Path Finding on LEGO EV3 Mindstorms Robot

Tran Trung Anh, 18520473, Truong Duc Vu, 18520194, Nguyen Phu Quoc, 18520343

***Abstract*—Path finding problem, studied by many researchers, have been growing fast and have remarkable results in the 21st century. The problem of path finding is defined as the discovery and plotting of an optimal route between two points on a plane. Its applications can be used on many real-life tasks. This article is about to explore deeper into some path finding algorithms, then build a simple robot to simulate the them on an already setup map. We choose Breadth-First Search and Markov Decision Process method for algorithms with briefly explanations and how to implement them. For the robot, we will build a LEGO EV3 Mindstorms line-follower robot, and explain how the robot works and how it recognise path on the map and traverse through it. Then, we calculate the time needed for the robot to traverse from starting point to destination point for each algorithm used. From that, we make explainations, comparisons between those algorithms and conclusion about the problem. Furthermore, we also discuss about applying a Reinforcement Learning approach on the problem, which tend to deal with the main disadvantage of the previous methods.**

***Index Terms*—Path planning, Shortest path problem, Automation, EV3, Lego Mindstorms**

I. Introduction

APPLYING autonomous robots has taken place in many industries from daily life task to space exploration. Path-planning is an important primitive for autonomous mobile robots that lets robots find the shortest – or otherwise optimal – path between two points. Otherwise optimal paths could be paths that minimize the amount of turning, the amount of braking or whatever a specific application requires. That leads to the need of effective path planning algorithms. Over recent years the increasing availability of low-cost programmable robotics system has lead to a growth in the implements of reinforcement learning control experiment. The Mindstorms robots manufactured by LEGO is one of this kind of robotics systems that have relatively low cost, programmable, and could accomplish a variety kinds of tasks with their flexible physical structure. The approaches that implement Breadth First Search and Markov Decision Process experiments for Lego Mindstorms robots may be classified into three sections. The first section is to give an overview of the robot's system and explain the path planning task. The second section includes short explainations on how the algorithms work. And the last section, we do experiments on the robot, then make conclusion about the problem. In this paper, we have successed on implementing BFS and MDP on EV3 robot. Though, it still has some issues due to the uncertainty of color sensors. We also try to implement a Reinforcement Learning approach, especially QLearning. But the limitaion of computing and storage resources, this work has not been successed. The primary goal of this paper is to implement the path planning algorithms for EV3 Robot. By that, we can practice, verify and improve our knowledge in the Artificial Intelligence field. We also want to contribute our work to the EV3 Mindstorms Robot research which is not very resourceful at this time.

II. EV3 Robot and Line Following Task

A. EV3 Robot details

Clearly one of the main restrictions on the adoption of educational robotics is the cost involved in providing enough robotics hardware for students to have sufficient hands-on experience. The Mindstorms robots manufactured by LEGO provide a possible platform for use in such a course. These robots are relatively lowcost, programmable, and due to being constructed from LEGO blocks have a flexible physical structure which can be adapted to a wide-range of tasks. In this article, we focus on the line-following task.

TABLE I
EV3 Robot Hardware details

OS	Linux
Display	178x128 pixel Monochrome LCD
CPU	TI Sitara AM1808 @300Mhz
RAM	64MB
Storage	MicroSD card 8GB
Connections	USB, Wifi, Bluetooth support

Our EV3 Robot, as shown in Fig. 1, has two wheels that can move back and forth freely, two Color Sensors and a .... for changing direction. The EV3 Color Sensor will emit a red light from the LED lamp and measure the intensity of light reflected back into the color sensor. Using a scale from 0 (very dark) to 100 (very light), the sensor will assign a number to the intensity reading.

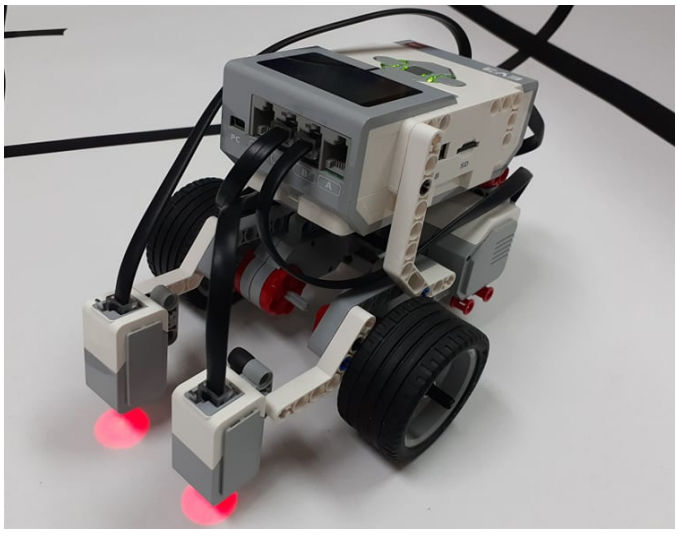

Fig. 1. EV3 Robot with two wheels and a two color sensors.

B. Line-Following Task

In this task, the mission of the robot is following a black line marked on a white surface, as shown in the Fig . 2. The line was sufficiently wide for both sensors to be positioned over it at the same time if the robot was oriented directly along the line. However, two Color Sensors work differently. The right sensor is used to track on the line, the left sensor is used to detect the changing direction pattern.

We use two sensors, the sensor has the main task of detecting

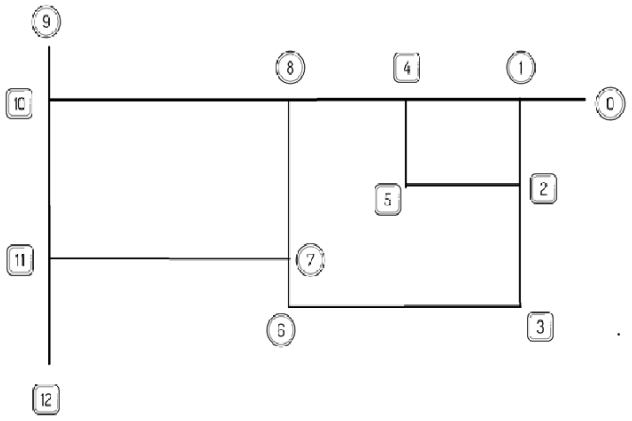

Fig. 2. The map.

the line in a straight line and is also used to identify the next state. The left sensor is used to detect the left path. The sensor works by using a color sensor. In our map there are only 2 colors: black and white. When the senser senses an all-black area, the value is lower and vice versa compared to white. The map includes 13 states and the reward of each route is predefined. We can choose any of 13 states to be starting point and destination point.

III. Algorithms

A. Breadth First Search

Breadth First Search (BFS) is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). Then move towards the next-level neighbour nodes. The pseudocode of BFS is shown as Algorithm 1.

Algorithm 1: BFS()

```
Result: Path
let Q be queue;
Q.enqueue( s );
mark s as visisted;
while Q is not empty do
    s' = Q.dequeue();
    for all neighbours n of s' in graph G do
        if n is not visited then
            Q.enqueue( n );
            mark n as visited;
        end
    end
end
```

B. Markov Decision Process

Decision making is an important task. Both the decision to be made and it's impact have immediate (short-term) and long-term effects. Immediate effects are transparent and can be judged at the current time-step, but long term effects are not always as transparent. It is easy to think of examples where we can have good short-term effects that result in poor overall long-term effects. We therefore want to choose an action that makes the right tradeoff to yield the best possible overall solution to maximize our reward. We use a Markov Decision Process (MDP) to model such problems to automate the robot and optimize its path. The components of an MDP model are:

1) A set of states S: These states represent how the world exists at different time points. Every such state, every possible way that the world can plausibly exist as, is a state in the MDP.
2) A set of actions A: These actions can affect the current state of the world, resulting in a new state; thus resulting in some effect and a different set of further actions that can be taken. An important point of MDPs is that effects of an action are probabilistic. Therefore, an action for a particular state can result in different states with different probabilities, thus allowing us to make a statistical decision.
3) A transition function $T(s, a, s_0)$: These are a result of taking an action at a state, as described above. This is also called the model or the dynamics and defines the probability of reaching state s0 from state s with action a i.e., $P(s' \mid s, a)$.
4) A reward function $R(s, a, s_0)$: These are a defined value for a (state; action) pair, the immediate reward value we obtain by taking an action at a particular state.

The solution to an MDP is called a policy, a mapping that defines for each state, the best possible action to take. For the sake of simplicity, we limit to finite horizon solutions, we assume that agents do not exist for infinite time periods. Therefore action values and rewards can decay with increasing time according to some γ, thus forcing agents to take action sooner rather than later. We also compute a value function that is similar to a policy, a mapping that defines for each state, a numerical value. From this value function, we can then easily derive the policy, given a numerical value for each state we can proceed to specify an action for each state, define a policy. Note that our MDPs, as their names specifically state, follow the Markov assumption. This says that for an MDP, the next state is solely determined by the current state and action. Note that there exist situations when we want to look back further, the effect of an action might not depend on the previous states, but can go back several states further. Our first-order Markovian assumption however, will not let us model these situations directly. Given an MDP, our goal is to find a policy, which is easy if we have knowledge of the values of states for a horizon length. To do this, Value Iteration Algorithm is a way

to compute the value function by finding a sequence of value functions, each derived from the previous one. Similar to every recursion function, we first define the base case. This is when our horizon $length = 1$, when we are at a state and need to make a single decision. Since we know the immediate rewards, for each state, we simply choose the action corresponding to the argmax over all rewards. Now the second iteration deals with horizon $length = 2$, when we have two steps forward. Here we want the immediate reward for the action we take, plus the reward for the future action. We've already computed horizon $length = 1$ for every state, so for each action, we simply sum the current reward plus the computed reward for the state it leads to and again, choose the action corresponding to the argmax over all rewards. We continue in this way until convergence. In all, this involves iterating over all states and weighting the values with transition probabilities. The formula is given below:

$$V^*_{i+1}(s) = max_a \sum_{s'} T(s,a,s')[R(s,a,s0) + \gamma V^*_i(s')]$$

This is also called the Bellman update. In short: $V^*_i(s)$ is the expected sum of rewards on starting from state s and acting optimally for horizon length = i. How do we know this converges? We want to show that $lim_{k->\infty} V_k = V_*$ i.e., we converge to unique optimal values. We know that for approximations U and V , as we continue, approximations get closer to one another. Therefore any approximation must get closer to the most optimal U and this therefore converges to a unique, stable, optimal solution. So in all, value iteration allows us to repeatedly update an estimate of the optimal value function according to the Bellman optimality equation, and we can thus compute the optimal policy (read: optimal value function). This therefore involves iterating over all states and using the transition probabilities to weight the values. Now look at Policy Iteration. This manipulates the policy indirectly, rather than finding it through value iteration. We start with some initial policy π_0 and at each iteration, perform a policy evaluation and a policy improvement. This therefore includes computing the policy at each iteration, solving the linear equation and then improving it, until we find the optimal π_0. The convergence property of policy iteration:$\pi \rightarrow \pi^*$. We can easily prove this by showing that each iteration is a contraction; therefore the policy must either improve at each step, or it must already be the optimal policy and does not change. Since the number of policies is finite (albeit exponentially large), policy iteration converges to the exact optimal policy. We could take an exponential number of iterations before we converge, but for most problems of interest, convergence occurs much faster. Interestingly, policy iteration requires fewer iterations than value iteration and gives us the exact value function. Value iteration often converges to the optimal policy long before the approximation of the value function is the correct value, but this is MDP-specific.

From above, we know that policy iteration requires fewer iterations than value iteration. However each iteration requires solving a linear system; which is in contrast to value iteration, where each iteration only requires applying the Bellman operator. In practice, policy iteration is often faster, especially when the transition probabilities are structured and sparse, which makes solving the linear system far more efficient.

Algorithm 2: MDP()

Result: Policy

1. Initalization: i = 0, $\pi_i(s) \in A(s)$ arbitrarily for all $s \in S$
2. Evaluation: For fixed current policy π_i, find values with policy evaluation:

Iterate until values converge:

$V^{\pi_i}_t(s) =$
$\sum_{s'} P(s'|s,\pi_i(s))[R(s,\pi_i(s),s') + \gamma V^{\pi_i}_{t-1}(s')]$

3. Improvement: For fixed values, get a better policy using policy extraction

One-step look-ahead:

$\pi_{i+1}(s) =$
$argmax_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_i}(s')]$

If $\pi_{i+1}(s) = \pi_i(s)$ for all $s \in S$ then return $\pi \approx \pi^*$; else i = i + 1 and go to 2.

IV. Experiment

We setup the EV3 Robot and run it on the map shown in Fig. 2.

In the initial trials, the reinforcement signal used rewarded the robot with positive reinforcement for any action which led to the robot remaining on the track (measured by applying a threshold to the summed value of the light sensors). As the actions available did not provide an option for staying still, this was expected to lead to the robot moving forward along the path and eventually traversing the circuit. However the learning algorithm discovered that alternating turning left and right allowed the robot to reverse slowly in a straight line, and hence maximal reinforcement could be achieved by travelling along a straight section of line at the beginning of the track, and then reversing back along that same section of track. To overcome this, the reinforcement function was modified so as to only reward the robot on time-steps on which it moved forward whilst remaining on the track. This required the reinforcement function to take into account which action had been performed (as there was no other means of determining whether the robot had moved forward) Therefore the reinforcement function was no longer being determined strictly on the basis of the environment which violates some of the principles of reinforcement learning. Unfortunately this could not be avoided given the limited sensory capabilities of the robot. (In principle the track itself could have been marked out with a greyscale gradient, but given the problems in sensing such a gradient which were observed during the walking task, this was not expected to be successful).

We set the starting point at state 0 and the destination point is state 12. The results are shown in Table II. We see that MDP found the more optimal path, but also comsumed more resource and more time to compute than BFS.

V. Future development

We currently simulate in the map which has no obstructions. If we apply the BFS algorithm or the MDP algorithm, it just

TABLE II
EXPERIMENT RESULTS.

Algorithm	BFS	MDP
Average run time	29s	25s
Average compute time	298ms	980ms

use for the case of map which has no obstructions.Inspired by the paper[1], we tried to approach Q-learning. It can be used for the map which has both cases. However, it caused a lot of problems.

A. Q Learning

Q-learning is a algorithm learning behavior based on expected reward of state-action. Q-learning is a temporal-difference method that learns a state-action value instead of learning state value. It is inspired value-function as folow:

$$V^*(s) = max_a Q(s, a)$$

Q-functions have an important property: a temporal value that learns for action selection of every state. It proceeds online and have the update equation for temporal-difference Q-functions: $Q(s, a) = Q(s, a) + \alpha(R(s) + \gamma max_{a'} Q(s', a') - Q(s, a))$ Where a is learning rate parameter which means a selection between exploitation and exploration. In here, the problem is for memory. If the number of state and action is and orderly about 1000 and 10. It means the table size of Q-values is $|S| * |A|$ equals to 10000 cells. It has a large quantity of memories. The chart shows Q-functions memories in Fig. 3:

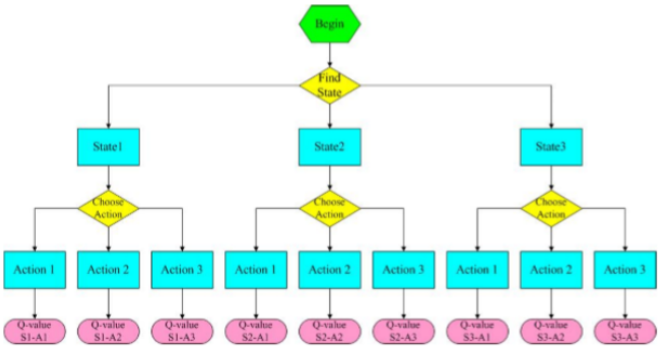

Fig. 3. State-action Q-values in original QLearning. Q-value table keeps every state-action's Q-value.

B. Simplified Q Learning

After every iteration, the robot has a new table and the table is memorized in a file. Unfortunately, the robot actually takes about 8 minutes to write to a file for 10000 cells in table. We can simplify the table memory by choose best action value for every state instead of list all state in every state. It will decrease a number cells of table from 10000 cells to about 1000 cells. Therefore, the time to wire a result file will be decrease from 8 minutes to 1 minute. The simplified chart of Q-learning shows in Fig. 4.

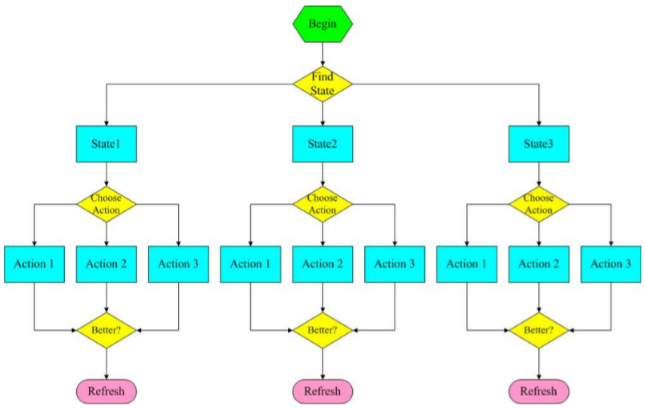

Fig. 4. State-action Q-values in original QLearning. Q-value table keeps every state-action's Q-value and its best action.

Implementing Q Learning on the robot should have better results but we have encountered some problems. We can not handle if any part of the map formed a cycle graph because its characteristic is state, not coordinate so if the state appears again, the robot can not be able to detect whether it appeared or not. Additionally, the time needed to compute Q Learning may take very long due to the limitation of hardware resources.

VI. CONCLUSION

In this article, we use Lego EV3 Mindstorms and address the problem of finding an obstacle course (the state of a negative point). From the map, the robot will collect status information and create a path to the end state. Our project mainly uses Markov decision processes (MDP) to solve the above problem. Our model is divided into two parts: path planning and run the robot on our map. We have experimented with Lego EV3 Mindstorms many times to come up with really good path planning, but when running, sometimes we encountered errors caused by sensors recognition issues. In the future, we want to improve our robots so that we can expand and run on more complex maps and above all, implement future development and implement Q-learning for our EV3 Robot.

REFERENCES

[1] K. Xu, F. Wu, and J. Zhao, "Simplified online q-learning for lego ev3 robot," 2016.