Data Intelligence Challenge Assignment 2

1. Introduction

Using cleaning robots saves time and energy of the person. The robots can clean the house and this automated process takes away the struggle of spending hours on cleaning manually. Using cleaning robots instead of manual cleaners can also be really sustainable, if the robot can clean the space efficiently.

With this project we aim to make an energy efficient cleaning agent that can clean a house. To achieve this we are using methods that are model-free and simulated (Monte Carlo Methods). Hence, in this assignment we do not require a model to obtain optimality, we use simulation and learn by interacting with the environment with the Monte Carlo Method (MC). On the other hand, Temporal Difference (TD) gives the maximum-likelihood estimate of the Markov process that generates the data. The MC method brings us efficiency, as the time required to estimate one state does not depend on the total number of states. The TD methods like Sarsa and Q-learning combine the aspects of Dynamic Programming and MC methods by using both bootstrap and sampling.

We are looking this problem at hand as a finite Markov Decision Process (MDP), which means that it can be represented by for 4 components, namely:

- A set of possible states. In our case, these are the cells of the grid.
- A set of possible actions. In our case, these are left, right, up and down.
- A transition probability function which tells us the probability of transitioning to a new state, given a state-action pair.
- A reward function which tells us the reward for performing a certain action and ending up at a certain state.

To solve this MDP we are utilizing value-based reinforcement learning techniques, in which we are trying to estimate the optimal value function that returns the maximum value possible under any policy. In the sections below, we will explain how we implemented the mentioned algorithms, and we will be comparing them in the means of efficiency and speed through the tests we ran.

2. Algorithms

2.1 Monte-Carlo

Monte Carlo Control (MC) is based on the idea of learning from repeated simulations. Simply, it starts with a random policy where every action for every state has the same probability, it runs several episodes (cleans a map until it dies or finishes), and at the end of each episode it updates the probabilities of the actions by looking at the rewards it gets from

doing them, and eventually finds an optimal policy. The main benefit of MC over Dynamic Programming methods is that it does not require an initial transition probability function, which means it does not require a perfect representation of the environment. This is a useful trait considering that most of the time, it is not possible to retrieve a perfect representation from a real life scenario. In Monte Carlo Control, exploration is quite important to be able to achieve an optimal policy, which means that the agent should be able to explore a lot of state-action pairs to be able to enhance its policy. Hence, to maximize exploration, we use two methods. Firstly, at each episode during training, the agent (cleaning robot) starts from a randomly selected dirty tile from the grid. Secondly, we use the ϵ-soft greedy approach, in which, while choosing an action to make during an episode, with probability $1-\epsilon$, agent makes the move that maximizes the value function, and with probability epsilon, it makes a random move. This method ensures that most of the possible state-action pairs are explored during training.

Monte Carlo Method tries to find the optimal action for all the states. This is obviously rather hard to achieve when the value of a state changes at each step (dirty tile becomes a clean tile, changing its reward) like in our problem. This makes the convergence to the optimal policy quite slow, and results in the agent performing sub-par. Another issue with the usage of the Monte Carlo method in our problem is that it cannot transfer its training between different grids, this means it has to be trained for every grid separately, which would take a lot of time. This makes the method mostly inapplicable to a real world cleaning robot.

2.2 SARSA

SARSA, also known as state-action-reward-state-action is a variation of Temporal Difference learning for MDP problems. This kind of algorithm is referred to as on-policy, which means that this is an algorithm that derives its value based on the current policy while incorporating some exploration steps. In each step of the algorithm, a matrix stores every possible state and every possible action from a state. Then, the scores for a certain state given a certain action are given by:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

where s_t and a_t refer to the current state and action, s_{t+1} and a_{t+1} refer to the next state and action and r_{t+1} refers to the current actions reward.

In order to ensure that the algorithm converges, a random exploration element is incorporated. This random exploration noise will ensure that a certain percentage of the time it will choose a random action. This makes it such that it will converge and explore a multitude of options ensuring a global minima result instead of a local one. SARSA is an algorithm that could be useful for real world applications. This algorithm is a near optimal policy technique, thus it does not produce the best results but avoids particularly poor areas, such as areas with many negative scores. With the extra exploration, certain decaying exploration functions are required, but are difficult to set up in practice.

2.3 Q-learning

The Q-learning algorithm is another variant of Temporal Difference learning for MDP problems. In contrast to SARSA, the Q-learning algorithm is referred to as off-policy. The only difference of Q-learning from SARSA algorithm is updating the current Q-value depending on the maximum Q-value available from the next state. The update rule is as follows:

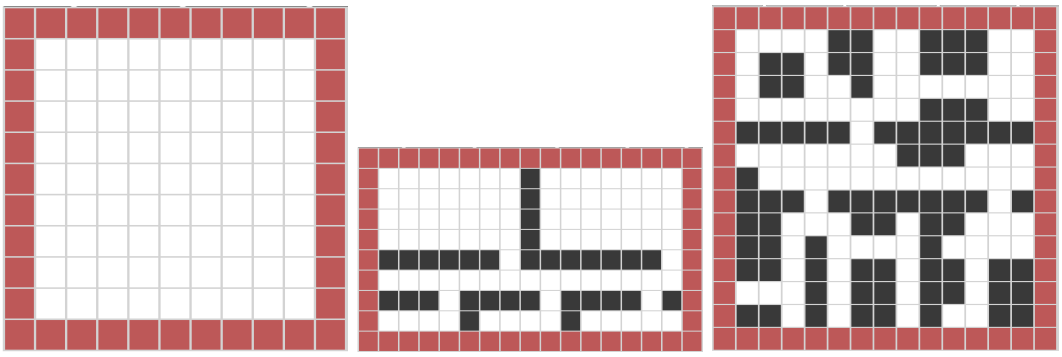
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma * max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

where s_t denotes the current state, a_t denotes the current action and r_t denotes the current reward.

In other words, the Q-learning algorithm considers best possible outcomes of future states at each time, with the usage of the *max* function. This ensures to learn an optimal policy, but being this "optimistic" about every next outcome brings some risks for the agent when exploring, depending on the environment. However in our implementation, since the agent has the whole grid access, it converges to optimal solution by selecting maximum Q-value actions without encountering environmental problems such as going to a death tile. The Q-learning algorithm can also be applicable to real world application scenarios, since it has been proven to converge to an optimal solution.

3. Experiments

In order to evaluate and compare the performance of these different algorithms, we tested their efficiency and speed on three different grids. These grids consisted of an *empty-room* (empty.grid, see figure 1a), *empty-house* (example-random-house0.grid, see figure 1b), and *furnished-house* (rooms-with-furniture.grid, see figure 1c) grid, designed to tests the algorithms' ability to adapt to increasingly more (irregular) obstacles. The three grids respectively have a total of 81, 89, and 90 unclean tiles to traverse.



(a) The empty-room grid (b) The empty-house grid (c) The furnished-house grid

Figure 1: The grids used for testing

Each test consists of several trials. For Q-learning and SARSA, a trial is defined as a time period of 25 seconds, in which the goal is to clean as much of the grid as possible. We defined

the trial this way in order to restrict the time it would take to test each algorithm. For Monte Carlo, a trial is defined differently. During the Monte Carlo trials, we implemented a battery drain of 2% which occurred with a probability of 0.1 with every move the robot makes. This battery drain is necessary because otherwise the robot runs the risk of getting stuck in an infinite loop, both during its training phase simulations as well as when it is deployed with its final policy. We also could not implement the 25 seconds time restriction, since it would be unclear whether the training phase would then need to fall under that restriction or not. We ran 1000 episodes during each training phase for Monte Carlo. Its performance may potentially improve with more episodes, but we could not run more due to the time restrictions. We also used the Greedy Random Robot as an extra baseline, since it is a simple example of an implementation which also does not have access to a model of the environment. Trials for the Greedy Random Robot are the same as those for SARSA and Q-learning.

3.1 Hyperparameter Search

Before running the main tests, we had to explore which hyperparameters had the best performance. For Q-learning and SARSA, we tested the hyperparameters α, γ, and ϵ. For Monte Carlo, we tested ϵ. For each hyperparameter we tested the values 0.2, 0.5, and 0.8. For each algorithm, We ran 5 trials per grid (15 in total) for each combination of hyperparameter values. The best-performing hyperparameter values are then defined as the combination of hyperparameter values which lead to the highest mean efficiency over all 15 trials. These values can be found in table 1.

Algorithm	α	γ	ϵ
Monte Carlo	-	-	0.5
Q-learning	0.8	0.5	0.5
SARSA	0.2	0.8	0.5

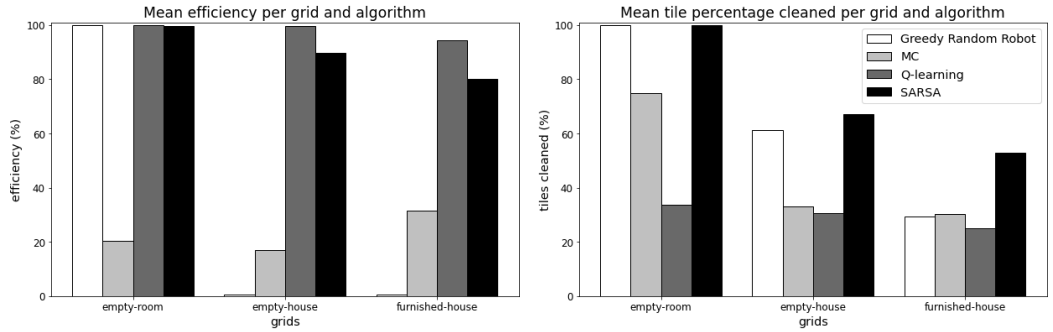
Table 1: Hyperparameter values with the best performance for each algorithm

3.2 Main Tests

Once the best performing hyperparameter values are known, we can run the main tests. For the main tests we ran 30 trials per algorithm per grid.

The main performance metrics we looked at were efficiency (a measure in percentage of how often the robot traversed clean tiles (potentially unnecessarily)) and cleanliness (the percentage of tiles the robot managed to clean within a trial). We also kept track of runtime (how many seconds each trial took), which is relevant in the Monte Carlo trials, as well as cases where the algorithm finished cleaning the room in less than 25 seconds.

Figure 2 shows the results of these tests. We can see that both SARSA and Q-learning have a consistently high efficiency, though it does drop slightly in grids with more obstacles. Q-learning's efficiency seems to be slightly higher than SARSA's, but SARSA's cleanliness is much higher in all grids. Since the cleanliness represents how much of the room these algorithms were able to clean in 25 seconds, this means that SARSA is much faster than Q-learning. SARSA seems to be the fastest algorithm overall, being the only algorithm to



(a) Mean efficiencies per grid per algorithm (b) Mean percentage of tiles cleaned per grid per algorithm

Figure 2: Results of the comparison between the current algorithms and the baseline Greedy Random Robot. The legend shown in figure (b) refers to both figures.

achieve 100% cleanliness in a trial, aside from the baseline Greedy Random Robot. However, it was only able to do this on the empty-room grid which contains no obstacles, with an average runtime of 22.9 seconds. Greedy Random Robot was able to do the same with a runtime of 0.01 seconds.

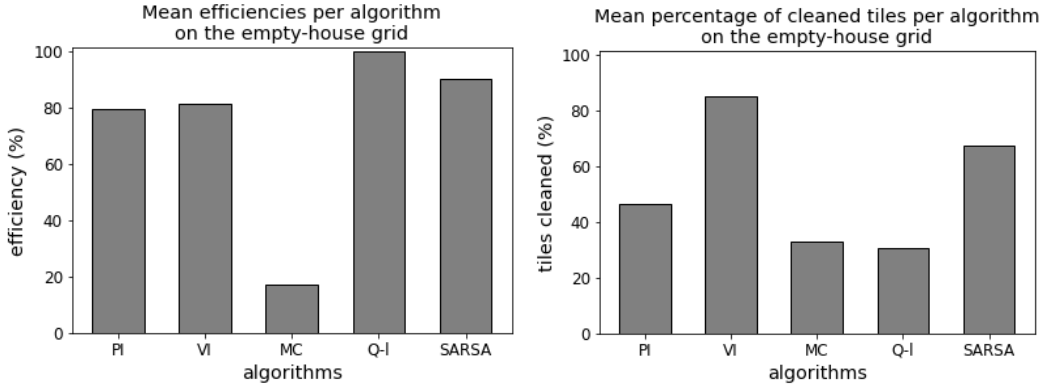
Monte Carlo has poor efficiency on all grids, but differs from the other algorithms in that its efficiency is highest for the grid containing the most obstacles (the furnished-house grid). This shows that perhaps its policy is still more random than is desirable, which might have been improved with more training episodes. Monte Carlo also has slightly better cleanliness scores than Q-learning. This is likely due to its higher speed, as it can make decisions very quickly once it has finished its training phase.

In terms of efficiency, each of the three algorithms beat Greedy Random Robot on all grids except the empty-room grid. This shows that for any space which is not one empty room, a more complex algorithm is necessary for good performance.

3.3 Comparison with Dynamic Programming methods

Since one of the grids we tested the current implementations on is the same as one of the grids we had previously tested our dynamic programming methods on, we were able to compare the results we got from our current tests with the results from assignment 1. The grid we used for this is the empty-house grid, and for both Policy and Value Iteration we only included the trials run with their best performing hyperparameters. For Policy Iteration this was a θ value of 0.2 and a γ value of 1.0, and for Value Iteration this was a θ value of 0.2 and a γ value of 0.5. For both Policy and Value Iteration a trial is defined the same way as for Q-learning and SARSA.

Figure 3 shows a visualization of this comparison. In terms of efficiency, both Policy and Value Iteration outperform Monte Carlo, but not Q-learning or SARSA. This shows that a complete model of the environment is not necessary in order to clean the grid efficiently. However, Value Iteration has the highest cleanliness score, which likely means that Value Iteration is faster than the other algorithms.



(a) Mean efficiency per algorithm on the empty-house grid

(b) Mean percentage of tiles cleaned per algorithm on the empty-house grid

Figure 3: Results of the comparison between previously implemented algorithms (PI: Policy Iteration, and VI: Value iteration), and currently implemented algorithms (MC: Monte Carlo, Q-l: Q-learning, and SARSA)

4. Conclusions

Our results have shown us that the model-free implementations Q-learning and SARSA have an improved efficiency when compared to implementations such as Policy and Value Iteration or a Greedy Random implementation. This is desirable, as a higher efficiency would lead to a lower use of power in the real world, which is more sustainable. However, in terms of speed Q-learning and SARSA are not necessarily improvements. This may also cause issues if we were to try to implement cleaning robots with these algorithms in the real world, as consumers may not be interested in a cleaning robot that cannot operate fast enough.

Our results also show that Monte Carlo is difficult to implement in this particular use-case. It may not provide adequate results in terms of efficiency, even though it is faster than Q-learning and SARSA once it has been trained.