ECE276B: Door Key Navigation using Dynamic Programming

Mustafa Shaikh
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, U.S.A

I. Introduction

As the role of autonomous robotic systems in our society expands, and these systems take on more varied tasks, the problem of autonomous navigation becomes ever more important. Autonomous navigation relates to the task of planning the optimal trajectory for agent to reach a desired goal state, while avoiding obstacles and constraints along the way. This has many uses in areas such as autonomous vehicles, where a car must reach its destination but adhere to traffic stops and avoid pedestrians. There are numerous other applications of this task such as in agriculture, manufacturing and military related applications. [4]

There are several approaches to solving the motion planning problem, most of which fit into categories such as exact methods, search-based methods and sampling-based methods. All approaches rely on an appropriate of representation of the environment. For complex environments, these approaches often include discretization of the environment using a variety of methods such as cell decomposition or lattice-based methods. However, in some cases, when the environment is relatively simple as in our doorkey project, the environment and state space is already discretized, and exists in a fairly low dimension. This allows us to find an exact optimal path from the start position to goal in a reasonable amount of time [4].

In this project, the task is to generate the optimal (in this case shortest) path from the initial point to the goal location while considering obstacles including walls and locked doors. Since the state space and environments are relatively simple, we use dynamic programming to solve for this optimal path. The rest of the paper is organized as follows: we present the problem formulation in Section II, the technical approach and implementation details in Section III, and the results of the algorithm in known and random environments are presented in Section IV.

II. Problem Formulation

In this project, the objective is to find the optimal path for the agent (red triangle) to the goal location (green square). There are two kinds of environments; one in which the location of the key and goal are known, and only a single door which is always locked. The other type of environment has 3 possible locations for the key and the goal, and there are 2 doors, either of which may be open or closed. If the door is locked, the agent must pick up a key to unlock the door. The agent has five possible actions available at any time: move forward (MF), turn left (TL), turn right (TR), pick up key (PK) and unlock door(UD). Each of these actions has a cost associated with it. The aim is to find the lowest cost sequence of actions that takes the agent from the start position to the goal. NOTE: in our formulation, we assume that the agent has reached the goal when it is in a cell adjacent to the goal and is facing the goal. In other words, we do not require the agent to step into the goal, but this will not change the optimal policy as the final step into the goal is a trivial step once the optimal policy is computed.

Motion planning is a deterministic shortest path problem, that can be considered a finite state, finite horizon optimal control problem by reformulating it as a Markov Decision Process (MDP) [1]. The motivation for this reformulation is that once a problem has been formulated as an MDP, it can be solved using methods such as Dynamic Programming that can find an optimal policy. MDPs require the precise definition of several quantities. We first present the formulation for part A, and then move on to formulation for part B.

1) Known environments:

- State Space (S): The state of the agent is represented as a tuple [kd,o,x,y], where $kd \in \{0, 1, 2, 3\}$ represents all combinations of carrying the key and door status. If kd = 0, the agent is not carrying the key and the door is open. If kd = 1, the agent is carrying the key and the door is open. If kd = 2, the agent is not carrying the key and the door is closed. Finally, if kd = 4, the agent is carrying the key and the door is closed. (x,y) refers to the agent's position in the grid, with $x \in \{0, 1, 2, .., w\}$ and $y \in \{0, 1, 2, .., h\}$, where w and h are the width and height, and (0,0) corresponds to top left of the grid. $o \in 0, 1, 2, 3$ represents the agent's orientation, i.e. the direction it is facing, with the integers corresponding to right, up, left, down in that order. The state space is of size 4*4*h*w (4 orientations, 4 key/door combinations, h*w cells)
- Control Space (U): In this environment, there exist 5 possible actions: move forward (MF), turn left (TL), turn right (TR), pick up key (PK) and unlock door(UD).
- Motion model, p_f: This model determines the transitions from one state to the next $s_{t+1} \sim p_f(s_t, u_t)$. Note that since this is a deterministic problem, there is no

randomness in the transition and hence no noise term. In other words, a specific action in a specific state always leads to the same next state. Note that in our formulation, we specify a naive motion model, in that we do not restrict the motion model based on reachable states. This will be taken care of by appropriately manipulating the stage costs, and serves to simplify the implementation. The motion model is fully described below:

- Action = MF:
- $s_{t+1} = [kd_t, o_t, x_t + 1, y_t]$ while facing right
- $s_{t+1} = [kd_t, o_t, x_t - 1, y_t]$ while facing left
- $s_{t+1} = [kd_t, o_t, x_t, y_t + 1]$ while facing down
- $s_{t+1} = [kd_t, o_t, x_t, y_t - 1]$ while facing up

- Action = TL:
- $s_{t+1} = [kd_t, 1, x_t, y_t]$ while facing right
- $s_{t+1} = [kd_t, 3, x_t, y_t]$ while facing left
- $s_{t+1} = [kd_t, 0, x_t, y_t]$ while facing down
- $s_{t+1} = [kd_t, 2, x_t, y_t]$ while facing up

- Action = TR:
- $s_{t+1} = [kd_t, 3, x_t, y_t]$ while facing right
- $s_{t+1} = [kd_t, 1, x_t, y_t]$ while facing left
- $s_{t+1} = [kd_t, 2, x_t, y_t]$ while facing down
- $s_{t+1} = [kd_t, 0, x_t, y_t]$ while facing up

- Action = PK:
- $s_{t+1} = [1, o_t, x_t, y_t]$ if not carrying key and door is open
- $s_{t+1} = [3, o_t, x_t, y_t]$ if not carrying key and door is closed
- $s_{t+1} = [kd_t, o_t, x_t, y_t]$ if carrying key and door is open (state remains the same)
- $s_{t+1} = [kd_t, o_t, x_t, y_t]$ if carrying key and door is closed (state remains the same)
- Note that under the action of picking up the key, the state can never transition to a state where agent does not have the key i.e. picking up key always leads to a state with the agent carrying the key

- Action = UD:
- $s_{t+1} = [kd_t, o_t, x_t, y_t]$ if not carrying key and door is open (state remains same)
- $s_{t+1} = [kd_t, o_t, x_t, y_t]$ if carrying key and door is open (state remains same)
- $s_{t+1} = [kd_t, o_t, x_t, y_t]$ if not carrying key and door is closed (state remains the same)
- $s_{t+1} = [1, o_t, x_t, y_t]$ if carrying key and door is closed (door switches to open and still carrying key)

- Planning horizon, T: This will be the number of states - 1 since in the worst case, the start state could be the furthest from the goal. So T equals total number of states in the state space - 1.
- Terminal state cost, q: This is initialized to infinity for all states except the states that are facing the goal. For example, if the agent is above the goal and is facing down, then the terminal cost of this state = 0.
- Stage cost, $l(s_t, u_t)$: This quantity represents the cost of taking an action u_t when the agent is in state s_t.
- The initial state s_0 differs for each environment in the known maps and is known
- discount factor, $\gamma = 1$ since this is a finite horizon problem

2) Random environments: We now move to formulating the second part of the project, in which there are 3 possible positions for the key and the goal, and there are 2 doors, either of which may be open or locked. We observe that we can reduce this problem to a minor extension of the first part by simply incorporating the possible key and goal locations into the agent's state. This means we expand the agent state to include the 3 possible key location, 3 possible goal locations, and 4 new key/door combinations in addition to the 4 from part A. Therefore the size of the state space has increased drastically. We present the formulation below.

- State Space, (S): The state of the agent is represented by a tuple [kd,o,k,g,x,y]. Since there are now 2 doors, $kd \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ represents all $2^3 = 8$ combinations of the binary states of the 2 doors and whether or not the agent is carrying the key. The variable $k \in \{0, 1, 2\}$ encodes the possible key positions, and $g \in \{0, 1, 2\}$ encodes the possible goal locations. Note that these variables related to the environment are now part of the agent's state, and though it leads to a much bigger state space, allows us to reduce the problem to that of part A with the expanded state space. The state space is now of size 8*4*3*3*h*w.
- Motion model: The motion model is extended to account for the extra door, though the logic remains identical. Essentially, picking up the key does not change the door status, and if the agent is already carrying the key, it does not change the state. Similarly, if the agent tries to unlock door 1, this does not affect the state of door 2, and vice versa. An important point to note here is that under any action, the state variables k,g that represent the possible key and goal positions to not change. For example, MF with the key and goal in state 0,0 will not change their position. This realization is important as it will allow us to reduce te complexity of calculations during the solution phase.

Once the MDP has been formulated, we can now state the optimal control problem in terms of the value function $V_t^\pi(x)$ that represents expected long term cost of being in state s, and the optimal policy π^* that represents the optimal actions to take in a given state in order to reach the goal with the lowest cost. The value function is defined as follows:

$$V_t^\pi(s) = E_{s_{t+1:T}}[q(s_T) + \sum_{\tau=t}^{T-1} l(s_\tau, \pi_\tau(s_\tau))|s_t = s]$$

Where π represents the mapping from states to actions. The expectation is not needed as this is a deterministic problem.

The optimal policy is given by:

$$\pi^* = \text{argmin} V_0^\pi(s_0)$$

where π^* corresponds to the sequence of actions that will result in the lowest cost path to the goal. Our aim is to find this optimal sequence of actions that will lead to the lowest cost path from the start position to the goal. Now that the problem has been formulated, we move on to the technical approach.

III. Technical Approach

Once the problem is formulated as an MDP, we can apply the Dynamic Programming algorithm which guarantees the optimal solution for a finite horizon optimal control problem, similar to the one we have formulated above [2]. First we present the DP algorithm, and then move on to a discussion of the definition of the stage cost and terminal cost, as well as how to extract the optimal policy.

A. Dynamic Programming Algorithm

Below we present pseudo-code of the DP algorithm as relates to our implementation. We also discuss the method of extracting the optimal policy. Note that our method for extracting the optimal policy is as follows: as per the algorithm above, we check the value function at the starting state for all times t. Note that until the start state is reachable, its value will be infinity. Once it is reachable, it will have a finite value, and this will occur at the first instance that it is reachable. We pick out this time index, and obtain the optimal action at that time index. We apply this optimal action through the motion model to our start state, and repeat until we reach the end of the loop. This results in the optimal action sequence for the given start state.

Data: State Space S, Door Positions, Key Position, Action Space A, Stage Cost Matrix L.
Result: Optimal Action Sequence
$V_T(s) \leftarrow \infty, \forall s \in S_T$ except $s \in S_T$ where s faces goal from any direction, where S_T is terminal state set.
for *time t in* $(T-1):0$ **do**
  $Q_t(s,u) = L(s,u) + V_{t+1}(p_f(s)), \quad \forall s, u \in S$
  $V_t(s) = \min_{s \in S} Q_t(s,u)$
  $\pi_t^*(s) = \text{argmin}_{s \in S} Q_t(s,u)$
end
$Pathlength \leftarrow$ first occurrence of finite value in $V_{0..T-1}(startstate)$
for *i in range(T-1-pathlength,T-1,step size = -1)* **do**
  append $u \in \pi_t^*(s)$ to optimal action sequence
  apply motion model under optimal action u in previous step to state s to obtain new state s i.e. $s = p_f(s_prev, u)$
end

Algorithm 1: Dynamic Programming algorithm

B. Stage cost, terminal cost and motion model

1) Terminal cost: The terminal cost is initialized to infinity for all states except those states that are facing the goal, and are adjacent to the goal; these states are set to 0 to allow them to have a finite value in the Q function and therefore allow DP to send the agent to these states. This includes facing down from above the goal, facing right from the left of the goal, facing up from below the goal, and facing left from the right of the goal. All these states are to be permissible as we do not know the optimal path beforehand. Also, we allow any key/door combination at the goal state, as we do not know beforehand whether the door is locked or not, and whether the agent needs the key or not. The general philosophy is to let the stage costs handle most of the constraints and keep everything else as simple as possible. In the random environment case, the stage cost is the same except now it is defined in the extended case considering all 3 possible goal locations. When the environment is loaded, the appropriate index for the goal location is selected and the corresponding terminal cost is selected.

2) Stage cost: There are several cases for which the stage cost is defined. For each action, if the state s_{t+1} is reachable with the action u_t applied to s_t, then the cost $l(s_t, u_t) = 1$. If a state is not reachable with a given action, or if we wish to prevent the agent from taking a particular action in a particular state, $l(s_t, u_t) = \infty$. This works because an infinite cost prevents this action from ever being selected in the argmin step in the DP algorithm. The cases are discussed below.

- Walls: If the agent is facing a wall, the MF action is set to ∞ for both known and random environments. TL and TR are always allowed in front of a wall. PK and UD are disallowed so set to ∞ (these cases are discussed in more detail below).
- Key: We initialize the stage costs of the action PK to ∞ by default since there are only a few cases in which we wish to allow the agent to pickup the key. If the key is in the cell in front of the agent, and the agent does not have the key, and the door is closed, then the agent MUST pickup the key AND do nothing else (this means defining the complementary event). This means it cannot MF, TL, TR or UD so the costs for these actions are set to ∞. If the door is open, the agent must not pickup the key so PK is set to ∞ in this case as we do not wish to waste cost by picking up the key unnecessarily. For the random environment case, the logic is the same, but now if both doors are closed, it must pickup the key, and if either door is open, it is allowed to pickup the key but not forced to. This is to allow it to make the optimal decision based on the shortest path from goal to start.
- Door: UD action is defaulted to ∞ except in a handful of specific cases. If the agent is facing the door and the door is locked and it has the key, it must unlock the door and do nothing else (i.e. it cannot MF, TL, TR, PK). If

any of these conditions is not met, it is forbidden from unlocking the door so the costs are set to ∞ for UD. This is to force it to pickup the key before unlocking the door. In the random environment case, the logic is extended to the second door. For example, if the agent is in front of the second door with the key and the second door is locked, then it must unlock the second door regardless of the state of the first door, and vice versa. Similarly, if the door is open and the agent is in front of it, it must not unlock it (do not waste cost on unlocking an open door), so we set the cost of UD in that case to be ∞.

- Out of bounds: We pad the cells with a layer of dummy cells all around the grid, and index the top left grid cell as (1,1) rather than (0,0). This is because we use a naive motion model implementation in which the next state is allowed to be outside the grid. This is a design choice, as we have opted to keep the motion model simple and instead handle all constraints in the stage costs so there is only one potential source of error. MF while facing any cell outside the grid is set to ∞ to prevent the agent from going out of bounds.

3) Motion model: As mentioned earlier, we use a naive motion model implementation for simplicity. For example, we allow the agent to go out of bounds if it is at the boundary and facing outside the grid, we allow it to MF. We also allow it to MF when facing a locked door. We also allow it to pickup the key in a cell that the key is not present in. Instead, these invalid actions are prevented in the stage costs. Recall that we initialized all PK actions to ∞ except a small subset of situation in which the agent is allowed to pickup the key. This ensures these invalid states are never reached. This design choice significantly simplified the implementation and made debugging much easier.

C. Technical Details

We use a MacBook Pro with 1.4 GHz Quad-Core Intel Core i5, 8 GB 2133 MHz LPDDR3, Intel Iris Plus Graphics 645 1536 MB. We make use of the well known numpy library for data manipulation.

IV. Results

We now present the results for the DP implementation in both the known and random environments. Specifically, we report the optimal actions sequences for each environment, and discuss some noteworthy cases. We also demonstrate the trajectory for a small sample of environments to highlight the path taken by the agent. To keep the report concise, we have not plotted all 36 random environments and all known environments, but rather a subset to highlight important points. In general, the DP algorithm was able to find the optimal action sequence in the known environments with DP calculated for each map separately, as well as in the case of a single optimal policy for the random environments. Note again that in our implementation, we assumed that the agent has attained the goal position when it is in the cell right in front of the goal and is facing the goal. So every action sequence can have an MF appended to it if we wish the agent to be inside the goal position. For the known environment sequence, the final MF action has been appended.

1) Known environments: For each known environment, the DP algorithm found the optimal path. Here are the optimal action sequences for each environment. Some noteworthy cases are plotted below, with the agent's trajectory drawn onto the map. The optimal policy is computed separately for each environment, and as such is different for each environment.

- 5x5 normal: TL, TL, PK, TR, UD, MF, MF, TR, MF
- 6x6 direct: MF, MF, MF, TR, MF, MF
- 6x6 normal: TL, MF, PK, TL, MF, TL, MF, TR, UD, MF, MF, TR, MF
- 6x6 shortcut: PK, TL, TL, UD, MF, MF
- 8x8 direct: MF, TL, MF, MF, MF, TL, MF
- 8x8 normal: TR, MF, TL, MF, TR, MF, MF, MF, PK, TL, TL, MF, MF, MF, TR, UD, MF, MF, MF, TR, MF, MF
- 8x8 shortcut: TR, MF, TR, PK, TL, UD, MF, MF

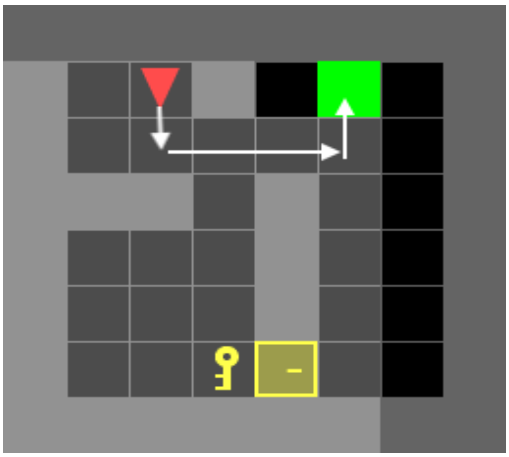

Fig. 1. This case is a good test of whether the agent can realize it should take the direct route rather than unlock the door. In our case, it successfully realizes this. Initially during the implementation, we found that the agent was taking the long route to pick up the key, and this was because we had forced the terminal state to have the door open. Once we relaxed this constraint, the agent took the shortcut.

The optimal policy is computed in a very short space of time due to the relatively small state space (4*4*h*w).

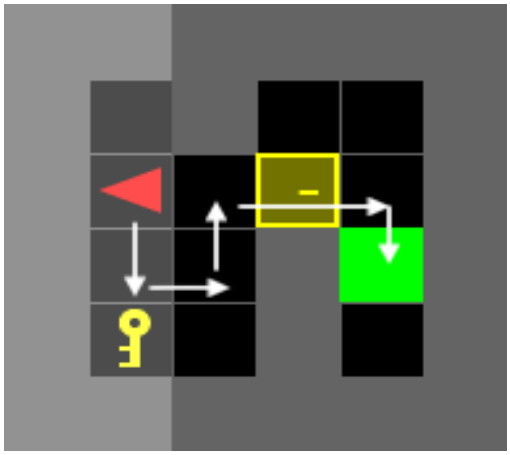

Fig. 2. Standard case showing that the agent goes straight to the key rather than trying to go to the door first. Then turns around and correctly unlocks the door and goes to the goal.

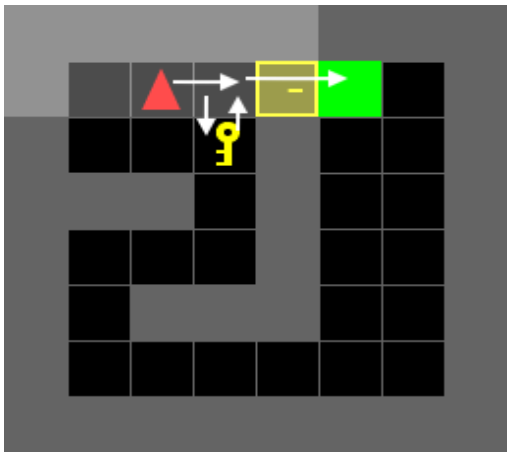

Fig. 3. Basic example showing the agent's trajectory. Agent recognizes that there is a short path that involves unlocking the door rather than taking the unobstructed but longer route.

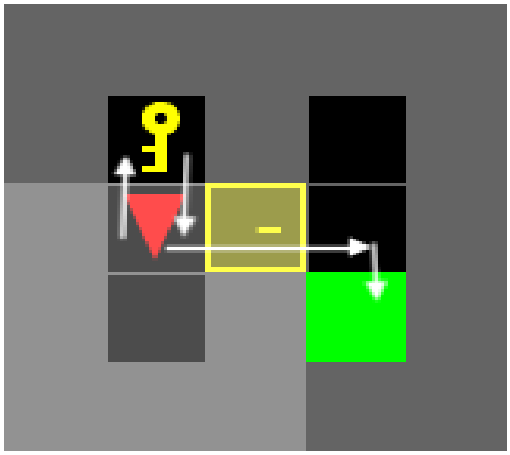

Fig. 4. Standard case showing the agent performing multiple turns in place to obtain the key, turn back around to unlock the door, and then move forward. It does not try to move forward before unlocking the door, which is correct behaviour.

2) Random environments: In contrast to the known environments, a single optimal policy is computed for the random environment. This means that the DP algorithm was only run once, and when the environment is loaded, we simply pick out the states that correspond to the given key and goal positions. The algorithm was able to successfully generate optimal paths for all 36 random maps. Note that the final 'MF' action has not been appended here and instead the agent stops when facing the goal. Some noteworthy cases are plotted and discussed below, along with the agent's trajectory.

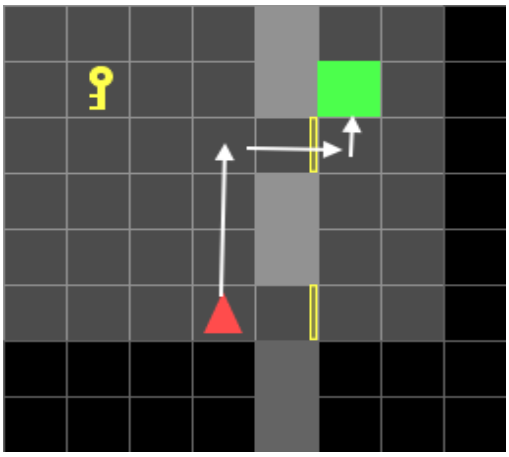

Fig. 5. Standard case showing agent going through open door while correctly ignoring the key. Since going through either door has the same cost, the DP algorithm has arbitrarily chosen the tie break when computing the argmin for the optimal action sequence.

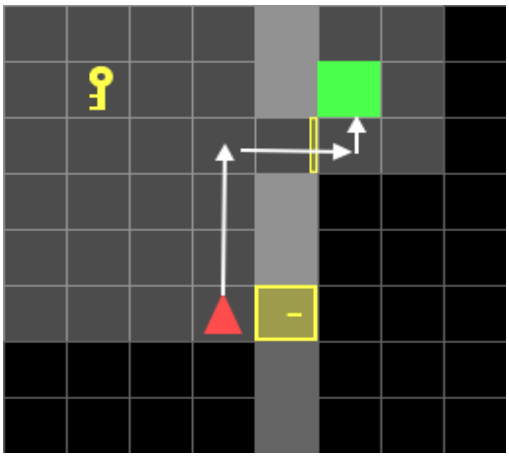

Fig. 6. Standard case showing agent going through the open door rather than picking up the key and trying to unlock the locked door.

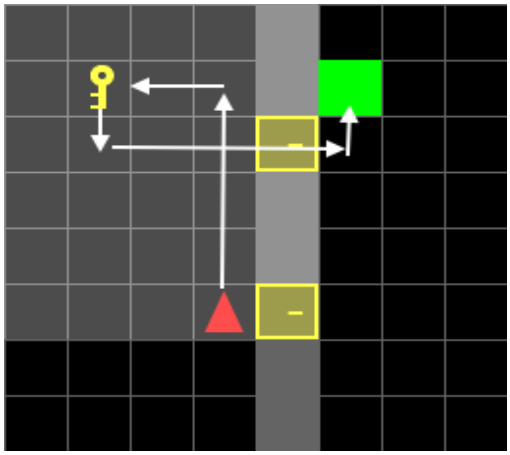

Fig. 7. An interesting case, where the agent picks up the key in an efficient manner and goes to the closest door to unlock it rather than going to the further door at the bottom of the map.

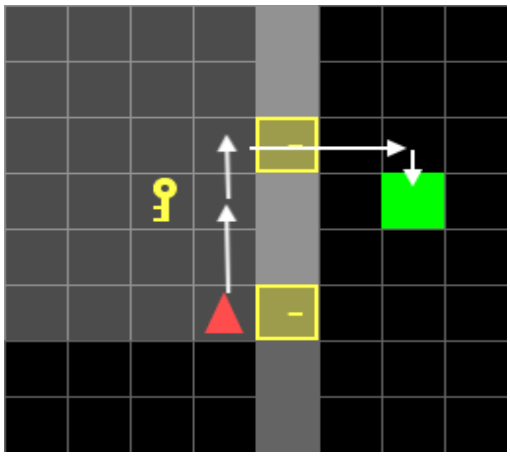

Fig. 8. Agent picks up the key on the way to the upper door rather than turning around back to the lower door, which would lead to a higher cost

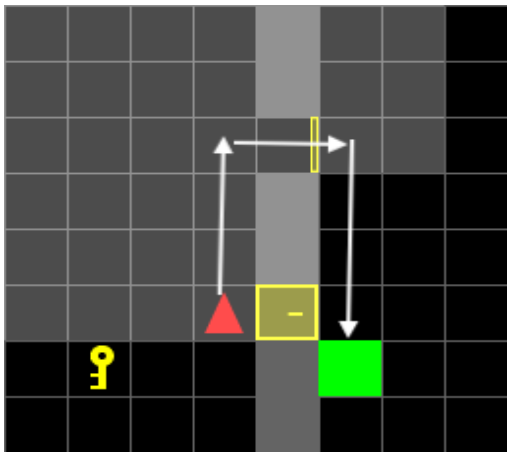

Fig. 9. One of the most interesting cases. At first, the path through the open door seems longer than picking up the key and unlocking the door. However, if the action sequences are compared, the key/unlock door route is 1 unit of cost higher than going through the open door. This shows the algorithm works well even in cases that are very close.

- 1: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL']
- 2: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL']
- 3: ['TR', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF']
- 4: ['MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', 'TL', 'MF', 'TL', 'MF', 'UD', 'MF', 'MF', 'TL']
- 5: ['TR', 'MF', 'MF', 'MF', 'TL', 'MF']
- 6: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF', 'TR']
- 7: ['TR', 'MF', 'MF', 'MF', 'TL', 'MF']
- 8: ['MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', 'TL', 'MF', 'TL', 'MF', 'UD', 'MF', 'MF', 'MF', 'TR']
- 9: ['TR', 'MF', 'MF', 'TR']
- 10: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
- 11: ['TR', 'MF', 'MF', 'TR']
- 12: ['MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', 'TL', 'MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'UD', 'MF', 'MF', 'TR']
- 13: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL']
- 14: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL']
- 15: ['TR', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF']
- 16: ['MF', 'MF', 'TL', 'PK', 'TR', 'MF', 'TR', 'UD', 'MF', 'MF', 'TL']
- 17: ['TR', 'MF', 'MF', 'MF', 'TL', 'MF']
- 18: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF', 'TR']
- 19: ['TR', 'MF', 'MF', 'MF', 'TL', 'MF']
- 20: ['MF', 'MF', 'TL', 'PK', 'TR', 'MF', 'TR', 'UD', 'MF', 'MF', 'MF', 'TR']
- 21: ['TR', 'MF', 'MF', 'TR']
- 22: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
- 23: ['TR', 'MF', 'MF', 'TR']
- 24: ['MF', 'MF', 'TL', 'PK', 'TL', 'MF', 'MF', 'TL', 'UD', 'MF', 'MF', 'TR']
- 25: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL']
- 26: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL']
- 27: ['TR', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF']
- 28: ['TL', 'MF', 'MF', 'TL', 'PK', 'TL', 'MF', 'MF', 'UD', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF']
- 29: ['TR', 'MF', 'MF', 'MF', 'TL', 'MF']
- 30: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF', 'TR']
- 31: ['TR', 'MF', 'MF', 'MF', 'TL', 'MF']
- 32: ['TL', 'MF', 'MF', 'TL', 'PK', 'TL', 'MF', 'MF', 'UD', 'MF', 'MF', 'MF', 'TL', 'MF']
- 33: ['TR', 'MF', 'MF', 'TR']
- 34: ['MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
- 35: ['TR', 'MF', 'MF', 'TR']
- 36: ['TL', 'MF', 'MF', 'TL', 'PK', 'TL', 'MF', 'MF', 'UD', 'MF', 'MF', 'TR']

Due to the enlarged state space (8*4*3*3*h*w), the computation time was longer than for the known environments. However, we exploited a piece of information that is available to us to keep the computation time fairly reasonable. We notice that the environments are not truly random, and the most important observation is that there can never be a state

transition between states in which they keys and goals are in different places. For example, we can never have the key change places just by moving forward. Therefore, since there is no linkage between the states that encode the different key/goal positions, the value function and action sequence can be computed in parallel, and at runtime, when the environment is loaded, we simply pick out the appropriate index. This is still a single optimal policy as the DP algorithm is only computed once (not 36 times), but we use the fact that there is no inter-linkage of certain states to keep computation time low.

V. Conclusion

We have showed that the DP algorithm is capable of providing the optimal path for all known and random maps once the problem is formulated as an MDP. One very interested extension of this work would be to tweak the stage costs of particular actions. Currently, each action has a cost of 1, but if the costs associated with certain actions are changed, this can dramatically change the optimal action sequence. For example, if the cost of picking up the key and unlocking the door is increased, it will encourage the agent to seek open door paths rather than take shortcuts. This can lead to very interesting analyses.

We also note that the DP algorithm computes the optimal path from every single starting position, and this leads to wasted computation time. In the future, we can experiment with using more efficient planning algorithms such as label correcting and A* algorithms to reduce unnecessary computation.

References

[1] N. A. Atanasov, "Markov Decision Processes," in ECE 276B Lecture 3.

[2] N. A. Atanasov, "The Dynamic Programming Algorithm," in ECE 276B Lecture 4.

[3] N. A. Atanasov, "Deterministic Shortest Path," in ECE 276B Lecture 5.

[4] "Motion planning," Wikipedia, 17-Jan-2022. [Online]. Available: https://en.wikipedia.org/wiki/Motion_planning. [Accessed: 01-May-2023]