Markov Decision Processes

Advanced Topics on Artificial Intelligence

This assignment revolves around Markov Decision Processes.

It is comprised of three independent parts, worth 6 or 7 points each.

The repository contains a number of (most) pythons files that are extensions of the code that was given for Lab 1. The structure of the code base is similar to the one you saw during the lab. However, the methods have been implemented, and you can use them. **Important: do not replace the implementation provided here with your own implementation**. If you do replace the implementation, you risk being incorrectly flagged for plagiarism. Files have been added; they are described in the next pages of this document.

In order to complete the assignment, you need to fork the following repository:

All the code is in python. You do not need to install any external library.

Comment: Refrain from using the python method `deepcopy`, as it has been responsible for a number of bugs in the previous years.

1 Modelling (7pts)

This exercise will test your understanding of Markov Decision Processes and your problem modelling skills. It is based on the DungeonMDP that was introduced during the lab.

We assume that the DungeonMDP is a portion of a real video game. The company that designs this game wants to balance their different game dungeons. For this purpose, they introduce all sorts of constraints in their dungeons. They want to ensure that the expected outcome of the optimal policy remains within acceptable values (between 30 and 40) when using discount factor $\gamma = 0.95$. Currently, the "basic dungeon" (the DungeonMDP built by using the method `basic_map`) yields a value of about 62.2, well outside the acceptable range of $[30, 40]$.[1]

In this exercise, you are asked to write methods that take an MDP as input and return a modified version of this MDP. We saw a similar situation in the lectures when we needed to make the assumption that the reward function was always strictly positive: we created an MDP M' from an existing MDP M by "copying" this MDP and adding δ to each reward. During the lecture, the optimal policy for M' was the same policy as the one for M, and we had $V'_\pi(s) = V_\pi(s) + \frac{\delta}{1-\gamma}$ (where $V_\pi(s)$ is the value of state s when using policy π in M and $V'_\pi(s)$ the value of state s when using π in M'). In the DungeonMDP (or in a more advanced version), we could decide to forbid the use of certain adventurers, items, or powers, change their price, modify their effects, etc. As a consequence of this modification, the optimal policy for the modified MDP M' might be different from the optimal policy for M, and the value of the initial state (i.e., the value of the dungeon) will also be modified.

You will apply these methods to the basic dungeon. You will then use Value Iteration (provided in `algos.py`) to determine whether, with the parameters you used, the new MDP yields an acceptable value.

The file `modelling.py` contains four methods. The first method is already implemented in order to illustrate how you can solve the problems. You will use the γ and ε values specified in this file ($\gamma = 0.95$ and $\varepsilon = 0.001$).

1.0 `add_cost_to_action(mdp, cond, cost)`

This method returns an MDP that adds a specified cost to all actions that satisfy the specified condition.

This method is provided as an example to illustrate what you are expected to implement. It returns an object of type `AddCostToEachAction` which is a wrapper for the MDP given as input. This wrapper returns the same values as the input MDP except for the `next_states` method where the specified cost is subtracted from all rewards whenever the action satisfies the condition.

This method is tested in file `modelling-test0.py`. It is tested with condition `q0_action_condition` which evaluates to *true* for all action except `NoAction`. In other words, the purpose of this method is to introduce a cost to all "real" actions.

If you run the main function of `modelling-test0.py`, you will notice that adding a cost of 1.8 to each action reduces the expected discounted outcome down to about 35.9, which is within the range $[30, 40]$.

[1] This value was estimated using VI and threshold $\varepsilon = 0.001$. Use these γ and ε parameters in your experiments. They are given in file `modelling.py`.

Feel free to ask questions about the implementation of this method in the discord. We are assuming that you are familiar with this type of programming, but we will happily help you understand how it works.

1.1 `penalise_state_action(mdp, a_cond, s_cond, cost)`

This method returns a modified MDP that adds the specified cost whenever is performed an action that satisfies the first specified condition in a state that satisfies the second specified condition.

Find the value of the cost parameter so that the expected discounted outcome of the Basic Dungeon falls between 30 and 40 for the given parameters `q1_action_condition` and `q1_state_condition`. The method, together with these conditions, penalises moving to an inn with a non-empty party.

Implement this method and compute a correct value for `Q1_ANSWER`.

1.2 `forbid_actions_in_states(mdp, a_cond, s_cond)`

This method forbids to perform actions that satisfy the first specified condition in states that satisfy the second specified condition *except* for those states in which this would mean that no action is applicable.

Find the value of the optimal policy when applying `forbid_actions_in_states` with parameters `q2_action_condition1` and `q2_state_condition1`, as well as the parameters `q2_action_condition2` and `q2_state_condition2`, The first pair of conditions forbids to hire a wizard if there is no soldier in the party; the second pair forces to hire a companion whenever possible.

Implement this method and compute the two values of the initial state.

1.3 `limit_action_number(mdp, a_cond, limit)`

This method limits the number of times that actions satisfying the specified condition can be used (except in states where those are the only available actions).

Find the value of the limit parameter so that the expected outcome of the basic dungeon falls between 30 and 40 for the given parameter `q3_action_condition`. This condition is satisfied by all action that hire adventurers. The method therefore limits the total number of adventurers that you are allowed to hire in the dungeon.

Implement this method and compute a correct value for `Q3_ANSWER`.

2 Strongly Connected Components (7pts)

As seen in class, some MDPs contain subsets of states that one cannot escape from. This is illustrated in Figure 1[2]: the states 8 and 9 form a sink; the states 5–7 form a loop, and are able to reach 8 and 9; state 4 is an isolated state that leads to 5–7; the state 1–3 form a higher level loop.

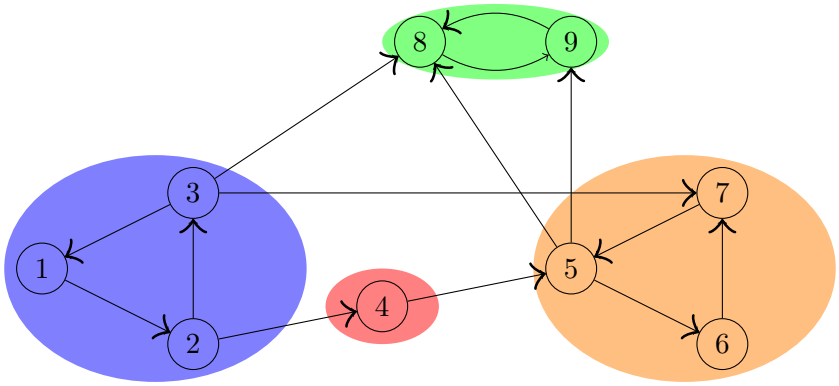

Figure 1: Example of a state transition system with four Strongly Connected Components.

A maximal set of states S' such that all states of S' are reachable from any other state of S' is called a **strongly connected component** (SCC). There are four SCCs in the example of Figure 1: $\{1, 2, 3\}$, $\{4\}$, $\{5, 6, 7\}$, and $\{8, 9\}$.

SCCs can be exploited to accelerate the procedure of computing a (optimal) policy. The idea is to run the algorithms (be it VI, PI, or something else) first in the inner SCCs (here $\{8, 9\}$) before considering the other SCCs. The reason is that the values of the states and actions in the outer SCCs depend on the values of the inner SCCs, and that it is generally not worth backing the inner SCCs' values too early.

In this section, you will have to use the implementation provided in `connectedcomp.py`. This implementation provides two classes:

- `ConnectedComponent` represents a Strongly Connected Component, defined as the set of states in this SCC as well as the SCCs that can be reached from these states by one transition. For instance, in the example of Figure 1, there would be four SCCs:
  - SCC_1 has set of states $\{8, 9\}$ and no children;
  - SCC_2 has set of states $\{5, 6, 7\}$ and one child, SCC_1;
  - SCC_3 has set of states $\{4\}$ and one child, SCC_2;
  - SCC_4 has set of states $\{1, 2, 3\}$ and children SCC_1, SCC_2, and SCC_3.
- `CCGraph` represents the graph of strongly connected components. The roots of the graph are those nodes that have no parent, SCC_4 in the example of Figure 1.

[2]Only the possible transitions are represented in the figure; actions are not.

2.1 Explain SCCs

File `decomposition.txt` contains a decomposition of the state space of the Dungeon domain (for the basic map) into SCCs. In the first part of the file, explain why there are several SCCs (instead of just one).

2.2 Topological-VI

Topological-VI is a variant of VI and a special case of Asynchronous VI that uses SCCs. As described above, Topological-VI performs Bellman backups on the inner SCCs (the SCC $\{8, 9\}$ in our example) before considering the outer SCCs.

In file `top.py`, implement the following method:

```
topological_vi(mdp: MDP, gamma: float, epsilon: float, graph: CCGraph)
```

Performs the Topological-VI algorithm on the specified mdp according to the specified CC-Graph, using the specified discount value, and stopping the algorithm when backups fall below the specify error threshold.

Notice that, because of implementation choices and because of the small size of the test domain, your implementation of Topological-VI may end up being slower than the implementation of VI[3].

You can use `top-1.py` to test your implementation.

2.3 Compute the Strongly Connected Components

There exist several algorithms to compute SCCs. The most efficient ones include *Tarjan's strongly connected components algorithm* and *Kosaraju's algorithm*. These algorithms are very clever. I propose that you implement a more expensive algorithm because it is easier to understand; you are allowed to implement instead one of the more complex algorithms mentioned above.[4]

In the algorithm I propose, you compute, for each state s, the set $Reach(s)$ of states that can be reached from s. The strongly connected components are subsets of states that share the same set of reachable states.

In file `connectedcomp.py`, implement the following method:

```
compute_connected_components(mdp: MDP)
```

Computes the CCGraph of the specified MDP.

You can use `top-2.py` to test your implementation.

[3] It is indeed the case for my implementation.

[4] You are free to look up their description in Wikipedia. Other sources should be explicitely mentioned.

3 Non-Deterministic Policies (6pts)

Given an MDP, there is often a single optimal policy. A policy indicates which specific actions ought to be performed in any given state; it is therefore very strict. In real applications however, there often are multiple nearly equivalent policies. It can therefore be useful to compute *non-deterministic policies*, in which the agent is given a *set* of actions that it may perform in any state.

Read the Section 2 of the paper from Fard and Pineau [FP08] that defines a non-deterministic policy Π (do not use the `pdf` from NIH which looks disgusting; the `pdf` from ResearchGate for instance is much better).

3.1 Explanation

Consider the policy presented in the MDP of Figure 2. For simplicity, it is assumed that, in this problem, each action is deterministic, i.e., it leads to only one state.

This policy is non-deterministic since (for instance) there are two possible actions from state 0.

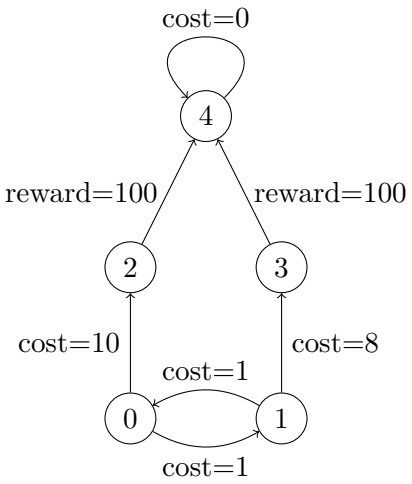

Figure 2: Example of non-deterministic policy for an MDP. Each action has deterministic effect.

We consider here a discount factor γ of .99 and a generous ϵ value of .5. Since the value of the optimal policy is 89.09 (starting from state 0), this means that the value of the non-deterministic policy should be at least ~ 44.5.

Because ϵ is so high, the non-deterministic policy allows the transition from 0 to 1 and from 1 to 0 because taking these transitions does not harm the total reward significantly (there is a cost of 1 and the effect of the discounting factor).

Yet, the policy in Figure 2 is not "conservative ϵ-optimal" according to the definition from Fard and Pineau. Explain why at the bottom of the file `nondet.py`.

3.2 Compute the Value of a Non-Deterministic Policy

In file `nondet.py`, implement the following method:

```
compute_policy_value(mdp: MDP, ndpol: NDPolicy, gamma: float,
                     epsilon: float, max_iteration: int)
```

Computes the value of the specified non-deterministic policy on the specified MDP with the specified discount factor. The algorithm should be iterative; Parameters `epsilon` and `max_iteration` are used to determine when to stop the computation.

You can use `nondet-1.py` to test your implementation.

3.3 Compute a Non-Augmentable Non-Deterministic Policy

Augmenting a policy means increasing the number of actions $\Pi(s)$, i.e., increasing the flexibility of the policy. When the policy is non-augmentable, it reached the point where increasing the flexibility will have negative consequences (e.g., it may significantly compromise the expected reward).

Exploit the property of monotonicity described in the paper by building a greedy algorithm that returns a non-augmentable non-deterministic policy that satisfies the property (6) from the paper:

$$V_M^{\Pi}(s) \geq (1-\epsilon)V_M^*(s) \quad \text{for all state } s.$$

In file `nondet.py`, implement the following method:

```
compute_non_augmentable_policy(mdp: MDP, gamma: float, epsilon: float,
                               subopt_epsilon: float, max_iteration: int)
                               -> NDPolicy
```

Computes a non-augmentable non-deterministic policy for the specified MDP with the specified discount factor. Parameters `epsilon` and `max_iteration` are used to compute the value of the non-deterministic policies; Parameter `subopt_epsilon` is the parameter that indicates how close to the optimal value the non-deterministic policy should be.

You can use `nondet-2.py` to test your implementation.

References

[FP08] M. M. Fard and J. Pineau. MDPs with non-deterministic policies. In *21st Advances in Neural Information Processing Systems (NeurIPS-08)*, pages 1065–1072, 2008.