Assignment 3: Markov Decision Processes

COMP4620/8620, Advanced Topics on Artificial Intelligence

Second Semester 2021

This assignment revolves around Markov Decision Processes.

It is composed of three independent parts, each worth 5 points. It is based on a solution to the Lab1.

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.

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

- `https://gitlab.cecs.anu.edu.au/comp4620/2021/comp4620-2021-s2-assignment1`

Make sure that your local copy is private.
Then follow the instructions in the next pages and edit the appropriate files. Commit and push your changes into your fork before the deadline:

Monday, October 18, 23.55 AEDT

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

1 Modelling (5/15)

In this exercise, you are asked to modify MDPs in order to incorporate additional constraints. These constraints can be used for instance in order to force the agent to modify their behaviour, for instance by penalising or encouraging some types of behaviours (e.g., forbid dangerous actions). This approach is generally better than directly editing the policy.

There are five methods to implement in the file `modelling.py`. Each problem is described below. In order to test your implementation, I also added the files `modelling-`i`.py` for i in $\{1, \ldots, 5\}$. These test files are based on the `Dungeon` example of Lab1. I also included `modelling-0.py`, which shows how the agent should behave optimally for the basic map, and I describe below how you should expect the agent to behave given the modification to the MDP.

By default, the agent hires peons from the start location, then tries to move to Room 1, Room 2, and the Market, in this order. From there, the agent hires a Wizard, move to the Large Chest Room, and finally go to the Small Chest Room.

Here are the five methods you are asked to implement:

1. `modify_action_reward(mdp: MDP, a: Action, delta: float)`

This method returns a modified version of `mdp` such that the execution of the action `a` modifies the reward by `delta` (so, if the execution of the action was supposed to provide a reward of `r`, it will instead give a reward of `r+delta`).

You can test it with file `modelling-1.py`. In this scenario, we give a huge subsidy to hiring peons. As a consequence, the optimal policy is hiring peons from the Start location and moving to the Large Chest Room in order to kill the peons; then return to the Start location to re-hire peons, etc.

2. `penalise_condition(mdp: MDP, condition: Callable[[State],bool], delta: float)`

This method returns a modified version of `mdp` such that every time a state is reached in which the specified condition is satisfied (i.e., such that `condition(state) is True`) the reward is reduced by `delta`.

You can test it with file `modelling-2.py`. In this scenario, there is a penalty for having a party of two. As a consequence, the optimal policy involves going to the Room 3 with the peon after Rooms 1 and 2 are visited, with the idea that losing him is actually better than keeping him around.

3. `forbid_action(mdp: MDP, act: Action)`

This method returns a modified version of `mdp` such that `act` is forbidden except in states where it is the only applicable action. (Notice that this can be abused by the agent who can put themselves in a situation where `act` is the only applicable action.)

You can test it with file `modelling-3.py` in which moving to Room 1 with the first adventurer of the party is forbidden. As a consequence, the optimal policy requires hiring two peons early instead of just one.

4. `forbid_two_actions(mdp: MDP, act: Action)`

This method returns a modified version of `mdp` such that `act` can be executed only once. Again, if `act` is the only action applicable, it remains allowed.

You can test it with file `modelling-4.py` in which moving to Room 1 with the first adventurer of the party is authorised only once. The big difference with the basic strategy appears if the peon fails to enter

Room 1 the first time: because the move to Room1 by the first adventurer is only allowed once, the agent is forced to go to the Market, hire a wizard, and hire a peon afterwards. In the basic strategy, hiring the wizard is postponed as much as possible.

5. `one_action_between(mdp: MDP, a: Action, b: Action)`

This method returns a modified version of `mdp` in which it is forbidden to perform action `a` twice without the action `b` in between. For instance, `accca` and `abbaa` are forbidden while `accbca` and `abbaba` are allowed.

You can test it with file `modelling-5.py` in which it is forbidden to hire two peons for a price of 10 without hiring a soldier (for a price of 100) in between. The optimal strategy for this scenario involves moving to the Market as early as possible, and hiring a soldier if necessary.

2 Strongly Connected Components (5/15)

As seen in class, some MDPs contains subsets of states that one cannot escape from. This is illustrated in Figure 1[1]: the states 7 and 8 form a sink; the states 4–6 form a loop, and are able to reach 7 and 8; the state 1–3 form a higher level loop.

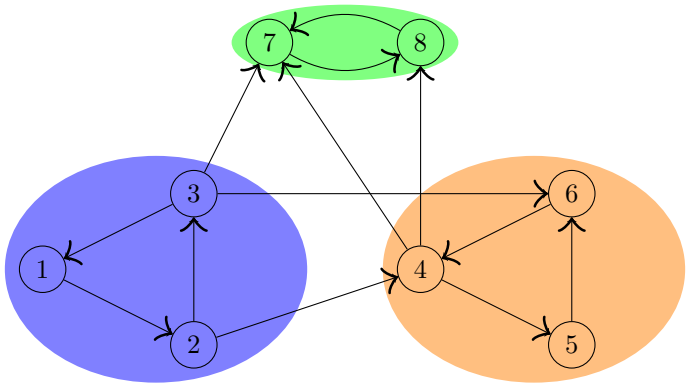

Figure 1: Example of a state transition system with three Strongly Connected Components.

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 three SCCs in the example of Figure 1: $\{1, 2, 3\}$, $\{4, 5, 6\}$, and $\{7, 8\}$.

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 $\{7, 8\}$) 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 these 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 three SCCs:
  - SCC_1 has set of states $\{7, 8\}$ and no children;
  - SCC_2 has set of states $\{4, 5, 6\}$ and one child, SCC_1;
  - SCC_3 has set of states $\{1, 2, 3\}$ and children SCC_1 and SCC_2.
- `CCGraph` represent the graph of strongly connected components. The roots of the graph are those nodes that have no parent, SCC_3 in the example of Figure 1.

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.

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

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 $\{7, 8\}$ in our example) before considering the outer SCCs.

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

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

Performs the Topological-VI algorithm on the specified mdp according to the specified CCGraph, 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 be slower than the implementation of VI[2].

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.[3]

In the algorithm I propose, you compute, for each state s, the set $R(s)$ 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:

7. `compute_connected_components(mdp: MDP)`

Computes the CCGraph of the specified MDP.

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

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

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

3 Computing Flexible Policies (5/15)

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.

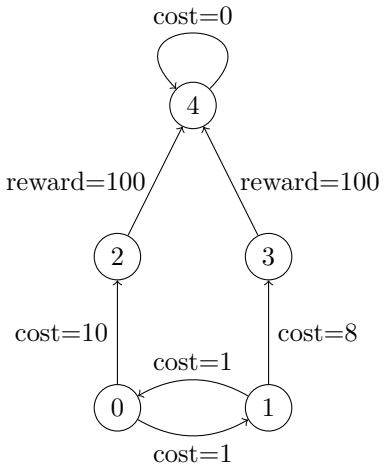

Figure 2: Example of non-deterministic policy for an MDP. Each action has deterministic effect.

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:

```
8. 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, then 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:

```
9. 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.

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.