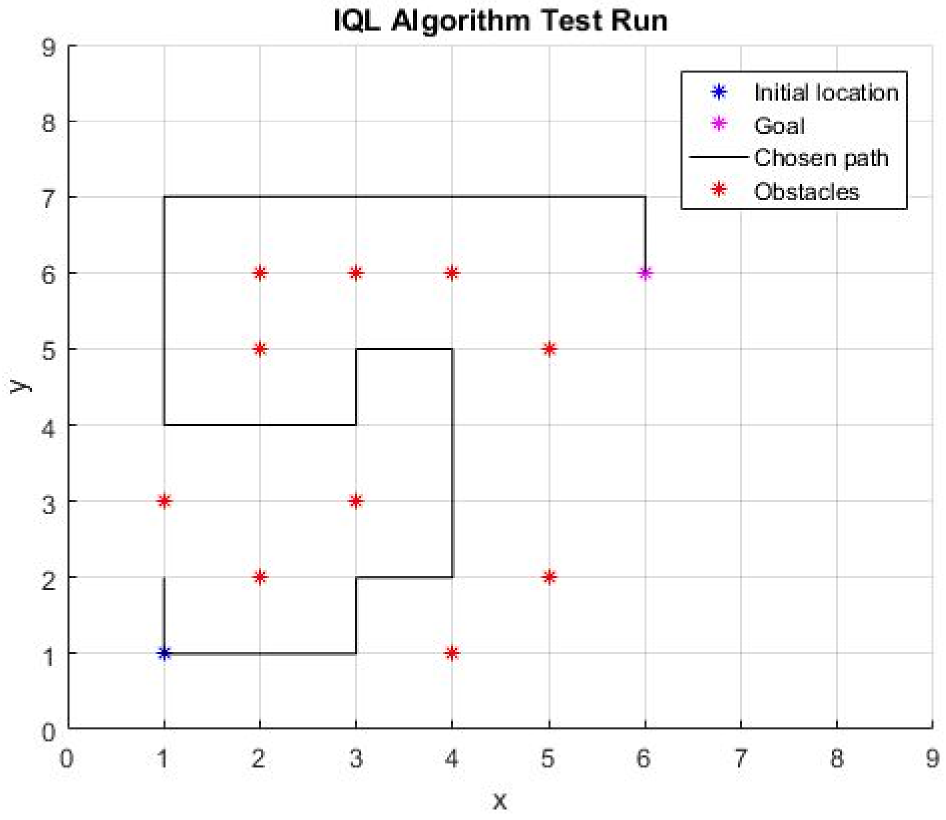


Figure 6: At state (x,y) = (4,4) the algorithm prefers not to change direction. This results into a local maximum and suboptimal path.

Finally, we introduce certain Q-table generation times for our algorithm to demonstrate the difference in performance between Konar et al. [1] and our algorithm. For a baseline for the comparison, the total planning time in the experiments by Konar et al. [1] ranges between 9 and 31 seconds for a 20x20 grid. With these measurements we can conclude that the performance of our algorithm is significantly lower than the ones presented by Konar et al. [1]. Also, our algorithm is not practical for any real-time applications that require online calculation of a grid below the size of 13x13. One of the reasons could be the computing power of the hardware, however, it is certainly not the only reason.

Grid size (units)	**Q-table generation time (seconds)**
8x8	0.250
13x13	7
14x14	47
16x16	Not finished in 500 seconds

Table 2: Q-table generation times for certain grid sizes.

Discussion

A dense grid discretization is required in an environment with narrow passages and small obstacles. As illustrated in the Results section, our implemented algorithm is not competent for such applications. In addition, the algorithm is not capable of rapid decision making. This problem can be resolved by improving the Q-table generation algorithm further.

A distinct limitation of the original algorithm is included within its assumption of available knowledge. The localization of the robot can be imprecise or unavailable, as well as the distance to the goal can be distorted by similar issues. These uncertainties is cause for huge random error, due to the nature of updating Q-values only once. This can devastate any path planning with a strong bias way of the margins. Therefore, a measure of accuracy for this information could be fruitful addition to increase the reliability of the algorithm in practice.

Our experiments demonstrate that the path planning step does not perform optimally in test cases that require choosing between equal maximum values. By choosing to iterate the algorithm several times or setting up other rules this phase could be improved in the future to overcome falling into local maximum values. For practical purpose, the shortest route is not necessarily the fastest one with physical robots. Therefore, we are not strictly considering only that as the optimal solution.

The algorithm outputs the same path as the implementation by Konar et al. [1] in simple, relatively small configuration spaces. Their algorithm runs every case in shorter time than our algorithm because of the time-complexity in Q-table generation step.

We introduced experiments that the original paper did not perform. The unwanted behavior of the discovered in these experiments may arise from their implementation as well as ours, however, it can be hard to say as there might have been aspects of the original algorithm lost in our implementation. Nevertheless, we may have revealed an issue to improve in their work as well.

We also noticed in our tests that the discount factor does not change the resulting path because every action value changes relatively. In addition, the discount factor could be time variant when iterating over the optimal path in the path planning step. This would allow the algorithm to ignore the reluctance to turning if the situation demands it.

Overall, the implementation revealead the challenges of implementing an unknown algorithm. This process illustrated the limitations and the open challenges of the algorithm. Extending this algorithm outside of the grid world would allow further interest in this topic. Especially, this would provide extended options in smaller turns, diagonal movement, and other aspects that would require more consideration when considering the movement of physical robots and their path clearing times.

References

[1] Amit Konar, Indrani Goswami, Sapam Jitu Singh, Lakhmi C. Jain, and Atulya K. Nagar.
A Deterministic Improved Q-Learning for Path Planning of a Mobile Robot.
IEEE SMCS, vol. 43, no. 5, pp. 1141-1153.

[2] R. S. Sutton and A. G. Barto.
Reinforcement Learning: An Introduction.
MIT Press, Cambridge, MA, 1988.

Commented source code to our project available at:
https://github.com/Hyrtsi/Robotics-Improved-Q-Learning-Project