MATLAB code description by walking through an experiment

Min Wen

July 31, 2019

In this document, I briefly show the results for the example in our IJCAI paper. The results can be done by running `run.m` directly, given that `minFunc` is installed properly. `minFunc` is an open-source MATLAB package for gradient descent algorithms that is available online. Please download the package and record the saved path to it (nothing to be installed). Make sure to modify the path to `minFunc` and the current MATLAB path in `run.m`.

config.m

Most of the configuration parameters. I have added comments for each variable. This file should be run before all the others.

One thing needs special notice: There is a `AP_struct` variable in this file, which maps from each color to an integer. This variable will be used in `product_automaton.m`, when we set the value of `label_to_trans` there.

mdp.m

The environment MDP is shown in Figure 1.

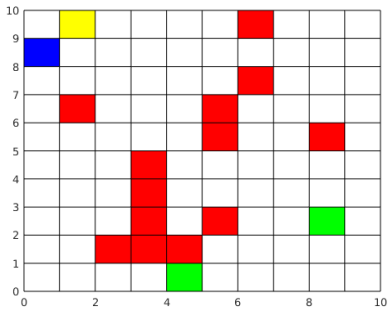


Figure 1: Environment map. The set of atomic propositions for this example is $\{b, g1, g2, r, y, w\}$.

In `mdp.m`, we construct a square grid world of size `grid_dim`-by-`grid_dim`, where `grid_dim` is specified in `config.m`. From each state, there are four actions: Up, Right, Down, Left. The transitions are stochastic: For each action, the probability to go in the consistent direction is `p_center`, while the probabilities to other directions is `p_others`. These two parameters are also specified in `config.m`.

dfa.m

The task specification for this example is

$$\varphi = (\mathsf{G}\ !r)\&(\mathsf{F}\ g1)\&(\mathsf{F}\ g2)\&(!y\ \mathsf{U}\ g1)\&(!y\ \mathsf{U}\ g2)\&(\mathsf{F}\ y)$$

In other words, the agent should always keep away from red states, visit both two green states and eventually visit the yellow state. The DFA is actually an DRA generated using `ltlfilt`. After running `dfa.m`, check the struct `dfa_struct` for details of the DRA. In this example, the result is

```
dfa_struct =

  struct with fields:

           N_p: 4
      state_no: 6
         trans: [65 double]
         pairs: 1
             K: {[1]}
             L: {[2 3 4 5 6]}
            S0: 5
           APs: {'r'  'g1'  'g2'  'y'}
   AP_to_color: {'yellow'  'green2'  'green1'  'red'}
```

So we know that the initial state of the DRA is state 5. There are 6 states in total, where only state 1 should be visited infinitely often. `trans` is the transition matrix, where each row corresponds to a DRA state and each column corresponds to an AP.

It should be noted that we have assumed that at most one AP can be true at each state, and thus the number of columns has been reduced from $2^{|AP|} = 2^4 = 16$ to $|AP|+1 = 4+1 = 5$. Each column in `trans` corresponds to a label in `APs` in the **reverse order** (according to Zhe). In this example, the first column in `trans` corresponds to 'y'; the second column corresponds to 'g2'; so on so forth. This relation is tracked by the variable in `dfa.m`. Since I cannot predict or specify `APs`, `dfa_trans` **needs to be manually specified**.

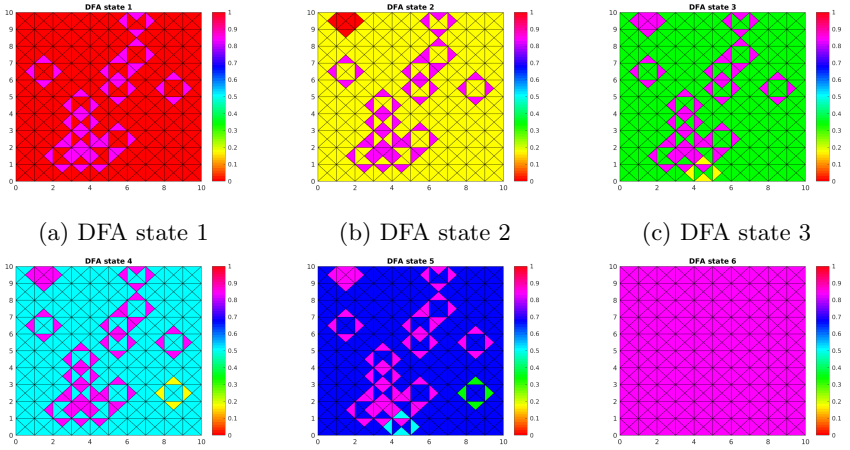
product_automaton.m

After running `mdp.m` and `dfa.m`, we construct a product automaton in this file. Specifically, we construct a big `transition` matrix for the product automaton.

There is one thing to be noted. **We need to manually specify the relation between labels in the environment MDP and APs in the DFA. The list saving this relation is** `label_to_trans`, which is of the same size as `AP_to_color` in `config.m`. `label_to_trans` maps each color in `AP_to_color` to its corresponding column index in `dfa_struct.trans`. In this example,

```
% AP\_struct = struct( 'blue', 1,    'yellow', 2,    'red', 3, ...
%                      'green1', 4,  'green2', 5,    'white', 6);
% dfa\_APs = {'yellow', 'green2', 'green1', 'red'};]
label\_to\_trans = [5, 1, 4, 3, 2, 5];
```

The first label in `AP_struct` is blue ('b'), which does not appear in `dfa_struct.AP_to_color`, so it corresponds to the last column. The same thing happens to the last label in `AP_struct`, which is white ('w'). The second label in `AP_struct` is yellow ('y'), which is the first one in `dfa_struct.AP_to_color`, so the second element in `label_to_trans` is 1. The third label in `AP_struct` is red ('r'), which is the fourth one in `dfa_struct.AP_to_color`, so the second element in `label_to_trans` is 4.



(a) DFA state 1 (b) DFA state 2 (c) DFA state 3

(d) DFA state 4 (e) DFA state 5 (f) DFA state 6

Figure 2: Illustrations of transitions in the DFA. Each state is decomposed into four parts (up, right, down, left), where the color of each part shows the DFA state of the next state by taking the corresponding action from this state. Please ignore the colorbar here.

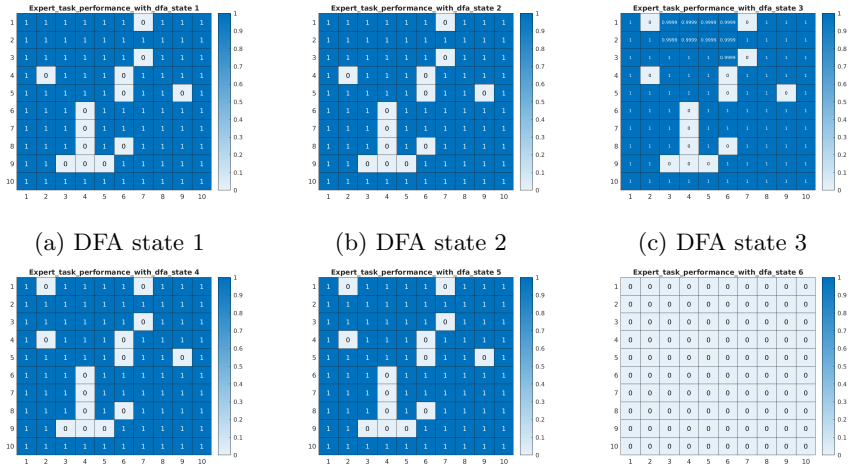
We also generate a set of reward features `F` in this file. We automatically construct one reward feature for each transition in the DFA. The transitions can be visualized by setting `testing` to be 1 in `config.m` and then run this file. For this example, the transitions are illustrated in Figure 2

If we need to construct a product automaton with more than one DFA, we need to specify one `label_to_trans` vector for each DFA. Then we can construct the transition matrix for the product automaton similarly by writing two `for` loops: one for action and one for state. Given a state (mdp state, DFA state for each DFA) and an action, read the MDP index of the next state, the label of the next state, the successor state in each DFA, decide the index of the next state in the product automaton, and save it.

generate_demos_4.m

After constructing the product automaton, we can generate the expert demonstrations. My current way to do this is to **manually** specify a reward function, then solve the corresponding expert policy, and then simulate the expert policy. For each reward function, we can compute the corresponding policy and evaluate it by `evaluateReward.m`. Expert demonstrations are saved in variable `demo_trajs`, whose dimension is $\#\_of\_demonstration\_transitions$-by-2 (for state-action pairs). The length of each demonstration trajectory and the number of demonstration trajectories are specified in `config.m`.

Here is the task performance of the expert policy. Figure 3 shows the prob. of satisfying φ from each state in the product automaton.



(a) DFA state 1 (b) DFA state 2 (c) DFA state 3

(d) DFA state 4 (e) DFA state 5 (f) DFA state 6

Figure 3: Task performance for expert policy for each state in the product automaton. Dark blue is probability 1; white is probability 0.

maxEntIRL.m

We implement the maximum entropy IRL algorithm here.

$$\begin{aligned} Q_\theta(s,a) &= R_\theta(s,a) + \gamma \sum_{s' \in S} T(s'|s,a) \log \sum_{a'} \exp(Q_\theta(s',a')) \\ R_\theta(s,a) &= F(s,a)\theta. \end{aligned} \tag{1}$$

The policy π_θ can be represented as the explicit expression of Q_θ in (2).

$$\pi_\theta(a|s) = \frac{Q_\theta(s,a)}{\sum_{a'} \exp(Q_\theta(s',a'))}. \tag{2}$$

The goal is to minimize the negative log likelihood of expert demonstrations.

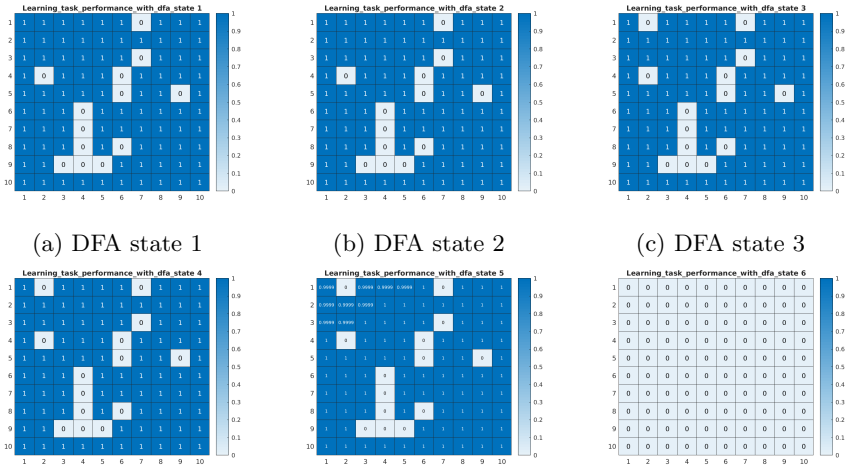
$$\min_\theta L_D(\theta) = \min_\theta - \sum_{\tau_i \in D} \sum_{j=0}^{n_i - 1} \log \pi_\theta(a_{i,j}|s_{i,j}). \tag{3}$$

Given `F` and the expert demonstrations, we solve the best θ by gradient descent. The gradients are

$$\begin{aligned} \frac{\partial L_D}{\partial \theta} &= \frac{\partial}{\partial \theta} \sum_{i=1}^{N} \sum_{l=0}^{n_i} \log \pi_\theta(s_{i,l}, a_{i,l}) = \sum_{i=1}^{N} \sum_{l=0}^{n_i} \Big(\frac{\partial Q_\theta(s_{i,l}, a_{i,l})}{\partial \theta} - \sum_{a'} z_\theta(s_{i,l}, a') \Big) \\ z_\theta(s,a) &= \pi_\theta(a|s) \frac{\partial Q_\theta(s,a)}{\partial \theta} \\ \frac{\partial Q_\theta(s,a)}{\partial \theta} &= \frac{\partial R_\theta(s,a)}{\partial \theta} + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi_\theta(a'|s') \frac{\partial Q_\theta(s',a')}{\partial \theta} \\ \frac{\partial \pi_\theta(a|s)}{\partial \theta} &= z_\theta(s,a) - \pi_\theta(a|s) \sum_{a'} z_\theta(s,a') \end{aligned} \tag{4}$$

In our code, θ is randomly initialized.

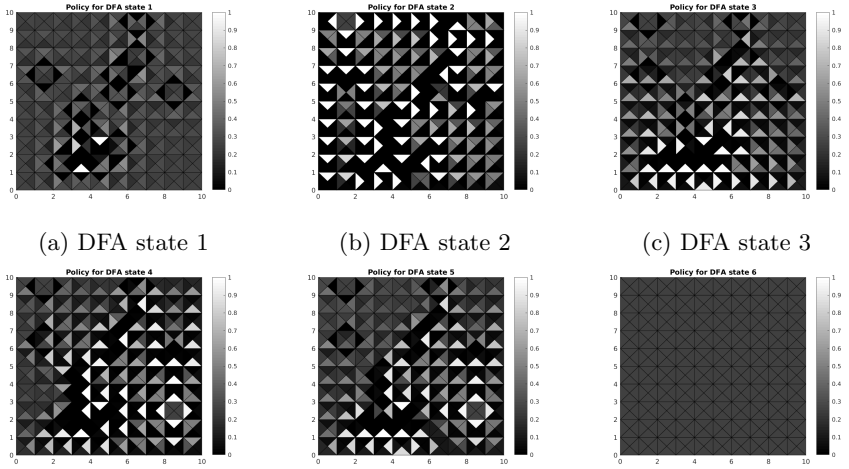
If we use 20 expert demonstrations, the task performance of the learned policy will be near-perfect, as illustrated in Figure 4. The learned policy is also illustrated in Figure 5.



(a) DFA state 1 (b) DFA state 2 (c) DFA state 3

(d) DFA state 4 (e) DFA state 5 (f) DFA state 6

Figure 4: Task performance for the learned policy for each state in the product automaton, if there are 20 expert demonstration trajectories. Dark blue is probability 1; white is probability 0.



(a) DFA state 1 (b) DFA state 2 (c) DFA state 3

(d) DFA state 4 (e) DFA state 5 (f) DFA state 6

Figure 5: Visualization of the learned policy. Each cell contains 4 colors, showing the probability to take the corresponding action from this state. White is probability 1; black is probability 0.