EC503 Final Project

-Learning Policies for Markov Decision Processes From Data: a case study-

Project Report

Samad Amini

Vittorio Giammarino

Erfan Aasi

Boston University
College of Engineering

Contents

Chapter 1

Abstract

This project will address the problem of learning a policy for a Markov Decision Process. We consider a simplified scenario where a drone needs to deliver a package as quickly as possible. First, the drone must fly to a pick-up station to collect a package and then reach the delivery station to drop it off. The transition of the drone from a state to another is going to be stochastic, indeed wind and other elements will influence the drone's trajectory with certain probability. This shortest path stochastic control problem can be solved by means of a dynamic programming approach for small-dimensional spaces; however, for high-dimensional spaces this method suffers of the well known "curse of dimensionality". Thus, our main goal is to explore more practical alternatives to solve the drone's task and learn more efficiently an at least sub-optimal policy. We implement and use the SARSA algorithm, which is a well-known method in the framework of reinforcement learning, and an apprenticeship learning approach which, by using a regularized logistic regression, will try to learn a policy from expert demonstrations.

Chapter 2

Introduction

The shortest path problem is, in graph theory, the problem of finding path between two vertices (or nodes or states) in a graph such that the overall cost of the path is minimized [1]. Given a small-dimensional space, this can be solved by means of a dynamic programming (DP) approach where, starting from the terminal state, each edge is associated with certain cost and the cost-to-go is minimized. This kind of algorithms are quite old and suffer from the curse of dimensionality problem, i.e. the complexity increases exponentially with the dimension of the states. Therefore, despite such algorithms give guarantees on finding an optimal policy, often they are not practically implementable and heuristic methods or approximations are in general preferred. In this shortest path context, we are going to try to solve this problem by means of a well-known reinforcement learning (RL) algorithm, the State-action-reward-state-action (SARSA) [2], and by using an apprenticeship learning approach [3] based on maximum likelihood estimation. In particular for the latter, we will store data from the trajectories of an expert, which does not necessarily have to be optimal, and given these training samples, a regularized logistic regression is performed to infer the expert's policy; this procedure, as we are going to see, should make the learning process much faster.

2.1 Motivation

This project is a case study on the use of machine learning to learn a policy for a Markov Decision Process (MDP). The reasons why we decided to investigate this problem are the following:

- In first instance, both RL and apprenticeship learning are quite hot topics of research that were briefly mentioned in class. We believe that starting to explore these algorithms will turn to be useful not only for the sake of this course but also for future possible research directions.
- In second instance, despite our case study is only a simplified version of what happens in the real world, we believe that it represents quite well the main challenges that a designer faces from the implementation point of view, and at the same time, it emphasizes the pros and cons of these algorithms. Our final purpose is to achieve some sort of trade-off between implementability and performance.

2.2 Formulation of the problem

We now state in details the stochastic shortest path problem that we want to solve. First of all, we generate a random world and discretize it (see Fig. 2.1). The drone operates in a discretized $N \times M$ map (M is south to north and N is west to east). The yellow cell represents the pick-up station, the blue the delivery station and the red the drone base station. Moreover, the green cells are trees, i.e. obstacles the drone has to avoid and the cyan cells are "shooters" stations. Once too close to the shooters, the drone can be shot down with certain probability; an optimal policy has to stay away from them.
Additionally, the state of the drone at time k is described by $x_k = (m, n, \psi)$, where $m \in \{1, \dots, M\}$

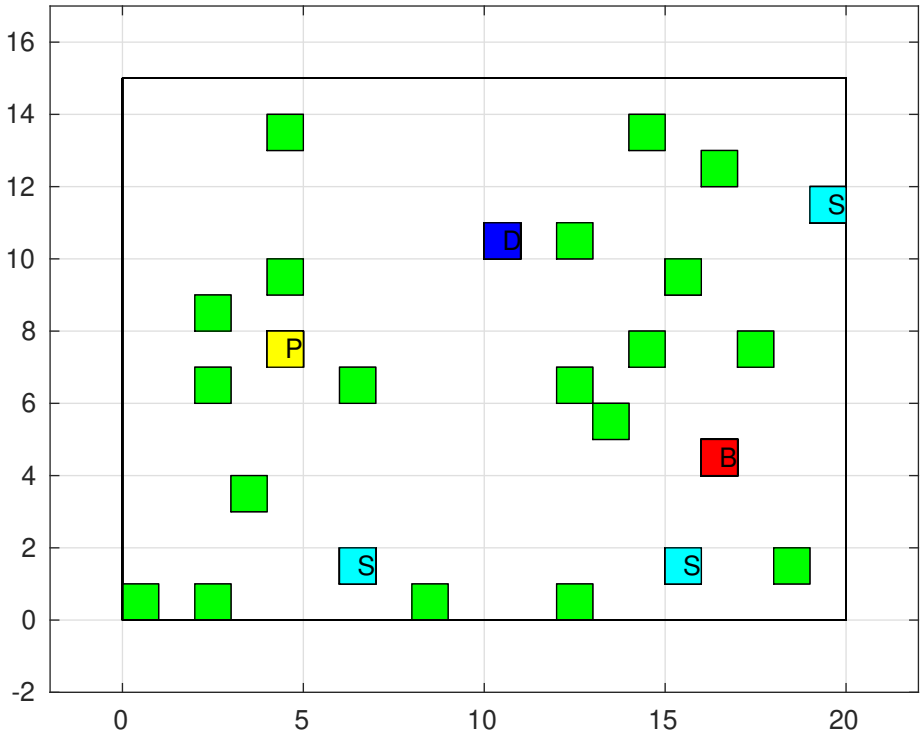


Figure 2.1: Top-down view of an example world of size 20×15

and $n \in \{1, \dots, N\}$ describe the drone's position on the south-north axes and west-east respectively; moreover, $\psi \in \{0, 1\}$, where $\psi = 1$ indicates the drone is carrying the package while $\psi = 0$ means there is no package. At each time step the drone is able to move north, south, east and west by one cell or stay where it is (hover). When the drone crashes on an obstacle (wall and trees) or it is shot down, it loses the package (if it was carrying it) and pays some additional cost to restart from the base cell. Finally, the behavior of the drone can be conditioned by some disturbances, which adds stochastisticity to the drone dynamics.
The position of the drone is randomly initialized and once picked the package up, it has to reach the drop off station without crashing.

2.3 Outline

The outline of the report is the following:

- Chapter 3 introduces the algorithms that we are going to explore to solve the stochastic shortest path problem. In particular, we start with a policy iteration in the paradigm of DP. We know that DP cannot be considered properly a machine learning algorithm; indeed, we are only going to use it to generate an example of optimal policy we aspire to achieve. Moreover, from the DP policy we sample the trajectories that are used for the apprenticeship learning algorithm. Afterwards, the SARSA [2] and the apprenticeship learning in [3] are briefly explained.

- Chapter 4, instead, focuses on the practical implementation of the aforementioned algorithms on our specific learning problem. The results are stated and the pros and cons of each algorithm are addressed.

Chapter 3

Methods

3.1 Dynamic Programming

Dynamic Programming (DP) is a method for solving a complex problem in a recursive step-by-step manner known as backward induction. The problem is broken down into a collection of simpler subproblems, which are individually solved, and their solutions are stored using a memory-based data structure. The final optimal policy is eventually obtained by combining the subproblems optimal solutions (principle of optimality [4]).

DP can be used to solve the stochastic shortest path problem when the structure of the MDP is known (i.e the transition structure, reward structure etc.). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. We can consider DP as an essential foundation for other approaches used in Reinforcement learning which, in fact, can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

We usually assume that the environment is a finite MDP. Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. Discretization, however, has been utilized to derive a finite MDP resulting in finding an approximation of the original problem. In particular, DP can be used for the planning in a MDP either to solve:

- Policy evaluation: Given a MDP (S, A, P, R, γ) where the finite state space, the action set, the state transition probability function, the reward function, and the discount rate are denoted by S, A, P, R, and γ respectively, and the policy π, find the value function V_π (which specifies the reward in each state, under the policy π). i.e the goal is to find out how good a policy π is.

- Optimal policy: Given a MDP (S, A, P, R, γ), find the optimal value function V_π and the optimal policy π^*. In other words, the goal is to find the policy which gives the most reward, with the best actions to choose.

Policy iteration is one of the methods by which one can compute the optimal policy. Policy evaluation and Policy improvement are the ingredients of Policy iteration.

3.1.1 Policy evaluation:

In this method we consider how to compute the state-value function V_π for an arbitrary policy π. Given that the environment is known, the state-value function is equal to expected value of the

reward incurred. Using Bellman equation it can be calculated as follow

$$\begin{aligned} V_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma V_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_k(s') \right] \end{aligned} \tag{3.1}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π. Also, the sequence V_k can be shown in general to converge to V_π as $k \to \infty$ under the same conditions that guarantee the existence of V_π. This algorithm is called iterative policy evaluation.

3.1.2 Policy improvement

Given a policy and its value function, we can easily evaluate the change in the policy at a single state to a particular action. It is a natural extension to consider changes at all states and to all possible actions, selecting at each state the action that appears best according to $q_\pi(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_\pi(s') \right]$. Therefore, to consider the new policy π' which is supposed to be better that the previous policy, the following can be seen as policy improvement formulation.

$$\begin{aligned} \pi'(s) &= \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \arg\max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_\pi(s') \right] \end{aligned} \tag{3.2}$$

3.1.3 Policy iteration

Once a policy, π, has been improved using V_π to yield a better policy, π', we can then compute $V_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions by repeatedly doing policy evaluation and policy improvement. Each policy is guaranteed to be a strict improvement over the previous one. Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations. The pseudo algorithm of policy iteration can be seen in Algorithm 1 [2].

Algorithm 1 Policy iteration

Initialization: $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrary for all $s \in \mathcal{S}$
while $\Delta < \theta$ **do** ▷ Policy Evaluation
  for each $s \in \mathcal{S}$
  $v = V(s)$
  $V(s) = \sum_{s^{'},r} p(s^{'}, r|s, \pi(s)) \left[r + \gamma V(s^{'})\right]$
  $\Delta = \max(\Delta, |v - V(s)|)$
end while
$\pi_{stable} =$ true. ▷ Policy improvement
For each $s \in \mathcal{S}$
  $a = \pi(s)$
  $\pi(s) = \arg\max_{a} \sum_{s^{'},r} p(s^{'}, r|s, a) \left[r + \gamma V(s^{'})\right]$
  If $a \neq \pi(s)$
    $\pi_{stable} =$ false
  EndIf
if $\pi_{stable} =$ true **then**
  Stop and return π
end if
$\Delta = \theta/2$
Go to 2

3.2 SARSA

SARSA is an on-policy algorithm where, in the current state s, an action a is taken and the agent gets a reward r, and ends up in next state $s^{'}$ and takes action $a^{'}$ in $s^{'}$. Therefore, SARSA stands for state-action-reward-state-action in learning a Markov decision policy. Also, SARSA algorithm is a slight variation of the popular Q-Learning algorithm known as an off policy technique and uses the greedy approach to learn the Q-value. SARSA technique, on the other hand, is an on policy and uses the action performed by the current policy to learn the Q-value.

SARSA algorithm considers transitions from state–action pair to state–action pair, and learn the value of state–action pairs as apposed to dynamic programming in which optimal policy is derived using transition probability from state to state. A general form of SARSA algorithm is given in Algorithm 2 in which γ, ϵ, and α are discount factor, parameter of ϵ-greedy, and learning rate, respectively [2].

Algorithm 2 SARSA Algorithm

Initialization: $Q(s, a)$, $\forall s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ arbitrarily
Repeat for each episode:
  Initialize S
  Choose A from S using policy Q (e.g. ϵ-greedy)
  Repeat for each step of episode:
    Take action A and observe R and $S^{'}$
    Choose $A^{'}$ from $S^{'}$ using policy derived from Q (e.g. ϵ-greedy)
    $Q(S, A) = Q(S, A) + \alpha \left[R + \gamma Q(S^{'}, A^{'}) - Q(S, A)\right]$
    $S = S^{'}$, $A = A^{'}$
  until S is terminal

As we know that SARSA technique is an on policy algorithm meaning the algorithm needs to update value functions strictly on the basis of the experience gained from exploration. Since SARSA uses the Q value of the the next action actually chosen by the learning policy, the conditions for convergence highly depends on the learning policy. In particular, because SARSA learns the value

of its own actions, the Q values can converge to optimal solution in the limit only if the learning policy chooses actions optimally in the limit [5]. In comparison with Q learning, SARSA method requires an additional assumption, which is convergence of learning policy, in order to converge to optimality. However, the trade-off is the more reward gained by this algorithm. The latter can be illustrated in Figure 3.1

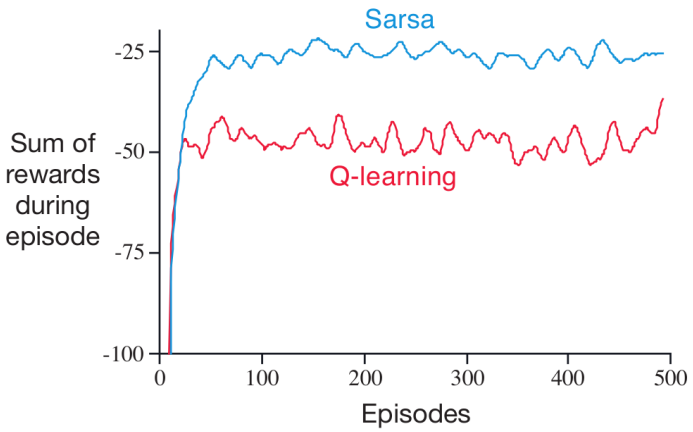


Figure 3.1: SARSA vs Q-learning for cliff walking task [2].

Also, there have been some work trying to simultaneously overcome the pros and cons of SARSA and Q learning by using backward Q-learning, which is successive to combine with the SARSA algorithm and the Q-learning [6].

3.3 Apprenticeship learning

In the MDP framework by knowing the reward function and state transition probabilities for all state-action pairs, the optimal policy for the MDP can be exactly determined, using dynamic programming techniques such as policy iteration. However, as we mentioned earlier, these methods suffer from the curse of dimensionality in large state-action space problems. Moreover, the reward functions are frequently difficult to specify; for instance, in learning from an experienced driver how to navigate autonomously. In such cases, the agent has no access to the exact reward function but has a set of state-action samples (also named as demonstrations). We assume these actions are generated by an expert player and based on a target policy that we desire to estimate.

The problem of learning from an expert is usually referred to as apprenticeship learning (or imitation learning, or learning form demonstrations). Due to the high-dimensional state-action space of such problems, the demonstrations usually contain samples from a small part of the state space and therefore a apprenticeship learning method must be able to select appropriate actions even at unobserved parts of the state space.

The problem of deriving the reward function using the demonstrations is called inverse reinforcement learning. In [3] by using both supervised learning and inverse reinforcement learning, they propose an algorithm for approximating the target policy, through mapping of the states' actions to a lower dimensional feature space and parameterizing a policy by a weight vector that represents the importance of the features. We have used this method in our implementations and will be explained briefly in below (for following the notation of [3], in this section we denote the state of the MDP by $x \in \mathcal{X}$).

3.3.1 Parameterize the Policy

We assume the expert is using a Randomized Stationary Policy (RSP), which is a mapping from the state-action pairs to a vector of features and is parameterized by a vector $\boldsymbol{\theta} \in \mathbb{R}^n$ that represents the

weights of the features. A set of RSPs is denoted as $\{\mu_{\boldsymbol{\theta}} : \boldsymbol{\theta} \in \mathbb{R}^n\}$, where for a given parameter $\boldsymbol{\theta}$, state $x \in \mathcal{X}$ and action $a \in \mathcal{A}$, $\mu_{\boldsymbol{\theta}}(a|x)$ denotes the probability of taking action a in state x. Boltzmann-type RSPs are used in here:

$$\mu_{\boldsymbol{\theta}}(a|x) = \frac{\exp\{\boldsymbol{\theta}'\boldsymbol{\phi}(x,a)\}}{\sum_{b\in\mathcal{A}} \exp\{\boldsymbol{\theta}'\boldsymbol{\phi}(x,b)\}} \tag{3.3}$$

where $\boldsymbol{\phi}$ is the feature vector associated with the state-action pairs and all the features are normalized between 0 and 1. It has been assumed that the policy and hence the feature vector are sparse, which means the number of nonzero elements of $\boldsymbol{\theta}$ is smaller than n and all of them are upper-bounded as $\forall i : |\theta_i| < K$.

We denote the state transition probability matrix, under the RSP $\boldsymbol{\theta}$ by $\mathbf{P}_{\boldsymbol{\theta}}$, where for each state pair (x, y), $P_{\boldsymbol{\theta}}(y|x) = \sum_{a\in\mathcal{A}} \mu_{\boldsymbol{\theta}}(a|x)P(y|x,a)$. Moreover by denoting the stationary probability of the Markov chains $\{X_k\}$ and $\{X_k, A_k\}$ by $\pi_{\boldsymbol{\theta}}(x)$ and $\eta_{\boldsymbol{\theta}}(x,a) = \pi_{\boldsymbol{\theta}}(x)\mu_{\boldsymbol{\theta}}(a|x)$ respectively, the average reward function corresponding to the RSP $\boldsymbol{\theta}$ can be formalized as $\bar{R}(\boldsymbol{\theta}) = \sum_{(a,x)} \eta_{\boldsymbol{\theta}}(x,a)R(x,a)$, where $R(x,a)$ denotes the one-step reward function.
Our goal is to learn the unknown target policy $\boldsymbol{\theta}^*$ by having $\mathcal{S}(\boldsymbol{\theta}^*) = \{(x_i, a_i) : i = 1, 2, ..., m\}$, which is a set of i.i.d. state-action samples generated by the policy $\boldsymbol{\theta}^*$ and according to the distribution $\mathcal{D} \sim \eta_{\boldsymbol{\theta}^*}(x,a)$.

3.3.2 Estimating the Policy

By having the set of i.i.d. state-action samples $\mathcal{S}(\boldsymbol{\theta}^*)$ and fitting a logistic regression function using the l_1-regularized maximum-likelihood estimation we have:

$$\max_{\boldsymbol{\theta}\in\mathbb{R}^n} \sum_{i=1}^{m} \log \mu_{\boldsymbol{\theta}}(a_i|x_i) \tag{3.4}$$

$$s.t.\ ||\boldsymbol{\theta}||_1 \leq B \tag{3.5}$$

where B is a tradeoff parameter that adjusts how well the algorithm fits the training data and how sparse is the obtained RSP to generalize well on the test data. The reason for applying the l_1-norm regularization is to induce sparse estimates and obtaining an RSP that only uses the relevant features, in addition to the fact that l_1-regularized logistic regression leads to a convex optimization problem. The procedure for obtaining the estimated RSP $\hat{\boldsymbol{\theta}}$, by adopting the logistic regression with an l_1-norm regularization is explained in the algorithm below.

Algorithm 3 Training Algorithm to Estimate the Target RSP θ^* From the Samples $\mathcal{S}(\theta^*)$

1: Initialization: Fix $0 < \gamma < 1$ and $C \geq rK$.
2: Split the training set $\mathcal{S}(\theta^*)$ into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ of size $(1-\gamma)m$ and γm respectively. $\mathcal{S}_1$ is used for training and $\mathcal{S}_2$ for cross-validation.
3: Training:
4: **for** $B = 0, 1, 2, 4, ..., C$ **do**
5: Solve the optimization problem for each B on the set S_1, and let $\boldsymbol{\theta}_B$ denote the optimal solution
end for
6: Validation: Among the $\boldsymbol{\theta}_B$'s from the training step, select the one with the lowest "hold-out" error on $\mathcal{S}_2$, i.e,
7: $\hat{B} = argmin_{B\in\{0,1,2,4,...,C\}} \hat{e}_{\mathcal{S}_2}(\boldsymbol{\theta}_B)$ and set $\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}_{\hat{B}}$

3.3.3 Selecting Features

Feature selection usually depends on the specific area of the problem and one may have intuition of the best type of functions that could be used as features. Here we we use kernel functions in a Reproducing Kernel Hilbert Space (RKHS) to obtain the desired features. To do so, by using

some kernel function $K(.,.)$ (such as linear or polynomial kernel) and a set of state-action pairs (x_i, a_i) that are encoded as $\mathbf{y}(x_i, a_i)\ \forall i = 1, ..., n$, we can formulate the feature function as $\phi_i(x, a) = K(\mathbf{y}(x_i, a_i), \mathbf{y}(x, a))$ and the feature vector as $\phi(x, a) = (\phi_1(x, a), ..., \phi_n(x, a))$.

3.3.4 Bounds on Regret

It can be shown that given $\epsilon > 0$ and $\delta > 0$, having enough samples $m = f(n, \epsilon, \delta)$ and estimating the target RSP $\boldsymbol{\theta}^*$ by $\hat{\boldsymbol{\theta}}$ based on the algorithm mentioned above, then with probability at least $1 - \delta$ we have

$$|\bar{R}(\boldsymbol{\theta}^*) - \bar{R}(\hat{\boldsymbol{\theta}})| \leq \sqrt{2\epsilon \log 2} R_{max}(1 + \kappa) \tag{3.6}$$

where

$$R_{max} = \max_{(x,a) \in \mathcal{X} \times \mathcal{A}} |R(x, a)| \tag{3.7}$$

and κ is the condition number that depends on the estimated RSP $\hat{\boldsymbol{\theta}}$.

Chapter 4

Results

In this chapter the algorithms we have introduced in the previous chapter are implemented and used to solve the stochastic shortest path problem stated in Chapter 2.

4.1 Dynamic programming

In the DP framework, we chose the policy iteration (PI) method without any particular reason, note that also linear programming or value iteration could have been used. The algorithm is easily implementable; however there are some major drawbacks:

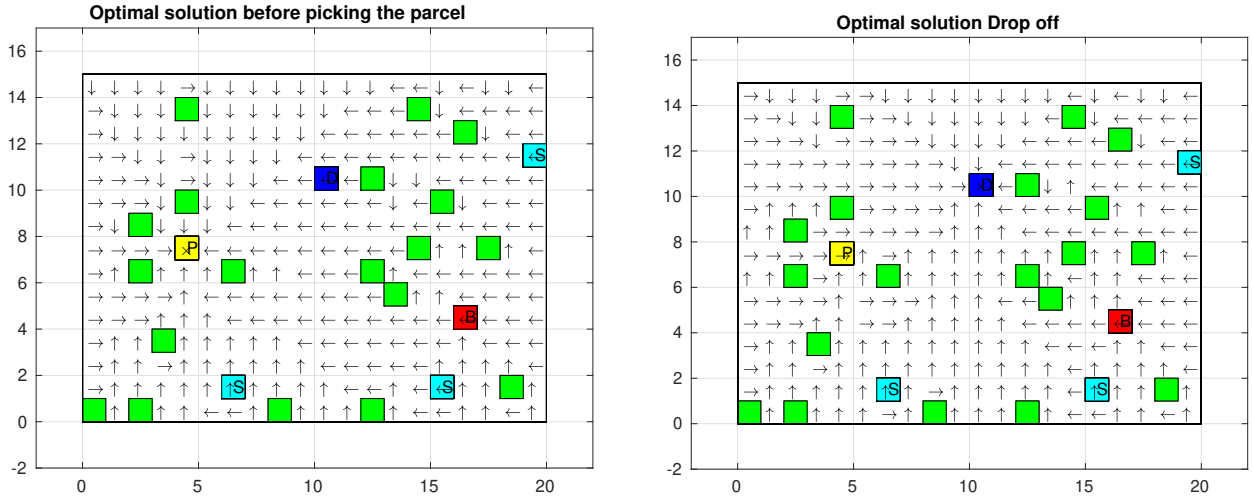
- First of all, the algorithm requires as input the transition probability P. In our problem, P is a tensor $|S| \times |S| \times |A|$ where for this specific simulation we have $|S| = 562$ number of states and $|A| = 5$ cardinality of the action space. This matrix represents the "model" of the drone's dynamics; clearly, such a model is not always available in real world problems and often it needs to be learnt from data. It comes naturally that the success of this algorithm strongly depends on how well we are able to estimate P.
- Similarly, we require a cost matrix G which contains the stage costs of all states in the state space for all control inputs ($|S| \times |A|$). Most of the times, it needs to be directly designed and for problems with many states and actions it is a huge amount of work. Also in this case, if there are mistakes in G the algorithm is not going to work effectively.
- Finally, the algorithm needs to compute the cost-to.go for each state-action pair which means we need to visit the entire dimension of P. Again, the size of the policy space increases clearly exponentially in the number of states.

Despite the shortcomings, this algorithm is able to find an optimal policy for any random starting point being therefore quite reliable. Note that, the policy is not unique but all the optimal policies will have the same optimal cost-to-go. Fig. 4.1 shows one of the optimal policy computed by the PI.

4.2 SARSA

The implementation of SARSA does not need to specify any transition matrix, we only needs to design a reward function based on the task we want to accomplish; hence, its implementation is much more straight forward with respect to the DP. The agent explores the map and for each action a_k in state s_k it receives a reward r_k according to which is going to be the next visited state s_{k+1}; this reward is then used to update the Q-function $Q(s_k, a_k)$. To make the algorithm work we need to specify an initialization for Q, note that a good initialization might remarkably speed up the learning process. Moreover, for the hyperparameters described in Chapter 3 we specify

$$\epsilon = 0.1, \qquad \gamma = 0.75, \qquad \alpha = 0.25, \tag{4.1}$$



(a) Policy before picking up the package.

(b) Policy after picking up the package.

Figure 4.1: Optimal policy of the PI algorithm. Note that, from any random starting point the agent is able to find its way to the goal. This policy is deterministic.

and we setup the number of episodes equal to $T = 10000$ with a maximum of 1000 steps per iteration. Fig. 4.2 depicts the cumulative reward over the number of episodes and Fig. 4.3 the final deterministic policy obtained with the algorithm.

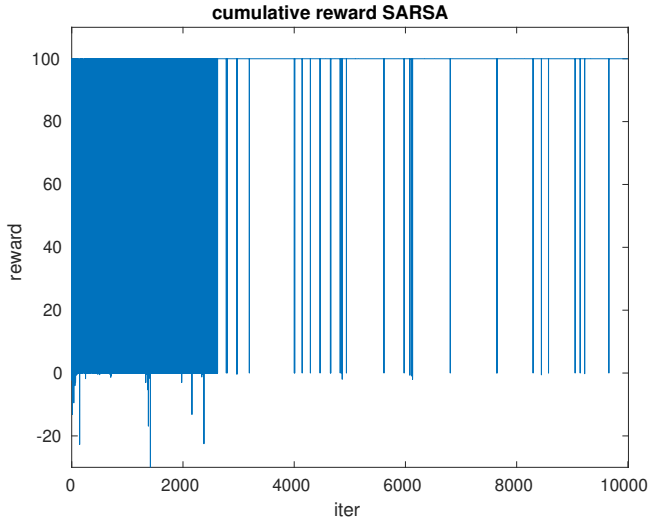
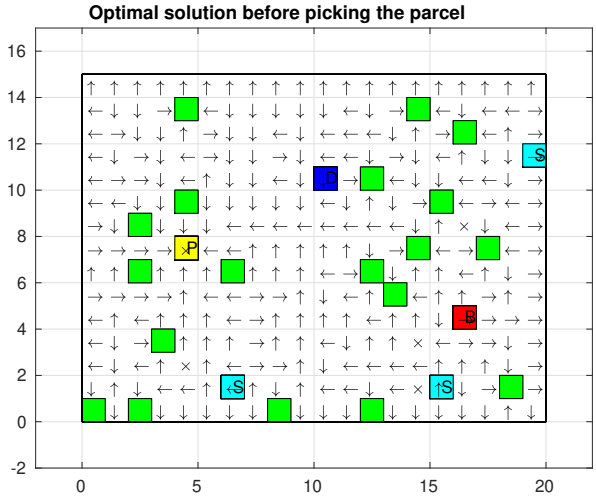


Figure 4.2: SARSA cumulative reward: it is evident how, in about the the first 3000 iterations, the algorithm fails and often the agent does not find its way to the goal. On the other hand, after the 4000th the agent, most of the times, is able to accumulate the maximal reward denoting that the learning succeeded.

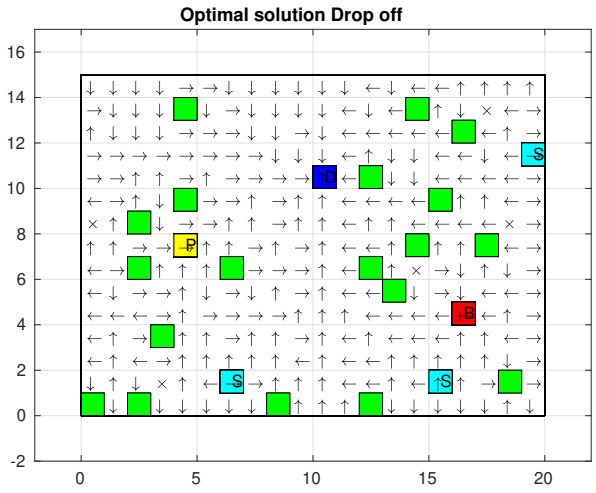
To summarize, the SARSA is much faster to be implemented with respect to the PI since it does not require any transition probability matrix but only a reward function. However, we have guarantees to converge to the optimal policy only for $T \to \infty$ and a satisfying policy is achieved only after $3000 - 4000$ iterations i.e. only when the agent has explored a sufficient number of states.

4.2.1 SARSA from expert

A simple way to speed up the learning process is by using a smarter initialization of the Q-function. For this purpose we sample from an expert, in our case the optimal policy obtained with the DP method (cf. Fig. 4.1), 100 random trajectories and use this data set to initialize Q by putting an higher weight on the state-action pairs observed from the expert. By means of this method, we



(a) Policy before picking up the package.



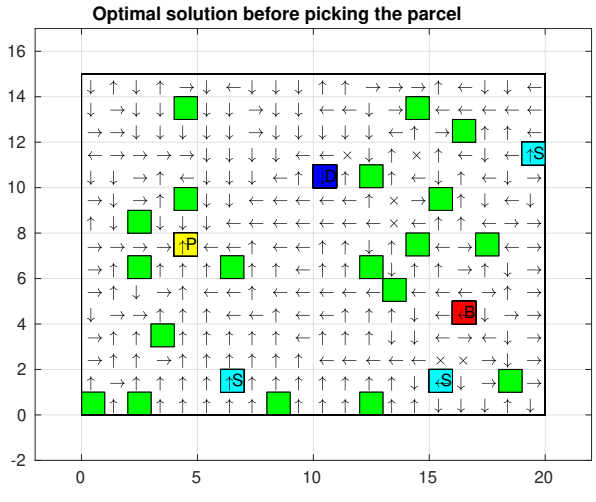
(b) Policy after picking up the package.

Figure 4.3: Optimal policy obtained with the SARSA algorithm. Note that, from certain starting points the agent fails. This policy is deterministic.

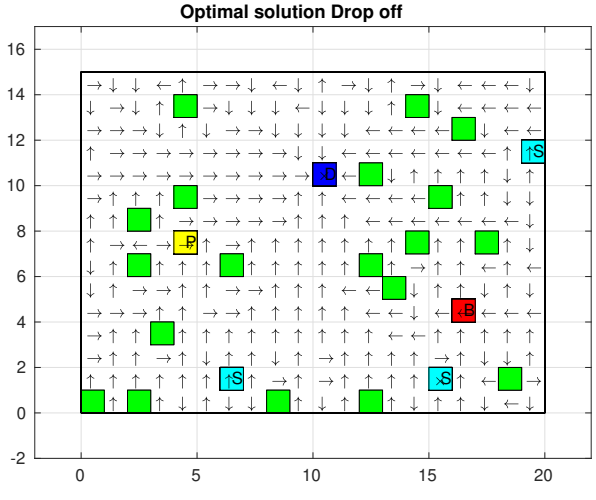
obtain the policy depicted in Fig. 4.4 with only $T = 1000$ episodes, 500 steps for each episode and the following hyperparameters

$$\epsilon = 0.01, \qquad \gamma = 0.75, \qquad \alpha = 0.25. \tag{4.2}$$

Thus, with remarkably less iterations the policy shows good performance, emphasizing the usefulness of initializing from experts data.



(a) Policy before picking up the package.



(b) Policy after picking up the package.

Figure 4.4: Optimal policy obtained with the SARSA algorithm with initialization from expert trajectories. Note that, from certain starting points the agent fails. This policy is deterministic.

4.3 Apprenticeship Learning

As presented in section 3.3, the algorithm we use for apprenticeship learning [3] parametrizes the policy as a Boltzmann-type random stationary policy; denoting therefore a stochastic policy rather than deterministic as we had so far. Moreover, the parameters θs of the policy are learnt through logistic regression based on a data set obtained from expert's demonstrations. As a matter of fact,

this algorithm does not need many specifications from the designer, however, the few are extremely crucial for learning. These are

- **Features functions** ($\phi(\cdot)$): The features are really important ingredients for the learning process and a bad designer choice might introduce a large approximation error in representing the target policy. Indeed, this is probably the most delicate part of the design which might determine whether the learning fails or not. Thus, as features topology we opt for a reproducing Kernel Hilbert space and in particular we select the Gaussian Kernel $K(\mathbf{x}, \mathbf{y}) = \exp\left(-c||\mathbf{x} - \mathbf{y}||_2^2\right)$ with $c = 10^{-4}$. We now denote $\phi(\mathbf{y}) = K(\mathbf{x}, \mathbf{y})$ where the first argument $\mathbf{x}$ is fixed. Our fixed points $\mathbf{x}$ are 134 waypoints on the map, where 2 of them are always the pick up station and the drop-off station; while the remaining are other cells on the grid evenly spaced. In summary, our policy is parametrized by 134 features and same number of weights θs.

- **Data Encoding**: The training data are not plugged into the features raw but each sample (x, a) is first encoded. In the algorithm we encode as follows $\mathbf{y}(\mathbf{x}, a) = \arg\max_{\mathbf{y}} P(\mathbf{y}|\mathbf{x}, a)$.

- **Training Data**: To perform the training we store 500 trajectories from the expert for about 9000 training samples. Of these data we use 70% for the training and 30% for the validation of the regularizer. The final choice is $\lambda = 4 \times 10^{-7}$ as illustrated in Fig. 4.5.

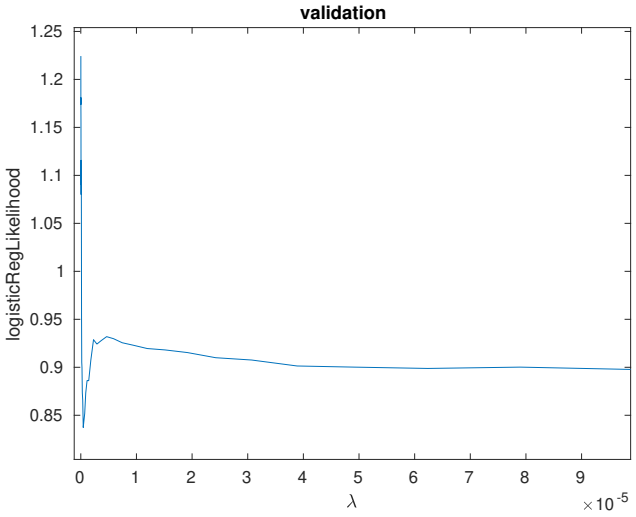
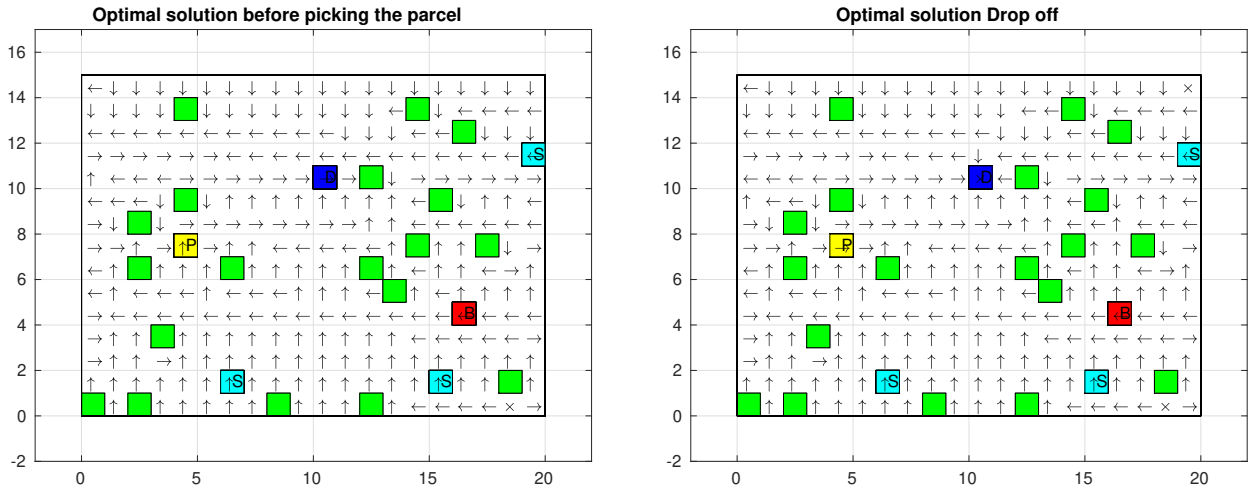


Figure 4.5: Validation for the logistic regression. We use 30% of the data set to determine the best regularizer λ.

The final policy obtained with this method is depicted in Fig. 4.6. Note that this policy is stochastic and the arrows in Fig. 4.6 are only considering the action with the highest probability.
In conclusion, the obtained results with this method are not as good as the previous algorithm but still decent. The agent rarely fails and the training is much faster with respect to the SARSA algorithm. The speed of the learning process is indeed the main pros of this algorithm.



(a) Policy before picking up the package.

(b) Policy after picking up the package.

Figure 4.6: Optimal policy obtained with the apprenticeship learning algorithm. Note that the policy is stochastic and arrows only represent the action with the highest probability not being representative of the actual behavior of the agent.

Chapter 5

Conclusion

In conclusion, throughout the project we tried to solve a stochastic shortest path problem by means of three different algorithms: Policy iteration, SARSA and apprenticeship learning. The first, when well designed, is the most reliable, albeit it shows many drawbacks, such as the exponential growth in complexity and the necessity of prior knowledge on the transition probability matrix, which negatively affect its use in real world applications. On the other hand, SARSA overcomes the issue on the prior knowledge but it remains quite slow, requiring many visits of the states before converging to an optimal policy. In terms of efficiency in learning the policy and amount of required prior knowledge, the apprenticeship learning is definitely a good trade off between SARSA and DP and represents a promising direction also to solve real world problems. Probably, more training data and a better features choice might have improved the obtained results which, despite decent, are not optimal.

Bibliography

[1] Wikipedia contributors. *Shortest path problem — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-April-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Shortest_path_problem&oldid=951399693.

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

[3] Manjesh Kumar Hanawal, Hao Liu, Henghui Zhu, and Ioannis Ch Paschalidis. "Learning policies for markov decision processes from data". In: *IEEE Transactions on Automatic Control* 64.6 (2018), pp. 2298–2309.

[4] Wikipedia contributors. *Bellman equation — Wikipedia, The Free Encyclopedia*. [Online; accessed 24-April-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Bellman_equation&oldid=952685800.

[5] Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. "Convergence results for single-step on-policy reinforcement-learning algorithms". In: *Machine learning* 38.3 (2000), pp. 287–308.

[6] Yin-Hao Wang, Tzuu-Hseng S Li, and Chih-Jui Lin. "Backward Q-learning: The combination of Sarsa algorithm and Q-learning". In: *Engineering Applications of Artificial Intelligence* 26.9 (2013), pp. 2184–2193.

Chapter 6

Code

Please find the code used for this project at `https://github.com/VittorioGiammarino/EC503_Learning_policy_for_MDP/tree/master`.