Dynamic Programming

Vaibhav Bishi
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, USA
vbishi@ucsd.edu

***Abstract*—This project presents an approach for implementing a Dynamic Programming algorithm, specifically Label Correcting algorithm, for autonomous navigation in a Door and Key environment. The results show the optimum estimation of the agent trajectory to the goal in various environments.**

Keywords—Dynamic programming, Markov Decision Process, Deterministic Shortest Path, Label correcting, Autonomous navigation

I. Introduction

Path planning is one of the most crucial problems in autonomous robotics. The problem is to find the best possible way to reach the goal from the starting position while minimizing the costs of executing the steps. This is basically an optimization problem, although we are aware of the environment here. This kind of shortest path problems can be solved extremely efficiently using Dynamic Programming (DP) algorithms, rather than simple brute-force methods. We basically solve the problem by constructing an optimal control policy for the robot from start to end. In this project, we implement the Label Correcting algorithm to solve the path planning problem.

II. Problem Statement

The navigation problem at hand is basically a shortest path problem and since this is a noiseless problem, we have a **Deterministic Shortest Path (DSP)** problem. This can thus be formulated as a **Markov Decision Process (MDP)**.

The **state space** is defined as

$$X = (x, y)$$

where (x, y) are the coordinates of the agent in the $m\ X\ n$ grid map of the environment. The agent direction is also incorporated indirectly into the agent's state as it plays an important part in its motion over time. The agent direction can be defined as

$$V = \{left: (-1,0), right: (1,0), up: (0,-1), down: (0,1)\}$$

The **control space** is defined as

$$U = \{move\ forward: M = 0, turn\ left: TL = 1, \\ turn\ right: TR = 2, \\ pickup\ key: PK = 3, \\ unlock\ door: UD = 4\}$$

These describe the various actions the agent can take.

One thing to note is that the map has a key and a door with specified locations. At all times, we can keep a check on the state of door, $D = \{open, closed\}$ and if the agent is in possession of the keys, $K = \{yes, no\}$.

The **motion model** is defined as

$$x_{t+1} = f(x_t, u_t)$$

When the agent has the input for turn left TL or right TR, the state remains unchanged but the orientation of the agent changes. The agent can only leave its current cell by moving forward to the cell in front by executing MF. So if the agent needs to move to its left, it takes the actions [TL, MF] in succession.

The other two miscellaneous actions are picking up the key [PK] and unlocking the door [UD] where the state remains unchanged.

The **initial state** is defined as s and the **terminal state** is defined as τ which are given in the problem.

The **planning horizon** is defined as T which is the maximum possible number of steps that the algorithm would require to finish.

$$T = grid\ width\ X\ grid\ height\ X\ \#\ of\ directions \\ + \#\ of\ miscellaneous\ actions$$

$$= width.height.4 + 2$$

The costs are defined as

Stage costs, $l(x,u) = \begin{cases} 1\ for\ all\ actions \\ 0\ for\ terminal\ state \end{cases}$

Terminal cost, $q(x) = \begin{cases} \infty\ for\ all\ states \\ 0\ for\ terminal\ state \end{cases}$

The objective is to find a path that has the minimum length from node s to node τ and this corresponds to the optimal policy that the agent should opt for reaching the goal.

III. Technical Approach

In order to solve the DSP problem, we implement a special case of Dynamic programming algorithm called the **Label Correcting (LC) Algorithm**. The LC algorithm have an advantage in the sense that they do not necessarily visit every node of the graph and thus are more efficient than simple forward and backward DP algorithms by prioritizing the visited nodes using cost-to-arrive values.

The pseudo-code for the algorithm is as follows:

Algorithm 3 Label Correcting Algorithm

```
OPEN ← {s}, g_s = 0, g_i = ∞ for all i ∈ V \ {s}
while OPEN is not empty do
    Remove i from OPEN
    for j ∈ Children(i) do
        if (g_i + c_ij) < g_j and (g_i + c_ij) < g_τ then   ▷ Only when c_ij ≥ 0 for all i, j ∈ V
            g_j = g_i + c_ij
            Parent(j) = i
            if j ≠ τ then
                OPEN = OPEN ∪{j}
```

In broad terms, firstly we estimate the optimal cost from s to each visited node in the state space. We determine all possible children of the nodes using the *child()* function. Then each time the label g_i is reduced, its children's labels g_j are corrected by adding the stage cost. We keep track of the parents when a child node is added to the open set in order to determine the optimal shortest path given by the set of nodes with least costs.

In order to find the shortest path from start s to goal τ, we break the problem down into multiple DSP sub-problems.

- Firstly, we check if there is a direct path from the start position to the goal and if the door comes in that path. We note the total path length.
- We find the shortest path from start to the key position such that the agent picks up the key in the most optimal way.
- Once the key is in possession, we find the best path from pickup location to the door and we unlock it.
- Once the door is unlocked, we estimate the shortest path from the door location to the goal.
- We then compare the direct path and the estimated optimum path (key+door+goal) and pick the path with the least length.

The shortest path obtained is a sequence of grid coordinates from start to end. Once we have this sequence, we extract the optimal policy from it using the *move()* function which gives the sequence of control inputs that the agent needs to implement in order to move along that shortest path. We take into account orientation of the agent in a given location and the orientation of the target cell with respect to the current cell. If it needs to go forward we put MF, to go back we put TR+TR+MF, to go right/left TR/TL+MF.

IV. Results

In this project, we have implemented the Dynamic programming algorithm on multiple environments with different sizes and settings. We have computed the optimal policy, referring to the best set of control inputs to reach the goal from start. We visualize the whole path trajectory of the agent over time using the *gym* and *gym_minigrid* modules in python. In part (a), we have computed the best policies and visualized the process for 7 different scenarios, 3 cases for a normal trajectory (key+door+goal), 2 cases for direct trajectory (direct path to goal) and 2 cases for shortcut trajectory (decide best path).

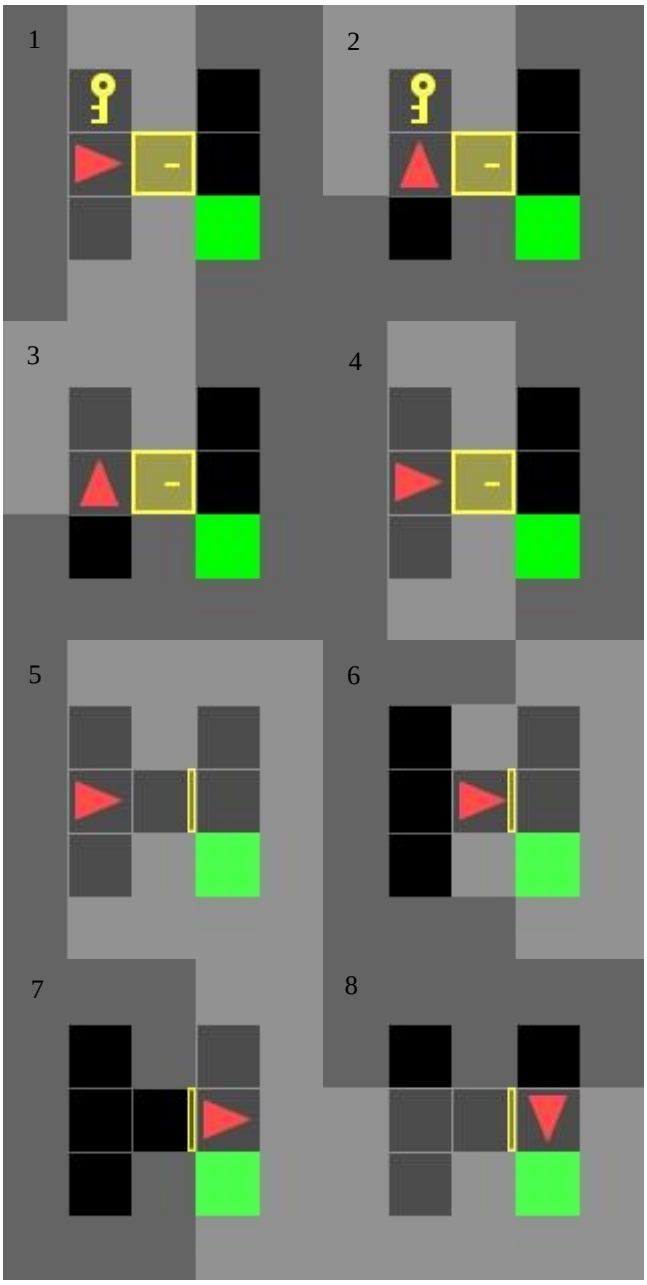
- 5X5 normal trajectory

In this scenario, the only way to go reach the goal is to go pick up the key and go through the door. We observe that the agent has taken the best possible path to achieve the goal. The control input sequence for the optimal policy is:

$$[1, 3, 2, 4, 0, 0, 2, 0]$$

or

$$[TL, PK, TR, UD, MF, MF, TR, MF]$$



Trajectory Visualization

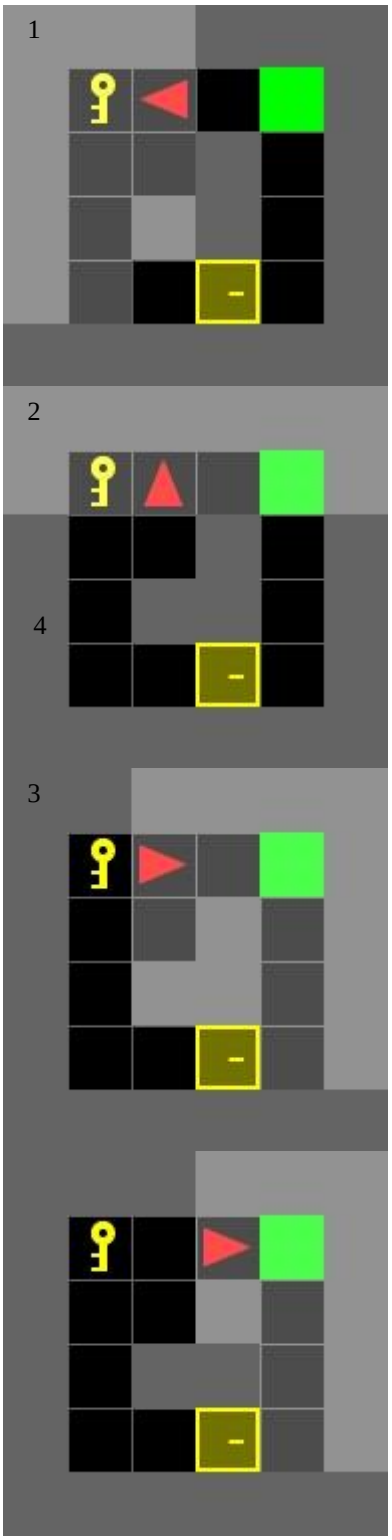
- 6X6 direct trajectory

In this scenario, the goal can be accessed directly without having to pick up the key and going through the door. We observe that the agent has just gone to the goal directly taking the best possible path. The control input sequence for the optimal policy is:

$$[1, 1, 0, 0]$$

or

$$[TL, TL, MF, MF]$$



Trajectory Visualization

- 8X8 shortcut trajectory

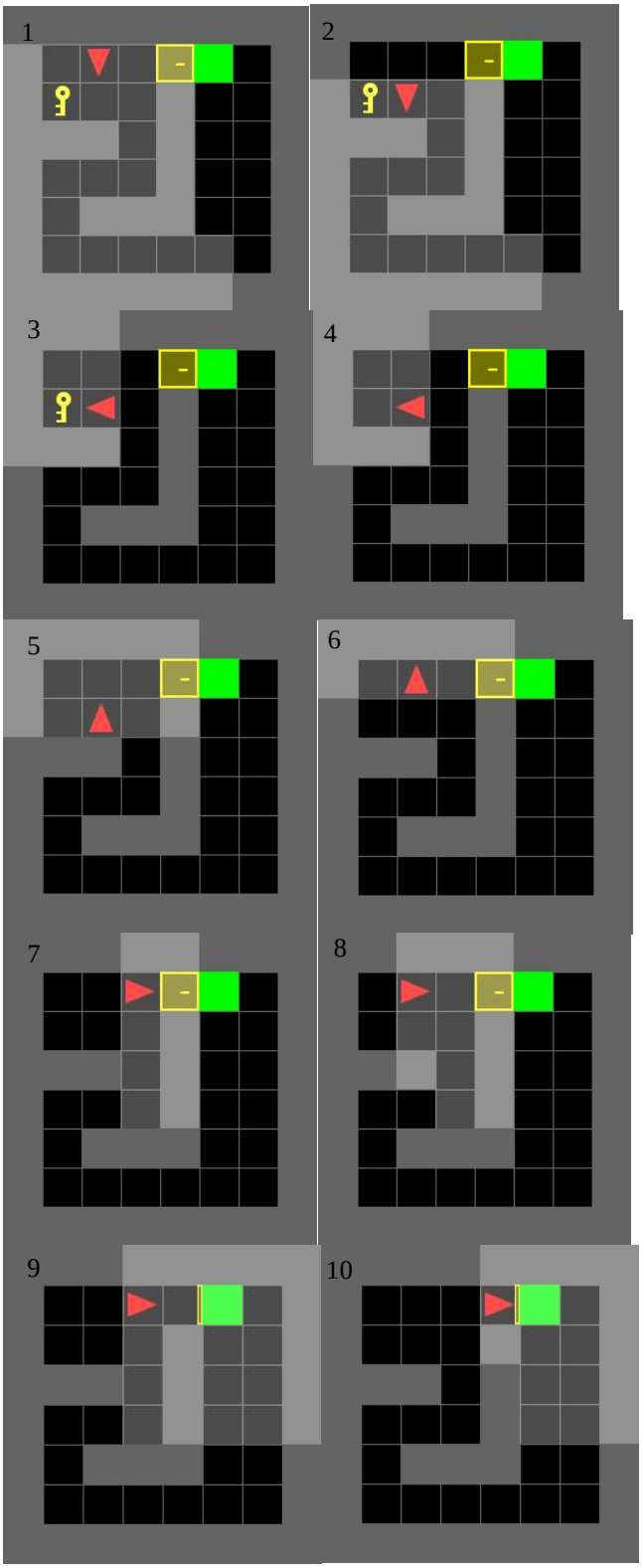
In this scenario, the goal can be reached directly or by picking up the key and going through the door, and we need to figure out the best possible path. We observe that the agent has taken the best possible path to achieve the goal by taking the key since that gives way for a shorter path than the longer route even if it is direct.

The control input sequence for the optimal policy is:

$$[2, 0, 1, 3, 1, 0, 0, 4, 0, 0]$$

or

$$[TR, MF, TL, PK, TL, MF, MF, UD, MF, MF]$$



Trajectory Visualization