Lab on MDP
COMP4620/8620

This lab is meant to illustrate the fundamental notions of MDP. You are asked to implement some methods, but you will need to read the existing code. You are provided with some test files; passing the tests **does not** imply your implementation is correct.

Discuss your issues and solutions on the dedicated discord channel `https://discord.gg/N5WzsDWnQM` also used for lectures.

File `MDP.py`

This file provides the basic classes and functions for MDPs: `State` and `Action` are essentially two placeholders used for typing purposes. An `MDP` is defined by the following methods:

1. `states(self)` returns the list of states in the MDP.
2. `actions(self)` returns the list of actions in the MDP.
3. `applicable_actions(self,s)` returns the list of actions applicable in state `s`.
4. `next_states(self,s,a)` returns the list of possible effects of executing `a` in `s`; each effect is a triple `(s',p,r)` where `s'` is the next state, `p` is the probability of this effect, and `r` is the reward of the effect.
5. `initial_state(self)` returns the initial state of the MDP.

Notice that the list of states and the list of actions could be deduced from the other methods (for instance, the implementation of `state(self)` in the `DungeonMDP` class of file `map.py` is purely based on the other methods).

The file also includes the classes `Policy`, `ExplicitPolicy` (which is an implementation of a `Policy` that uses a dictionary), and `History`.

File `statemachine.py`

This file contains an implementation of MDPs as a state machine (`SMMDP`). Essentially, you enumerate the states in the MDP and the transitions between the states.

The file also includes a method `state_machine_from_mdp(mdp)` that takes an MDP as input and returns an SMMDP that is equivalent (bisimulation), together with the dictionary that explains what the SMMDP states and actions correspond to in the original MDP. The idea is that computing this MDP once and for all makes it easier to manipulate. If, for instance, a state is represented by a complex data structure, verifying the equality between two states may be expensive; with the translation into an SMMDP, each state is simply represented by a string and the comparison is computationally cheap. The problem is that we need to generate all reachable states, which some methods may have been able to avoid. In practice, we might want to build the SMMDP on the fly.

So consider an MDP M that contains states s_1, s_2, and s_3 and action a, and such that applying a in state s_1 leads to state s_2 with probability .7 and to s_3 otherwise. Then, this method will generate three objects: an MDP M', a dictionary D_S, and a dictionary D_A, such that:

- M' contains three states s'_1, s'_2, and s'_3 and an action a' such that applying a' in s'_1 leads to s'_2 with probability .7 and to s'_3 otherwise;
- $D_S[s'_i] = s_i$ for $i \in \{1, 2, 3\}$; and
- $D_A[a'] = a$.

Finally, the file provide a class `TranslatedPolicy` that translates a policy for a given MDP into a policy for a different MDP (given dictionaries between states and actions). So, to reuse the example above, if π' is a policy for M' and if $\pi'(s'_1) = a'$, then `TranslatedPolicy`$(\pi')(s_1, {D_S}^{-1}, D_A)$ will equal a.

File `example1.py`

This file contains an example of an MDP implemented as a state machine.

File `example2.py`

This file contains an example of an MDP implemented implicitly. This example is equivalent to the one of `example1.py` where the next states are computed at each iteration.

File `algos.py`

This file contains the algorithms that you are supposed to implement. It first contains classes to manipulate state value functions (represented by V in the lectures) and action value functions (represented by Q).

Implement the following methods (the methods are generally no longer than 10 lines each as they often depend on the previous methods):

- `simulate_one_step(mdp,state,action)`

  Simulates the execution of the specified action in the specified state. This returns the new state as well as the reward. Test your implementation with file `test01.py`.

- `simulate(mdp,pol,nbsteps)`

  Simulates the execution of the specified policy over the specified number of steps. Test your implementation with file `test02.py`.

- `greedy_action(mdp,q,s)`

  Given an action value function and an MDP, computes the greedy best action in the specified state. Test your implementation with file `test03.py`.

- `greedy_policy(mdp,q)`

  Given an action value function and an MDP, computes the greedy policy of the MDP. Test your implementation with file `test04.py`.

- `one_step_lookahead(mdp,v,gamma,s,a)`

  Computes the Q value of the specified pair of state/action given the specified state value function and discount factor. Test your implementation with file `test05.py`.

- `compute_q_from_v(mdp,v,gamma)`

  Computes the action value function as a one-step lookahead of the specified state value function with the specified discount factor. Test your implementation with file `test06.py`.

- `compute_v_from_q_and_policy(mdp,pol,q)`

  Computes the state value function given the action value function and the policy. Test your implementation with file `test07.py`.

- `bellman_backup(mdp,v,gamma)`

  Performs the Bellman backup on the specified MDP given a state value function, with the specified discount factor. Test your implementation with file `test08.py`.

- `compute_v_of_policy(mdp,pol,gamma,stopping_threshold,starting_value)`

  Computes the state value function of the specified policy given the specified discount factor by iteratively performing one-step lookahead. The stopping threshold is used to determine that the state value function is precise enough. The optional starting value function can be used to accelerate convergence. Test your implementation with file `test09.py`.

- `is_policy_nearly_greedy(mdp, pol, epsilon, q)`

  Determines whether the specified policy is nearly the greedy policy for the specified action value function, where "nearly" means that the value of the action chosen by the policy in any state is at most `epsilon` away for the greedy (best) value. This method is useful to guarantee termination of Policy Iteration because approximation errors can prevent convergence. Test your implementation with file `test10.py`.

- `policy_iteration(mdp, gamma, epsilon, stopping_threshold, starting_pi)`

  Performs the policy iteration on the specified MDP with the specified discount factor. `epsilon` is used to decide when the current policy is nearly greedy, and `stopping_threshold` is used to determine when the value function is precise enough. Test your implementation with file `test11.py`.

- `value_iteration(mdp, gamma, epsilon)`

  Performs the value iteration algorithm on the specified MDP with the specified discount. `epsilon` is used to determine when to stop. Test your implementation with file `test12.py`.

File `map.py`

This file presents an example of a domain.

In this domain, the goal is to explore a dungeon in order to collect treasures. You need to hire companions who will join you in your party, have them fight monsters, hire more companions if necessary, and collect the rewards!

(*A Dungeon Too Many*, Joann Sfar, Lewis Trondheim, Manu Larcenet)

Specifically, here are the rules.

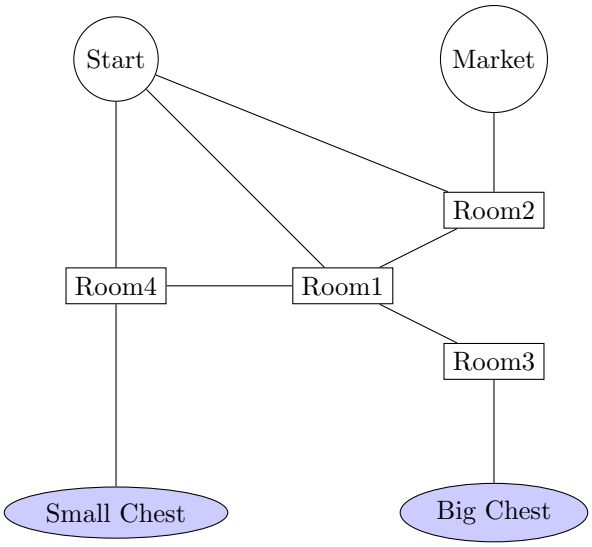

Figure 1: Example map: circles represent inns, ellipses represent chest rooms.

- A map is comprised of a number of rooms, which can be dangerous rooms (with a monster each), inns (in which you can hire companions), and chest room (in which you get some loot). Dangerous rooms and chest rooms become empty once you successfully entered them; in practice, this means that the monster of the dangerous room and the reward of the chest room are only relevant if the room does not appear in the list of visited locations.
- When in an inn, you can hire a companion. Each inn has a different choice of adventurers, with different prices. You can hire as many adventurers of any type as you want, but your party is limited to `PARTY_MAX_SIZE` adventurers (by default set to 2).
- When you move into a dangerous room, you must choose which of your adventurers will fight the monster (in the `MoveAction` class, this is represented by the variable `index_` which indicates that the adventurer at this index in your party is moving in). The probability that your adventurer will die is a function of the strength and magic abilities of both of them, as described in the function `probability_of_dying`. If your adventurer dies, you do not get to enter the dangerous room.
- When you enter a chest room, you automatically collect the reward.

A map is given in the file, and drawn in Figure 1.

In the implementation, the effect of each action is defined in the action classes. Your task is to implement the method `next_states` of the three classes `HireAction`, `MoveAction`, and `NoAction` (they all have to `TODO` flag). You can test your implementation for `HireAction`, `MoveAction`, and `NoAction` with files `test21.py`, `test22.py`, and `test23.py`, respectively.

Fell free to run the file `map.py` too, which generates a 20-long simulation of the optimal policy for this map. Interestingly, the first step of the optimal policy is to move to Room1 and then go to the Market. I can find two good reasons to go to Room1 first. **Can you find these reasons? Share these reasons on discord.**

If you play with the map or the problem definition and come up with counter-intuitive behaviours, please share them with me and your classmates via discord.