Report: Assignment 1

Erik Kuiper[1], Tommy Maaiveld[2], and Whitney Mok[3]

[1] {VU University Amsterdam; 2521056; e2.kuiper@student.vu.nl}
[2] {VU University Amsterdam; 2528586; t.m.maaiveld@student.vu.nl}
[3] {VU University Amsterdam; 2546975; w.mok@student.vu.nl}

Introduction This report presents the implementation of the environment and four Dynamic Programming algorithms specified in the assignment. All must-have algorithms were implemented, as well as Simple Policy Iteration, and interpretations that address Questions 3 and 4 are discussed in the Evaluation section. Question 6 is discussed in the final section of the report, where the pseudo-code for a small backup implementation of the Value Iteration algorithm is given along with an explanation.

1 Implementation

Environment The environment implements the game as a Markov Decision Process (MDP), which is defined by a finite set of states (S), a set of actions associated with these states (A), a transition probability for each action between two states ($\mathcal{P}^a_{ss'}$), and a reward function ($\mathcal{R}^a_{ss'}$). Constructing the environment required setting up a table of rewards representing the treasure, given a value of 20, and the goal state which had a value of 100. Cracks were given a value of -10. Both the goal state and the tiles with cracks are terminal states. The indices of this table serve as a representation of the possible states and action; transitioning into a state immediately yields the displayed reward. The transition function maps a probability of transitioning to one or more states for each state-action pair. An action from a border state that would take the agent off the grid leads to a transition back to the same state with a probability of 1.

Random Policy Evaluation Policy evaluation was implemented and applied with $\gamma = 0.9$ to a simple random policy that chooses each action from each state with equal probability. Subsequently, the policy is evaluated by iterating the Bellman equation to generate a table of state values.

Value Iteration To implement value iteration, each state in the value table is filled in with the value generated by taking the optimal action from that state, weighed by the transition probability for each goal state ($\mathcal{P}^a_{ss'}$). Once the values have converged below a set benchmark, an optimal deterministic policy is returned from the final optimal actions.

Howard's Policy Iteration Howard's Policy Iteration combines iterative policy evaluation and policy improvement in a loop until a stable policy is achieved. First a random policy is initialized, and state values are iteratively recomputed under the current policy until (near) convergence. Then, an optimal policy is formulated for the new state values. The algorithm then reinitializes the policy evaluation step to generate a new value table, which in turn is used to generate a new policy. This loop is repeated until the policy remains unchanged after iteration.

Simple Policy Iteration Simple policy iteration is similar to Howard's Policy Iteration, except that each state is only updated once in the improvement step. After evaluation, the algorithm updates the policy for one state, and reinitializes policy evaluation. The state is improved until the policy remains unchanged after evaluation, upon which the next state is improved in a similar fashion. In the script, the method for Howard's Policy Iteration is used, but improvement is terminated upon finding an improvable state.

2 Evaluation

The performances of the four Dynamic Programming algorithms that were applied to the given MDP were assessed based on computation time. Additionally, iterations until convergence are considered in the discussion of the results. The Value Iteration algorithm was tested for values of γ in a range $[0.85, 1)$, in increments of 0.005. In the interest of completeness, the other algorithms were also run for these values of γ, and their results have been included accordingly.

Figure 1 shows that the run time of each algorithm does not differ much for values of γ in the range $[0.85, 0.91]$, while the control algorithms (Value Iteration, Howard's and Simple Policy Iteration) require more time until convergence for high values of γ (≥ 0.92). Random Policy Evaluation does not suffer from this effect, as it does not update or maximize its policy at all. The increase proportional to γ is near linear, as a higher discount factor simply means longer series of subsequent states to evaluate as the algorithm evaluates all possible move sequences, including non-terminal ones. However, for the control algorithms, γ presumably becomes large enough at this threshold for the reward of continually visiting the treasure state to weigh more heavily than visiting the goal state and ending the game with a +100 point boost. Thus, the agent reasons that going back and forth between the treasure state from its two adjacent states until a slip into a crack ends the game is the best way to maximize score. When $\gamma \geq 0.95$, even for the state adjacent to the goal, the optimal action is to head towards the treasure. Because of this, the optimal policy generated by Value Iteration varies for different values of γ. The optimal policies for different values of γ are given in Table 2.

Upon comparing the algorithms' performances for $\gamma = 0.9$, it seems that Value Iteration is the most efficient, followed by Howard's Policy Iteration, Policy Evaluation, and Simple Policy Iteration, which is least efficient. It was expected that Random Policy Evaluation would be outperformed by Value Iteration and Howard's Policy Iteration, as the combination of policy evaluation and policy improvement control algorithms is known to accelerate convergence. In this problem, Howard's Policy Iteration has little added utility over Value Iteration, as policy iteration can safely be truncated after a single sweep, reducing algorithm run time. Simple Policy Evaluation is the least efficient, as it evaluates each state independently and computes full backups for each single policy update, which is computationally expensive. The amount of iterations, seen in Figure 1b, does increase for higher values of γ. This is because Howard's Policy Iteration and Simple Policy Iteration only iterate before a check shows that the optimal policy no longer changes. For Value Iteration, a threshold parameter θ is set for the minimum update to be made to the value table before the algorithm terminates. Larger values of γ signify a longer exponential decay of the value function - in other words, the algorithm evaluates more future states and weighs them more heavily. θ was set to 0.0001, although different settings can yield lower iteration counts.

For $\gamma = 0.9$, the values learned by Policy Evaluation are shown in Table 1a, whereas the three control algorithms learned the same state values as shown in Table 1b. An identical optimal policy was learned by all three control algorithms for $\gamma = 0.9$, not just Value Iteration. Optimal policies learned by the latter under the whole range of γs are shown in Table 2. The optimal policy was learned by the Value Iteration algorithm for not only $\gamma = 0.9$, but for all values of γ in $[0.85, 0.91]$.

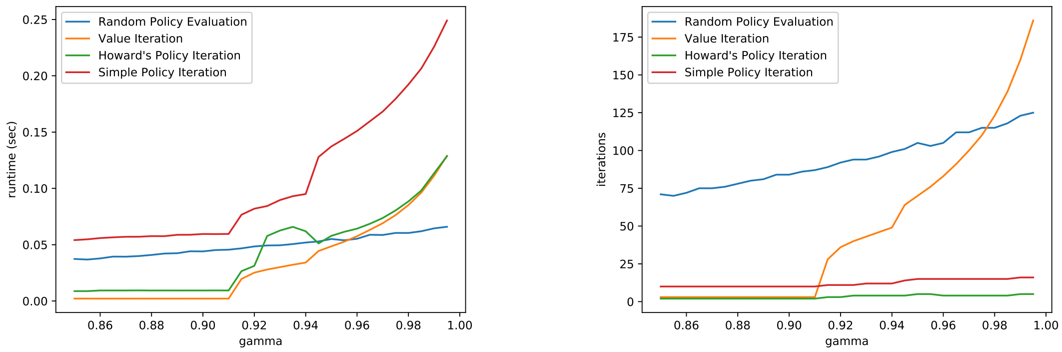
5.02	10.03	36.92	0
−2.66	0.	7.19	0
−2.93	−1.77	−3.46	0
−5.58	0	0	0

(a) Random Policy Evaluation

82.38	90.5	100	0
74.14	0	90	0
74.86	88.14	81.45	0
67.71	0	0	0

(b) Value Iteration, Howard's & Simple PI

Table 1: State value tables for various algorithms ($\gamma = 0.9$).



(a) Runtimes of the four algorithms as a function of γ

(b) Number of iterations until convergence for the algorithms as a function of γ

Fig. 1: Plots of the results for Random Policy Evaluation, Value Iteration, Howard's Policy Iteration and Simple Policy Iteration.

$\gamma = 0.85$	$\gamma = 0.86$	$\gamma = 0.87$	$\gamma = 0.88$	$\gamma = 0.89$
→ → → !	→ → → !	→ → → !	→ → → !	→ → → !
↑ × ↑ ×	↑ × ↑ ×	↑ × ↑ ×	↑ × ↑ ×	↑ × ↑ ×
→ → ↑ ×	→ → ↑ ×	→ → ↑ ×	→ → ↑ ×	→ → ↑ ×
↑ × × ×	↑ × × ×	↑ × × ×	↑ × × ×	↑ × × ×
$\gamma = 0.90$	**$\gamma = 0.91$**	**$\gamma = 0.92$**	**$\gamma = 0.93$**	**$\gamma = 0.94$**
→ → → !	→ → → !	→ → → !	→ → → !	→ → → !
↑ × ↑ ×	↑ × ↑ ×	↑ × ↓ ×	↓ × ↓ ×	↓ × ↓ ×
→ → ↑ ×	→ → ↑ ×	→ → ↑ ×	→ → ↑ ×	→ → ↑ ×
↑ × × ×	↑ × × ×	↑ × × ×	↑ × × ×	↑ × × ×
$\gamma = 0.95$	**$\gamma = 0.96$**	**$\gamma = 0.97$**	**$\gamma = 0.98$**	**$\gamma = 0.99$**
↓ → ↓ !	↓ → ↓ !	↓ → ↓ !	↓ → ↓ !	↓ ← ↓ !
↓ × ↓ ×	↓ × ↓ ×	↓ × ↓ ×	↓ × ↓ ×	↓ × ↓ ×
→ → ← ×	→ → ← ×	→ → ← ×	→ → ← ×	→ → ← ×
↑ × × ×	↑ × × ×	↑ × × ×	↑ × × ×	↑ × × ×

Table 2: Graphical representation of optimal policies obtained from applying Value Iteration with different values of γ. Note that the optimal policy is the same for $\gamma = [0.85, 0.91]$ and that this is identical to the optical policies found by Howard's and Simple Policy Iteration for $\gamma = 0.9$.

3 Efficient Planning on MDPs in Small Backups

3.1 Making planning more efficient

The downside of most planning methods is that they sweep through backups of the state-space or state-action space, until the policy or value function converges. In domains with a high action-selection frequency this can become computationally expensive, since a large amount of actions get checked on each iteration. Effective use of small backups will limit the amount of backups that have to be swept, which can help make planning more efficient.

3.2 Usefulness in a model-based learning context

Since this method samples on successor states rather than state-action pairs, the computational cost of using small backups does not scale based on the amount of transition probabilities. Additionally this practice means that this method of sampling does not introduce sampling variance. Because of this there is no need for step-size parameter tuning, which saves a considerable amount of time in model-based learning.

3.3 Small backups in value iteration

Algorithm 1 Value iteration

1: **while** $\Delta > \theta$ (a small positive number) **do**
2: $\Delta \leftarrow 0$
3: **for** each $s \in S$ **do**
4: $v \leftarrow V(s)$
5: $V(s) \leftarrow max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
6: $\Delta \leftarrow max(\Delta, |v - V(s)|)$
7: **end for**
8: **end while**
9: $\pi(s) = argmax_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$

Algorithm 2 Value iteration updated to include small backups

1: **while** $\Delta > \theta$ (a small positive number) **do**
2: $\Delta \leftarrow 0$
3: **for** each $s \in S$ **do**
4: $\Delta v \leftarrow V(s) - v$
5: $v \leftarrow V(s)$
6: $V(s) \leftarrow V(s) + max_{a'}[\gamma \mathcal{P}^{a'}_{ss'} \Delta v]$
7: $\Delta \leftarrow max(\Delta, |v - V(s)|)$
8: **end for**
9: **end while**
10: $\pi(s) = argmax_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^{a'}_{ss'} + \gamma V(s')]$

The differences between the standard value iteration algorithm and the small backups algorithm are fairly small. The differences are made to lines 4-5 in algorithm 1, which are lines 4-6 in algorithm 2. Instead of directly replacing v with $V(s)$ like the basic value iteration algorithm, the small backups version checks for differences between the stored value and the newly acquired value first, by creating a variable Δv which subtracts the old value v from the new value $V(s)$. This saves time in line 6, where a complete recalculation of the state-values is no longer needed. Instead of recalculating the value of all possible actions, we only need to calculate update the values that are related to the changes state-values we found earlier. We can use Δv to do this by looking at our a' rather than our a.