Planning and Reinforcement Learning 2019

MDP Assignment Part 1

by Julio Lopez Gonzalez (2644792), Luisa Ebner (2638040), Anca Diana Vicol (2649551) and Burcu Kücükoglu (2638866)

Abstract

This report concerns the application of a number of fundamental planning methods to a small, finite MDP problem. The latter models the scenario of a single robot, starting in a fixed state and seeking to collect a treasure at the opposite side of a slippery ice word comprising cracks, which - if being stepped onto - terminate the robot's treasure hunt.
Section 1 provides a description of the environmental characteristics and explains decisive choices within the implementation process of the latter. Thereafter, six different planning algorithms are presented as being applied to the specified MDP. Most popular and basic among these methods are policy- and value iteration. Prioritized Sweeping and Simple Policy Iteration are implemented as modifications of the latter. For each of the aforementioned, a random policy, taking each action with equal probability was used as initial policy input to the planning algorithms. Furthermore, we present Linear Programming as a fundamentally different approach to finding the optimal policy. Due to problems in the implementation process, no final planning solution could be obtained therefor. Eventually, a comparative performance analysis shows that value iteration outperforms policy iteration in terms of elapsed run time and the number of iterations needed until convergence. Section 2 reflects about the scientific paper *Efficient Planning in MDPs by Small Backups* written by Harm van Seijen and Richard S. Sutton. Focus is put on the planning efficiency of small backups and the benefits, they provide particularly the context of model based RL. Eventually, a variant of interposing small backups to the traditional value iteration is presented in form of pseudo code, accompanied by a verbal explanation thereof.

1 Planning an optimal treasure hunt

1.1 The Environment

The RL problem underlying this report is designed as a finite Markov Decision Process (MDP). We are provided with complete information about the environments dynamics. As such, a full distribution model of the latter is available. In concrete terms, the assumed environment constitutes a discrete, 4x4 grid world, within which a single agent can move horizontally or vertically, presenting him with a set of four possible actions in each state. In terms of content, a robot may be imagined that seeks to reach a terminal treasure state (r = 100) from a fixed starting point, while preferably collecting an additional shipwreck treasure (r = 20) on its way. As a further hurdle, the environment comprises seven crack states. Besides a negative reward of -10, transitioning to these states terminates the agent's treasure hunt. State transitions are not fully deterministic due to the slippery nature of the environment. With a 5% probability the agent will transition to the edge of the grid or to the first crack in its path, reachable in accord with his selected action direction.
The environment was implemented using Python 3.7, particularly making use of the `Gym` library, which enables for tailored environment specifications on the basis of predefined skeletons. Accordingly, the following specifications were implemented:

1. the environment's shape (as a rectangular grid comprising 16 states)
2. the action set ({UP, DOWN, RIGHT, LEFT} encoded as {0, 1, 2, 3})
3. the crack, shipwreck, starting point and treasure coordinates
4. the rewards for every state
5. the initial state distribution (probability 1 to state (3, 0), 0 to all others)

Next up, the environments dynamics were computed in form of transition probabilities to potential successor states and stored as lists of tuples $[(P^a_{ss'}, s', r, \textit{terminal bool})]$. Depending on the possibility of sliding, transition probabilities are in {(0.05, 0.95), 1}. This being specified, the function `super` - taking the number of states, the number of actions, the transition probabilities and the initial state distribution as input parameters - creates our desired "icy grid".

1.2 Planning Methods

A random policy (RP)

We started off, implementing a random policy, which should later on be used as the initial policy, input to the algorithms of our planning methods. Concretely, we implemented $\pi_{random}(s)$ as a `numpy` array, assigning equal probability 0.25 to each of the four possible actions in each of the non-terminal states. Policy evaluation of π_{random} results in the state values shown in the Table 1, which portrays the 4x4 grid. As such, the value 5.02324 defines the state value of state (0,0).

	0	**1**	**2**	**3**
0	5.02324	10.028	36.91918	0.0
1	-2.65685	0.0	7.19236	0.0
2	-2.93093	-1.77392	-3.45944	0.0
3	-5.58178	0.0	0.0	0.0

Table 1: Policy Evaluation of RP with $\gamma = 0.9$

According to definition, all terminal state vaulues are 0. The increased distance to the goal state and risk of entering a crack state let state values in the bottom left corner of the environment be negative. Even in state (0, 2) - right in front of the goal treasure - the random policy reaches a state value of only ≈ 36.92.

Value Iteration (VI)

Implementing Value iteration with a discount factor $\gamma = 0.9$ outputted considerably higher state values, as shown in Table 2. The corrsponding, optimal policy shown in Figure 1 tells the agent to move up in the initial state, then walk right until collecting the shipwreck treasure and finally proceeding straight to the goal treasure.
Changing the discount factor γ to 0.6 systematically decreased all state values (except for state (0,2)). As the relative value sizes remained, the optimal policy remained the same.
Further decreasing the discount factor to 0.1 expectedly decreased the state values further. They are depicted in Table 4. In this case, also the optimal policy changed for state (1, 2). With such strong discounting of future rewards it has become preferable to enter the shipwreck treasure in one step than reaching for the goal treasure in 2 steps.

	0	**1**	**2**	**3**
0	82.3775	90.5	100.	0
1	74.13975	0	90.	0
2	74.85948625	88.13975	81.45	0
3	67.71184824	0	0	0

Table 2: Value Iteration $\gamma = 0.9$

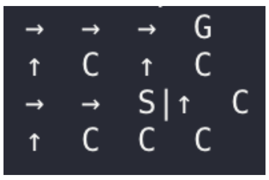

Figure 1: Optimal policy

	0	**1**	**2**	**3**
0	6.3775	14.5	100.0	0.0
1	0.63775	0.0	18.71642	0.0
2	1.27806	18.71642	2.27806	0.0
3	0.1533	0.0	0.0	0.0

Table 3: Value iteration $\gamma = 0.1$

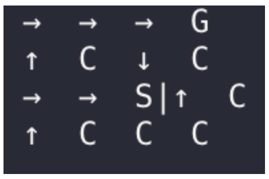

Figure 2: Optimal policy for $\gamma = 0.1$

Policy Iteration (PI)

Howard's policy iteration yielded the same values with value iteration for all three choices of γ. This was expected, since the major difference between PI and VI lies merely in the fact that VI uses a shortcut by not waiting for policy evaluation to converge for its calculation.

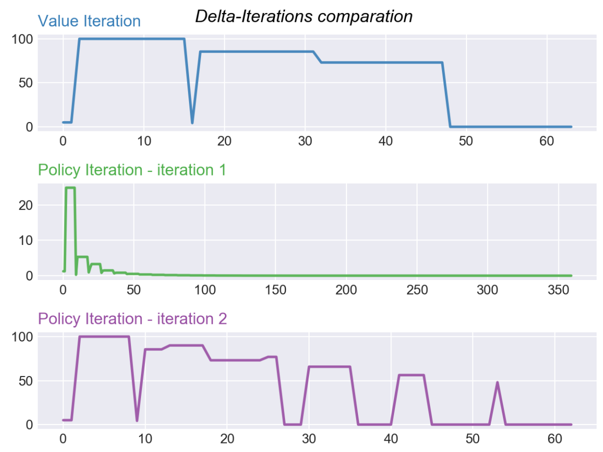

Figure 3: Comparation between value iteration and policy iteration

Figure 3 graphically revaels the differences between VI and PI. Whereas VI reaches an optimal policy in one iteration, PI required 2 iterations to converge. Whereas the x-axis indicates the number of iterations, the y-axis shows δ.According to their algorithmic implementation, convergence of $\Delta \leftarrow 0$ equals the convergence of the planning methods to an optimal, stable policy. As VI constantly regards the max over all actions in its value updates, it remains in plateaus most of the time before quickly proceeding to optimal state values. PI's updates to the contrary are more gradual. However, PI sweeps over the entire state set to compute updates. Convergence is reached only after ≈ 60 iterations in the second PI iteration.

Simple Policy Iteration (SPI)

Simple policy iteration conforms to Howard's traditional Policy iteration except for the fact that policy improvement is applied to only one instead of all improvable states. Given the value function for of a current policy π, the π is greedified only for one particular state s. As to that, Simple Policy Iteration spares sweeping through the entire state space before proceeding to an improved policy π'. [1]

Linear Programming (LP)

Linear programming was aimed to be presented as a fully different technique of planning an optimal policy. Due to the small problem size, the scope of the corresponding linear system was still trackable. The key concept for solving MDPs through linear programming entails maximizing the sum of values of all states under the constraint that the lower bound of optimal total cost of each state is the greatest possible. The optimal solution to this would give the optimal total cost function of the MDP. In view thereof, we created a corresponding set of linear equations and tried to solve it by use of the `pulp` library. Unfortunately the soving process did not terminate, presumably due to errors in the implementation process. As to that, no further analysis of this planning mathoed can be presented. [2]

Prioritized Sweeping (PS)

Prioritized Sweeping aims to combine high real time performance with high accuracy by storing all previous experiences to prioritize important dynamic programming sweeps and thus concentrating computational effort on the parts of the environment, that produce the greatest change in the state values. [3]

Comparative Performance Analysis

	elapsed time	**conv. its.**
VI $\gamma = 0.9$	0.00619	63
VI $\gamma = 0.6$	0.00483	64
VI $\gamma = 0.1$	0.00580	80
PI $\gamma = 0.9$	0.02333	423 - 2 iterations
PI $\gamma = 0.6$	0.01850	162 + 45 + 63
PI $\gamma = 0.1$	0.01460	54 + 72 + 72
SPI	0.12430	427
PS	1.31660	481

Answering the its primary conception, value iteration outperforms PI in terms of run time and the number of iterations needed to reach convergence. Especially the number of needed iterations is only a fraction of that under PI. We did not observe a trend regarding the different discount factors. However, these affect the values but not the computational performance of planning methods

2 Paper Review on Planning with Small Backups

a) Efficiency of small backups Within traditional planning methods in dynamic programming, the main planning operation consists in a full state (-action) value backup based on the current value estimates of all possible successor states s'. [1] The concomitant demand to loop over all possible s' lets the computation time of such backups be proportional to the number of successor states. In domains with a confined computation time frames, where the agent needs to quickly compute state (-action) values to determine and execute actions, these computationally expensive, full backups are often adverse, if not impracticable. In response thereto, [4] propose *small backups* as an efficient backup alternative for model based reinforcement learning. The core idea of a small backup is to iteratively regard only one particularly relevant successor value instead of all possible successor values in the update of $V(s)$. Traditional value iteration estimates a $V(s)$-update based on the sum of all possible successor values as part of the Bellman equation. [4] The small backup approach to the contrary, stores all precedent successor values v_i' that led to the current value $V(s)$ and updates the same only by the difference between one updated successor state V_i' and its old version v_i' as

$$V_i \leftarrow V_i + V_i' - v_i' \tag{1}$$

Put differently, this approach remembers and reuses the "old" estimate apart from one component that has changed. The latter change is added as the difference between the new and the old estimate.
Small backups are particularly planning efficient for more complex MDPs. The spared need to loop over all possible successors makes computation time independent of the number of successor states. The slightly increased memory demand to store the old estimates is no serious drawback in settings small enough to store the entire model. Furthermore, smaller updates make the planning method more incremental and this flexible and fine grained. The planning process can specifically be adjusted to the computation time available. Finally, small backups can be used jointly to create a "reversed full" backup, where the values of all *predecessor* states are updated through small backups. After every update, the Δ between a new and an old estimate is used to determine the value of its predecessor. This is a particularly interesting approach in the context of *Prioritized sweeping*. Knowing which state caused the greatest update and thus which state to prioritize for updates can spare (almost) redundant computations. [4]

b) Model based RL versus Planning Overall, there is a close relationship between planning and learning an optimal policy. However, both involve the incremental updating of value estimates in a long series of small backup operations. According to [1], the term *planning* is used for any "computational process that takes a model as input and produces or improves a policy for interacting with the modelled environment." [1] As to that, planning alone requires a complete MDP as a tuple (S, A, P, R, γ), inclusive of a full description of all possible state transitions, their probabilities and the accompanying rewards. Any dynamic programming method purely based on planning requires a suchlike *distribution model.* In practice however, *sample models* are much rather obtainable. In all these cases, the search for an optimal policy is referred to as *model based reinforcement learning (RL)*. The assumption of a MDP is upheld, but the transition probabilities $P_{sa}^{s'}$ and rewards R_{sa} of the stochastic environment are unknown. Accordingly, they must be replaced by estimates $\hat{P_{sa}^{s'}} = N_{sa}^{s'}/N_{sa}$ and $\hat{R_{sa}} = R_{sa}^{sum}/N_{sa}$, which need to be updated after each interaction with the environment. Only with a current set of estimates and thus a complete model estimate, actual planning backups can be performed. Model based RL requires acting, model-learning and planning to interact in a circular fashion, each producing what the other needs to improve. Consequently, planning may be referred to as the core step within the process of model based RL. [1]

[4] present small backups explicitly for model based RL and not only for planning for the following reasons.

First of all, planning alone suffices only for a very restricted set of MDP problems and can rarely be applied in practice. With an explicit reference to model-based RL, [4] expand the application scope and -potential and thus the practical relevance of small backups.

Secondly, the efficiency advantage of small backups is even more evident in model based RL, where the computation of state (-action) values depends on actual actions and interactions with the environment. This is not the case in pure planning. Consequently, the time to compute optimal policies is a less urgent issue in planning.

Finally, small backups can easily integrate the model estimation step at a computational cost of only $O(1)$ as

$$Q(s, a) \leftarrow [Q(s, a)(N_{s,a} - 1) + \gamma U_{s,a}(s')]/N_{s,a}$$

Due to the fact that only one successor value is included in a small backup,the full reward estimate $\hat{R_{sa}}$ never needs to be calculated explicitly. This spares additional computation time and memory.

c) Value Iteration using Small Backups

Algorithm: *Value Iteration with small backups*

```
Data:
Set $\theta$ as threshold for estimation accuracy
Initialize $V(s)$ arbitrarily $\forall s \in S$
Initialize $Q(s, a)$ arbitrarily $\forall s \in S$, $\forall a \in A$
Initialize $U_{sa}(s')$ arbitrarily $\forall s, s' \in S, \forall a \in A$
Initialize $\Delta U_{sa}(s')$ arbitrarily $\forall s, s' \in S, \forall a \in A$
$\Delta \leftarrow 0$
  Loop for each $s \in$ S:
      $v \leftarrow V(s)$
      Loop for each $a \in$ A:
          $\Delta U_{sa}^{s'} \leftarrow V(s') - U_{sa}(s')$
          $U_{sa}(s') \leftarrow V(s')$
          $Q(s, a) \leftarrow Q(s, a) + \gamma P_{sa}^{s'} \Delta U_{sa}^{s'}$
      $V(s) \leftarrow max_a Q(s, a)$
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$
  until $\Delta < \theta$
```

Value iteration normally performs sweeps of backups through the state space S, until the value function has converged. [1] In the pseudo code above, we adapted the original planning technique so as to use small instead of full backups.

Initially - as value iteration only approximates the optimal policy π^* - θ is set as a small numbered threshold to determine π's estimation accuracy.
Next up, state value and state action value for each state s and action a is initialized arbitrarily.
Moreover, $U_{sa}(s')$ and $\Delta U_{sa}(s')$ need to be stored additionally in order to make use of small backups.
Finally, Δ is initialized as 0 and will later on be used as the exit criterion to the while loop, creating value updates.

In the loop, the actual sweep of backups is performed using the Bellman equation. Other than in the traditional approach, $\Delta U_{sa}(s')$ defines the exact effect of one performed, small backup. Thereafter, $U_{sa}(s')$ is updated in order to further represent the correct successor values $V(s')$. Then, at the core of the algorithm, the corresponding state-action value estimate is given a small backup, which uses only the current value of a single successor state in the form of (1). By definition, $V(s)$ is updated as the maximum over all a of the updated $Q(s, a)$ values. Finally, the yet maximal update size is stored in Δ. After no small backup for no state s causes a change $\geq \theta$ anymore, the value function has (approximately) converged and an deterministic policy $\pi(s) \approx \pi*$ as an optimal state-action-mapping can be derived.

References

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[2] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving markov decision problems. In *Proceedings of the 11th International Conference on Uncertainty in Artificial Intelligence*, pages 394–402, 1995.

[3] Andrew W Moore and Christopher G Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning*, 13(1):103–130, 1993.

[4] Harm Van Seijen and Richard S Sutton. Efficient planning in mdps by small backups. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28, 2013.