Model-Based Reinforcement Learning

by

Eirini Vandorou

University of Piraeus, Department of Digital Systems

Supervisor

Professor George Vouros

Contents

The Goal

The start of this thesis and initial purpose was to **combine two algorithms of reinforcement learning,** in the base of, the more information the agent learns about the environment, the better it will be in finding an optimum policy to reach its goal.

Towards this goal and following a model-based approach, the agent collects data and uses this data to learn how to solve the task. The two algorithms combined are the **Adaptive Dynamic Programming for learning** and the **Policy Iteration for computing and improving the solution based on what has been learnt.** However, since we need the agent to explore so as to learn the best policy, we introduced function that forces the agent to explore to an extent proportionate to the inverted exponential of the steps made so far. So, the combined idea is to, start with the ADP in order to explore the environment, and learn the model of the environment, provide that information to policy iteration in order to calculate the best policy, and so on, until both algorithms converge.

All this, in a base of learning new things on the field of reinforcement learning and see if this is a fit for my future goals. For the purpose of this thesis everything was made from scratch. No libraries or pre-implemented code for the main algorithms or their surroundings. However, libraries where used in order to create the report and the programs output.

A small introduction to the theory applied

Learning by getting signals from the environment

Imagine having to complete a puzzle, without being given all the pieces at once, and not knowing the picture of the puzzle, but being given a clap when placing a piece correctly and a bow down if you complete the puzzle. This is a problem that an *agent* might have to face in Reinforcement Leaning (RL). The agent is you in this case, being team spirits, we'll do it together. Each move we make is called an *action*, whereas, the puzzle is the *environment*. The instructor is the one that gives us a clap or bows down to the puzzle's correct completion. Now, how could we proceed?

Possible solutions

One way, is by **trying to get the most out of the information given** to us. Meaning, that after getting a piece we try to figure out a way to set the pieces we have in the correct place. As we don't know how the picture looks, we can easily end up failing. But that can't stop anyone from trying. In order to proceed we could, when given a piece, putting it down wherever it can be placed based on shape and color. In this case, the previous pieces matter to us because they provide valuable information on whether it would be a good move to place the next piece next to them and thus get a clap. We would ask questions like, do the colors match or is the shape correct? However, trying can be done in other ways as well. The process of using previous experience is called *exploitation* in the Reinforcement Learning field, while the process of trying more-or-less randomly is called *exploration*.

Thus, we may put pieces into place using a **brute force approach**. Trying to match pieces regardless of color or shape. After all, a puzzle can contain color transitions that we can't predict and pieces might seem or be alike, regards to shape. This is the exploration approach. Although, this is a plan, since we would at some point complete the puzzle, this would take a long time.

An improvement

The answer becomes easier when we **combine the two methods**. A better plan is to try in both ways, playing on the safe side and occasionally trying a random action. Using as much as we can of the information before placing a piece, together with trying new actions at least once in a while will cover all the possible cases. For example, if the colors differ a lot from piece to piece, via the exploration we may get over the problem in much less time than first trying to match colors. Information holds answers or hints to the answers, but we can't know whether we have all the information or just a part of it. Even so, if we do not have information (e.g. the color, the shape) the problem would be much harder.

The problem gets much harder by being given less or no information. Like if the pieces were upside down so we could not see the color. Information though does not always come only before an action is made, but also after. As stated in the first paragraph of this section, if we receive a clap it means we did well in our action. This is called a *reward* in RL, and it we receive it immediately after the action is made. Reward does show a lot about the action we made, but, what if we set all the pieces down and the puzzle is wrong. This means we will not get any hint from the instructor.

The bow shows greater respect, it is worth more than the claps, and it has a greater effect on the value of our actions. So, we won't stop until we get it and when we do we will re evaluate all the action made to determine what we did that lead us to the correct completion of the puzzle and thus the bow, in order to better know what to do if we were to do this all over again.

How is all this relevant?

Though the method described above is a pretty simple approach to solve a puzzle problem, we can identify some essential reinforcement learning concepts. The agent in this project faces more challenges, and stabs into more difficulties. Imagine someone saying "Don't use that piece yet" this would put an obstacle in our way, even temporarily (i.e. "… yet"). This section was an introduction to the theory needed, it is a good start, but there is a lot more to describe.

Theory Applied

Dynamic programming (DP) and reinforcement learning (RL) are algorithmic methods for solving problems in which actions (decisions) are applied to a system over an extended period of time, in order to achieve a desired goal. DP methods require a model of the system's behavior, whereas RL methods may not (Busoniu, Babuska, Schutter, & Ernst, 2010). In our case we are trying to solve a decision problem, whether we should choose one action over another and why.

Markov Decision Process - MDP

A Markov process is a stochastic process that satisfies the Markov property. This property states that the future behavior of the process given its path only depends on its current state. A Markov process whose state space is discrete is also called a Markov chain, whereas a discrete-time chain is called stationary or homogenous when the probability of going from one state to another in a single step is independent of time (Katja, Ann, Peter, & Maarten, 2008). What is more, an MDP relies on the notions of *state*, describing the current situation of the agent, *action* (or decision), affecting the dynamics of the process, and *reward*, observed for each transition between states (Garcia & Rachelson, 2010). An MDP can also be described as a controlled Markov chain, where the **control** is given at each step by the chosen action. Markov decision processes (MDPs) and their extensions provide an extremely general way to think about how we can act optimally under uncertainty (Kolter, 2016).

Markov decision processes are defined as controlled stochastic processes satisfying the Markov property and assigning reward values to state transitions. The environment model contains information about the environment. Information as such are the probabilities from a state, to the resulting state for a given action also called transitions. Formally, they are described by the 5-tuple (S, A, T, p, r) where: (Garcia & Rachelson, 2010)

- S is the state space in which the process' evolution takes place;
- A is the set of all possible actions which control the state dynamics;
- T is the set of time steps where decisions need to be made;
- $p(\,)$ denotes the state transition probability function;
- $r(\,)$ provides the reward function defined on state transitions.

About the above definition; for the entire set of possible states the symbol S is used, and for individual states the symbol s. For the model, the $Probability(s' \mid s, a)$ is the probability of the state s' as a result of the action a made in state s, which can be visualized as a set of transition triplets composing of the s, a and s'. Finally, for the reward, multiple definitions may be given since it may depend on the current state $R(s)$, the current state and action $R(s, a)$ or the current state, the action made and the next state $R(s, a, s')$. The rule according to which rewards are generated is described by the **reward function**. The process dynamics and the reward function, together with the set of possible states and the set of possible actions (respectively called state space and action space), constitute a Markov decision process (MDP) (Busoniu, Babuska, Schutter, & Ernst, 2010).

Sergey Levine in the lecture on Deep Reinforcement Learning (Levine, 2018), gives a nice explanation on what an **MDP** is, at its core, it **models an interaction between the agent and the environment** which consist of a loop (representing time), the agent makes decisions that are referred to as **actions**, and the world responses to these action with consequences. There are two types of consequences in RL, **observations** and **rewards**. Rewards are the immediate feedback the agent receives when making an action and moving to a next state. On the other hand, **utility** is a more accurate metric to the agent towards its goal. The utility of each state in this project is calculated using *Bellman equations*

Equation 1 Bellman equation

Let $\gamma < 1$. Using the symbols we defined earlier:

$$\forall\, s \in S, \qquad U(s) = \max_{a \in A}\left(r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s, a)\, U(s') \right)$$

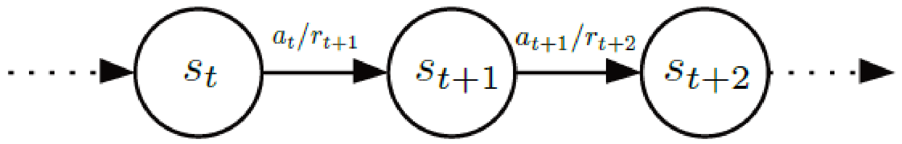


Figure 1 Illustration of the states and transitions of an MDP at different times. (Mohri, Rostamizadeh, & Talwalkar, 2018)

Explanation of Figure 1:

s_t: A state at time t

a_t: An action made in s_t

s_{t+1}: The state the agent is after making a_t

r_{t+1}: The reward given for making action a_t when in state s_t and reaching s_{t+1}

Deterministic and Stochastic MDPs

MDPs may fall in two categories, deterministic and stochastic. In the deterministic case, taking a given action in a given state always results in the same next state, while in the stochastic case, the next state is a random variable. In a **stochastic** MDP, the next state is not deterministically given by the current state and action. Instead, the next state is a random variable, and the current state and action give the probability density of this random variable. In this thesis, we consider a stochastic MDP.

Policy

The main problem for an agent in an MDP environment is to determine the action to take at each state. This (stochastic) mapping from states to actions is an action **policy** (Mohri, Rostamizadeh, & Talwalkar, 2018).

Optimal Policy

In DP and RL, the goal is to find an optimal policy that maximizes the utility for each state. Starting from a state $s \in S$, to maximize its reward, an agent naturally seeks a policy π with the largest value (Mohri, Rostamizadeh, & Talwalkar, 2018). As seen in Figure 2, the optimal policy might not consist of exactly one optimal action per state. This is because in some cases in the environment, the utility for some states may be equal if seen from a later position. In our case, the optimal policy presents one action for each state.

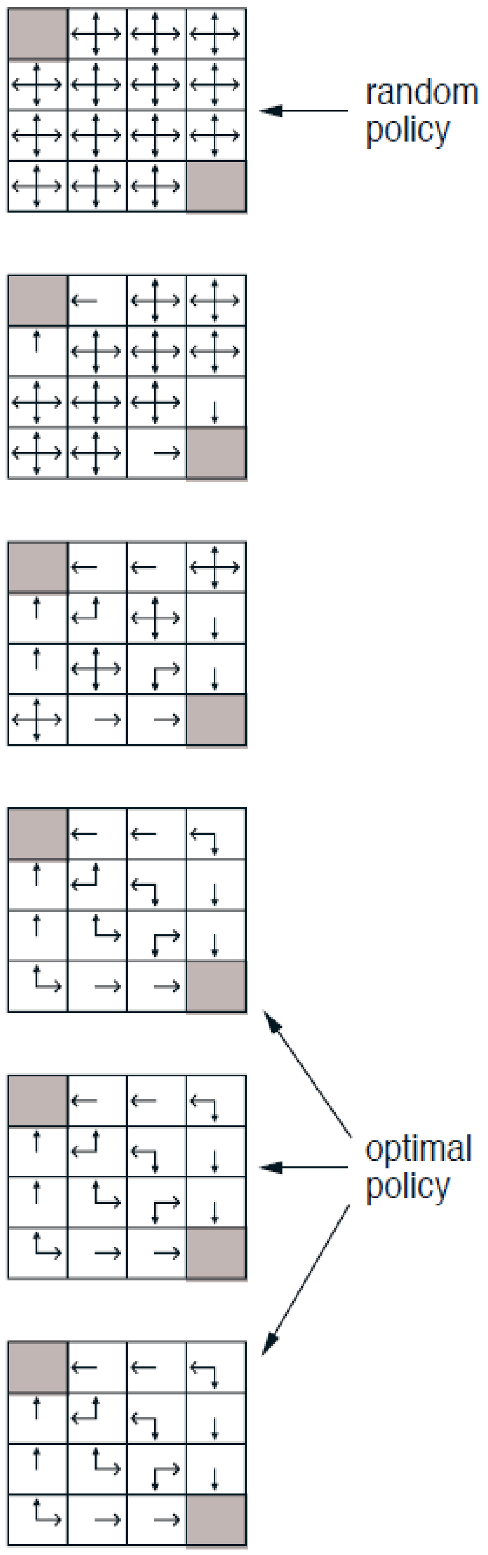


Figure 2 Convergence of iterative policy evaluation on a small gridworld The last policy is guaranteed only to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal. (Sutton & Barto, 2018)

Optimization algorithm for MDP

There are a lot of ways to find the optimum policy for an MDP, one of which is *Policy Iteration* algorithm.

Policy Iteration - PI

Policy iteration, in this case, is used to determine the best policy using a complete knowledge of the environment; obstacles and goal/end state.

Here is a short version of the algorithm of Policy Iteration:

```
1. Start with $\pi_0 \leftarrow Guess$
2. do
3.     **Evaluate** the given $\pi_t$ calculate $U_t = U^{\pi_t}$
4.     **Improve** $\pi_{t+1} = argmax_a \sum P(\, s' \mid s, a)\, U_t(s')$
5. while $\pi_{t+1}$ not equal to $\pi_t$
```

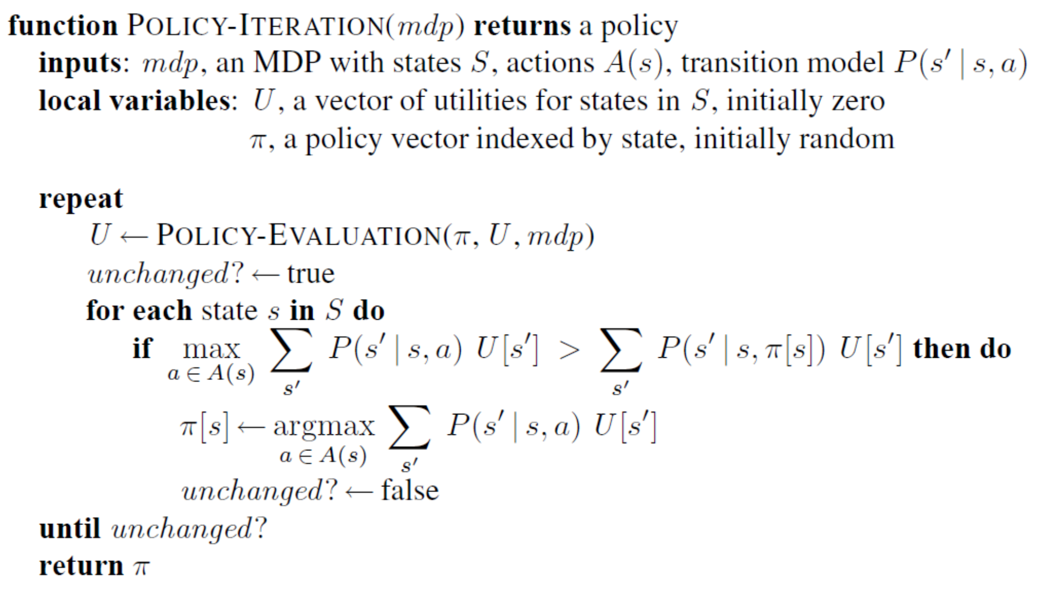
In step 1 it starts with some initial policy; called π_0. This is a guess, so it's going to be an arbitrary set of actions that the agent should take in different states. Then in step 3, it evaluates how good that policy is, and the way it evaluates it at time t is by calculating utilities with the help of Bellman equations. The utility in a time t (U_t), is equal to the utility that is got by following a policy (U^{π_t}). So, given a policy, it evaluates it by figuring out what the utility of that policy is. In step 4 the policy is updated, policy of time $t+1$ (π_{t+1}) , to be the policy that at each state it takes the action that maximizes the expected utility, i.e. the sum of possible next states' utilities (U(s')), w.r.t. the states' transition probabilities . Steps 3 and 4 are repeated until no change appears in the policy. Figure 3 shows the algorithm in detail, as specified in the book *Artificial Intelligence, A Modern Approach* (Russell & Norving., 2003).

Policy Evaluation

Having a policy π_i, we have to understand how this policy affects the states utilities. One way is to use a simplified version of bellman equations (Equation 1):

$$\forall\, s \in S, \qquad U_{i+1}(s) = r\big(s, \pi_i(s)\big) + \gamma \sum_{s' \epsilon\, S} p(s' \mid s, \pi_i(s)\,)U(s')$$

Since we want to evaluate the policy itself, we consider the action the policy indicates to be the best and calculate the utility of the state based on that action.



```
function POLICY-ITERATION($mdp$) returns a policy
  inputs: $mdp$, an MDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: $U$, a vector of utilities for states in $S$, initially zero
                   $\pi$, a policy vector indexed by state, initially random

  repeat
      $U \leftarrow$ POLICY-EVALUATION($\pi, U, mdp$)
      $unchanged? \leftarrow$ true
      for each state $s$ in $S$ do
          if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\, U[s']$ then do
              $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
              $unchanged? \leftarrow$ false
  until $unchanged?$
  return $\pi$
```

Figure 3 The policy iteration algorithm for calculating an optimal policy. (Russell & Norving., 2003)

Let us consider that in the initial state the agent does not have a complete knowledge of the environment: Neither its objects and end state, nor the model (i.e. the states transition probabilities). To determine the best policy, the agent has to learn these by exploring the world and getting feedback from the environment. In doing so, the agent has to figure out the MDP, or at least very closely approach it. One way of doing so is by gathering information using the Adaptive Dynamic Programming (ADP) algorithm run before PI. In order to explain ADP, we need to first introduce some concepts.

Machine Learning - ML

Reinforcement Learning (RL) evolves around the field of machine learning (ML). The learning methods can be classified according to the way they need to exploit data during the learning process: Therefore, in case they do need training data (i.e. examples from the desired mappings to be produced) to learn, they are classified as supervised methods, otherwise they are called unsupervised. RL is a special class of machine learning methods, where the agent learns by acting in the real world, i.e. it is neither a supervised nor an unsupervised learning method. In all algorithms, the input data is usually referred to as the data that is passed to the agent in order to produce the output data (whether annotated training data or samples of data); expected to be the appropriate in relation to the problem.

Reinforcement vs Supervised vs Unsupervised Learning

Reinforcement learning and Supervised Learning differ since the latter requires both input and output data, as well as accurate mappings between this, in order to properly learn from. Also, RL is different from what machine learning researchers call *unsupervised learning (UL)*, which is typically about finding structure hidden in collections of unlabeled input data. Therefore, Supervised and Unsupervised learning differ as well, with UL not needing examples of correct input and output mappings to learn.

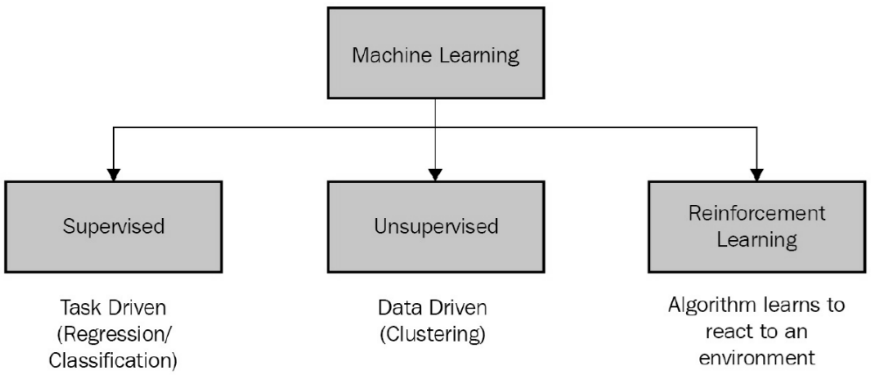


Figure 4 Types of machine learning (Farrukh, 2017)

Reinforcement Learning - RL

As Sutton and Barto describe in the first chapter of their book (Sutton & Barto, 2018), Reinforcement Learning is learning what to do -how to map situations to actions- so as to maximize a numerical reward signal.

Reinforcement learning is the study of planning and learning in a scenario where a learner agent actively interacts with the environment to achieve a certain goal. This active interaction justifies the terminology of *agent* used to refer to the learner. The achievement of the agent's goal is typically measured by the reward it receives from the environment and which it seeks to maximize (Mohri, Rostamizadeh, & Talwalkar, 2018). This type of interaction can be seen in Figure 5. Information as such is the probability to “land” to a state when performing an action at a state. These are also called transition probabilities. It is important to note that not all models use probabilities to denote the possible transitions. In deterministic environments there is only one possible outcome having made a certain action at a state, thus the probability of reaching a state from another is equal to 1 for each action that can be applied,

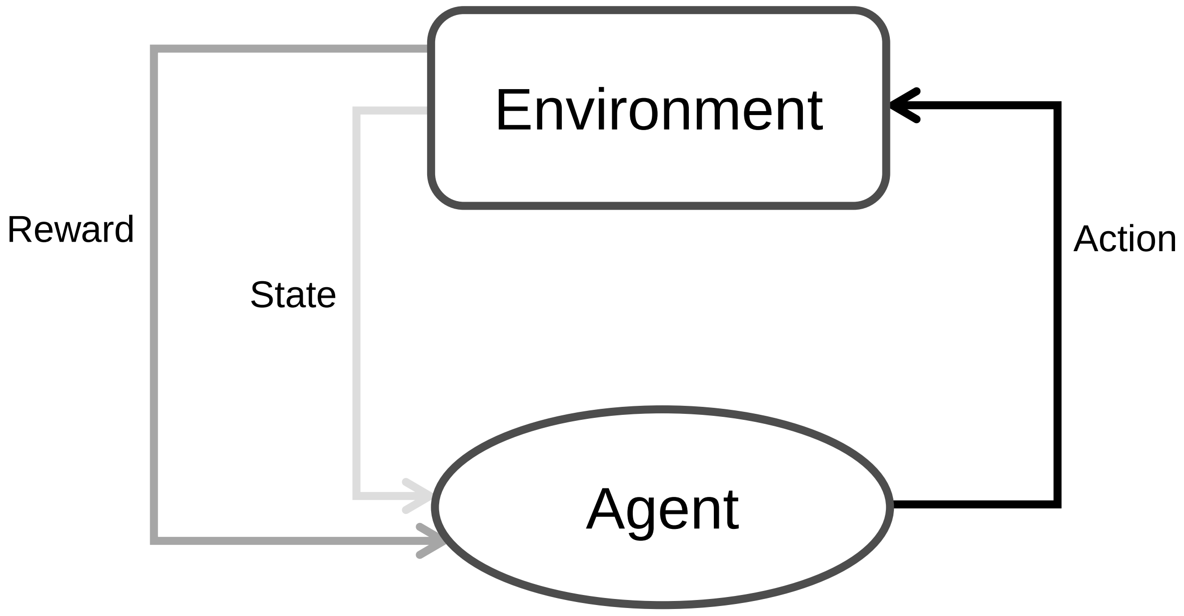


Figure 5 Reinforcement learning elements

or zero, if a state cannot be reached by any action. Having information about the environment dynamics (i.e. the environment model), the agent can easily determine the best way to move in it. This will be better explained in the next pages. In general, there are two different approaches to RL: model-based and 'model-free'

Model-based Reinforcement Learning – Model-based RL

The **model-based** methods are motivated by the idea that a system can be controlled better as more knowledge is available about the system (Kamalapurkar, Walters, Rosenfeld, & Dixon, 2018). In model-based RL an estimation is made on the state transition function, and that is used for improving the policy. In this thesis, first, knowledge (learning) about the system/environment is collected. This is used as it was the correct model of the environment in order to improve the policy. After improving the policy, the policy is run again in order to get more samples and thus, more knowledge about the environment, and learning starts again.

The used model-based RL algorithm is represented in Algorithm 1 and Figure 6 Model-based Reinforcement Learning Algorithms

1. *Fit a model / estimate the return - Learn the probabilities of moving inside the environment.*
2. *Improve the policy*
3. *Generate samples (run the -new improved- policy)*

Algorithm 1 Model-based RL

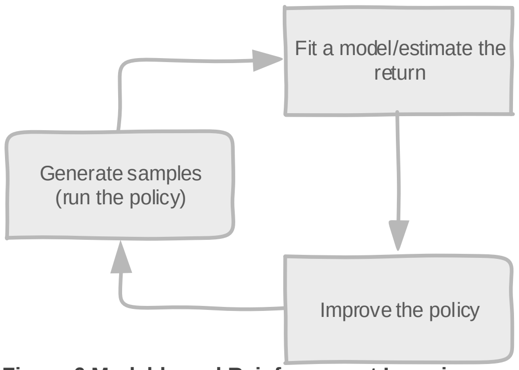

Fit a model/estimate the return
Generate samples (run the policy)
Improve the policy

Figure 6 Model-based Reinforcement Learning Algorithms (Levine, 2018)

Passive Adaptive Dynamic Programming – PADP

Dynamic programming is a powerful technique for optimization and comprises breaking the problem into smaller subproblems that are not independent. An **adaptive dynamic programming** (or **ADP)** agent works by learning the transition model of the environment as it goes along and solving the corresponding Markov decision process using a dynamic programming method. For a **passive** learning agent, this means plugging the learned transition model and the observed rewards into the Bellman equation (Equation 1) to calculate the utilities of the states (Russell & Norving., 2003). As seen in the book *Artificial Intelligence, A Modern Approach* (Russell & Norving., 2003), the basic structure of the algorithm is to first start with:

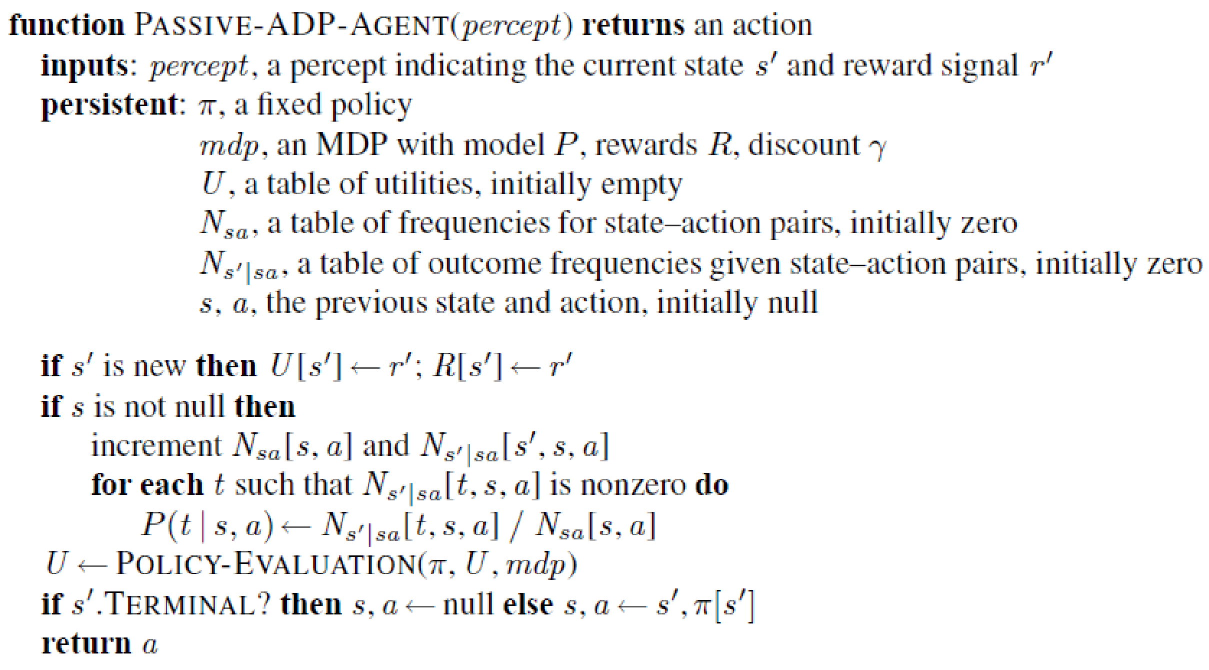
- empty lists representing:
    - the model/MDP (mdp),
    - the utilities for each state (U),
    - the frequency list for the times the agent visited a state and made a certain action (N_{sa}),
    - and the times the agent visited a state, made a certain action and ended in a certain other state ($N_{s'|sa}$).
- As well as,
    - Empty (null) current state s
    - Empty (null) action made in current state a

- a fixed policy π

And with each step of the agent to increase the lists' values from the states visited and occurred. Figure 7 displays the entire algorithm, as seen in the above-mentioned book.

Policy Evaluation

Is the same as in PI.



```
function PASSIVE-ADP-AGENT($percept$) returns an action
  inputs: $percept$, a percept indicating the current state $s'$ and reward signal $r'$
  persistent: $\pi$, a fixed policy
              $mdp$, an MDP with model $P$, rewards $R$, discount $\gamma$
              $U$, a table of utilities, initially empty
              $N_{sa}$, a table of frequencies for state–action pairs, initially zero
              $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
              $s$, $a$, the previous state and action, initially null

  if $s'$ is new then $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  if $s$ is not null then
      increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
      for each $t$ such that $N_{s'|sa}[t, s, a]$ is nonzero do
          $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \,/\, N_{sa}[s, a]$
  $U \leftarrow$ POLICY-EVALUATION($\pi$, $U$, $mdp$)
  if $s'$.TERMINAL? then $s, a \leftarrow$ null else $s, a \leftarrow s', \pi[s']$
  return $a$
```

Figure 7 A passive reinforcement learning agent based on adaptive dynamic programming (Russell & Norving., 2003)

Exploring and Exploiting

PI is great in finding the optimum policy and ADP is great in calculating the utilities for each state given a policy. However, the agent does not explore sufficiently the environment. It just uses a path towards its goal which is dictated by the current policy, which may be further improved by PI, but it may not always result to the optimal policy. Thus, the learning process was **enhanced with exploration and exploitation.**

During the exploitation phase **the agent tries to play it safe** by following the best action at each state, i.e. the actions that result to the states with the greatest calculated utilities. Therefore, it follows the so far given and improved policy.

On the other hand, **during exploration the agent aims to actions not yet tried at states, potentially resulting to states not yet visited**. In doing so it acts optimistically, considering that actions never tried may provide larger utilities. In doing so it gathers new information about the environment. In this way the agent gets to explore more states of the environment and evaluate what it would be a better choice in the long term.

After been given a reward the agent calculates the state’s utility. The state’s utility represents how useful is for the agent to go to this state in the future in contrast to other states in order to achieve/reach its goal. The greater the utility the better. The end state has the maximum utility possible for the world**.** You may ask then how does the agent starts. This will be answered later. Taking into consideration the next state, means we must somehow determine which state matters the most. This is basically a math problem we face here but it can easily be solved by adding a parameter of how much should we consider a value. Of course**, the newest the visit of a state the more value it has**.

Implementation

The entire project is a combination of:

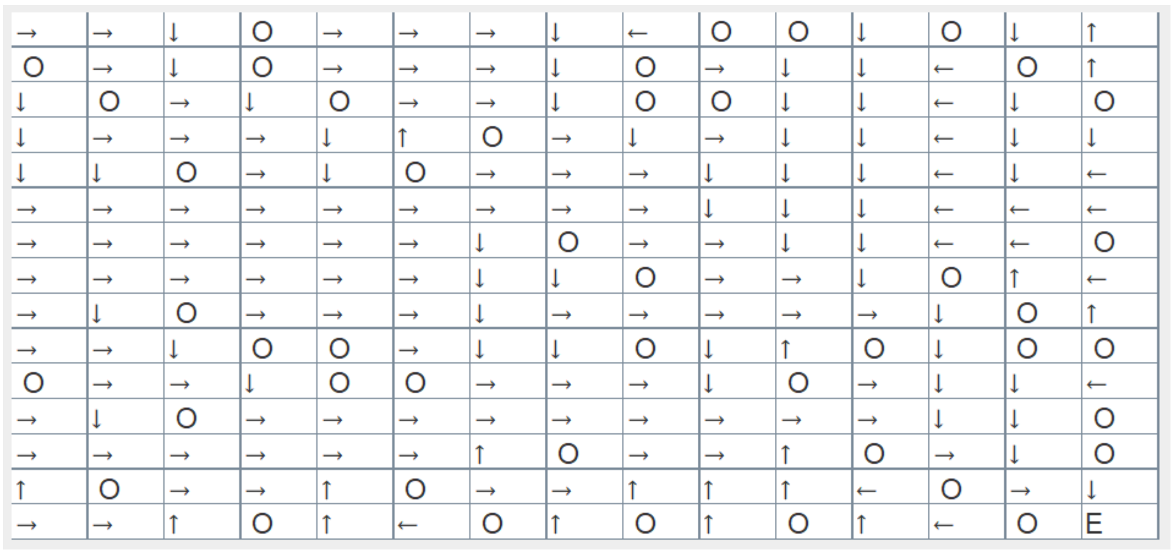
1. Methods concerning the Environment
    a. Generation of the environment
    b. Reward function
    c. Transition probabilities for the states
2. Structures for representing the world
3. Adaptive Dynamic Programming algorithm
4. Policy Iteration algorithm
5. Exploration and exploitation methods
6. Reporting functionality

ADP as well as PI have an implementation based on the book *Artificial Intelligence, A Modern Approach* (Russell & Norving., 2003), as also described above.

Methods for the Environment

The environment is a simplified grid, with the following objects: A start state, an exit state (E), and obstacles (O).
Obstacles, as any other object, including the agent, are positioned in a specific row and column on the grid. If the agent tries to step on an obstacle it will be redirected to another state or even stay where it is. Obstacles in this version of grid-world are represented with the an “O” inside the corresponding cell.

In order to watch the agent’s progress during each run, a grid is displayed to the user showing the changes of the policy, as the agent learns.



→	→	↓	O	→	→	→	↓	←	O	O	↓	O	↓	↑
O	→	↓	O	→	→	→	↓	O	→	↓	↓	←	O	↑
↓	O	→	↓	O	→	→	↓	O	O	↓	↓	←	↓	O
↓	→	→	→	↓	↑	O	→	↓	→	↓	↓	←	↓	↓
↓	↓	O	→	↓	O	→	→	→	↓	↓	↓	←	↓	←
→	→	→	→	→	→	→	→	→	↓	↓	↓	←	←	←
→	→	→	→	→	→	↓	O	→	→	↓	↓	←	←	O
→	→	→	→	→	→	↓	↓	O	→	→	↓	O	↑	←
→	↓	O	→	→	→	↓	→	→	→	→	→	↓	O	↑
→	→	↓	O	O	→	↓	↓	O	↓	↑	O	↓	O	O
O	→	→	↓	O	O	→	→	→	↓	O	→	↓	↓	←
→	↓	O	→	→	→	→	→	→	→	→	→	↓	↓	O
→	→	→	→	→	→	↑	O	→	→	↑	O	→	↓	O
↑	O	→	→	↑	O	→	→	↑	↑	↑	←	O	→	↓
→	→	↑	O	↑	←	O	↑	O	↑	O	↑	←	O	E

Figure 8 A random snapshot of the user experience during the run of the program

Generation of the environment

The generation of the environment the agent would explore, has been custom made into an process that runs at the beginning of the running process. The algorithm was implemented in order to take into account parameters as well as compute metrics throughout the runs.

The parameters are specified through a text file. The initiator prompts to specify parameter values when opening the program. More specifically the parameters available are the following given along with their explanation:

NUMBER_OF_ITERATIONS_BEFORE_PI: The number of times that each ADP (exploring and exploiting) will run and finish (reach the end state) before the PI runs once.

WORLD_HEIGHT: The height of the world. Also, the number of rows that appear.

WORLD_WIDTH: The width of the world. Also, the number of columns that appear.

NUMBER_OF_ITERATIONS_FOR_SAME_WORLD: How many times the agent will iterate in the same environment to reach a policy.

OBJECT_ANALOGY: How many obstacles will be put in the environment. This is an integer. The inversely proportional value of this parameter is actually used. So the bigger the number the fewer the obstacles.

NUMBER_OF_PARALLEL_AGENTS: This is a parameter that currently is not considered. This allows for more than one agent to be put in the environment. For now, this value remains always “1”.

END_STATE_X: The number (index) of the column that the end state will be at.

END_STATE_Y: The number (index) of the row that the end state will be at.

REPORTING_PATH_FOR_FILES: The path in the folder structure to place the generated report after a complete run.

During the environment generation, *obstacles* are placed into the world randomly, in free cells. The specified (in the parameters) end-state is set and at the end the agent is placed on the environment to start its exploration.

The agent in this version of the grid world is allowed to make four actions and these are up, down, left or right.

Reward Function

Each action the agent performs has a result of putting it in a state, sometimes it is the same state as before it made the action. The reward for getting to a state, is calculated by:

1. `maxSum` = total number of rows plus the total number of columns in the grid plus 1.
2. sumOfAgent = row number plus column number the agent is positioned on the grid
3. reward = $\frac{\text{sumOfAgent} \cdot 100}{\text{sumOfAgent}}$
4. if the agent’s action would lead to an obstacle then it is multiplied by 0.8
5. if the agent’s action would lead to a grid border then it is multiplied by 0.7

The reason for the greater decrease in case of a wall is that this action for sure would not lead to the end state since it points away from the grid.

The reward is between zero and one hundred (0-100) and it is made to resemble the percentage of how close the agent is to the end state. Keep in mind, the utility is different that the reward, utilities are calculated using Bellman equations as seen in the former chapter.

Structures of the world

A basic structure was implemented in order to keep track of the:

- states,
- the obstacles,
- the policies,
- the MDP
- other useful objects, like ReportFileEnum an enumeration to keep the names of the report files.

Adaptive Dynamic Programming & Policy Iteration algorithms

Both algorithms' implementations are very close to the reference book (Russell & Norving., 2003), as described above. It is worth to say that the exploration and exploitation were made in the ADP algorithm whereas the policy improvement in PI.

Exploration

The exploration is based on probabilities. First, we need to define the parameters:

`shouldExploreProbability` = a number, to determine the probability of explorating (i.e. choosing a random action). This number is a constant.

`loopNumber` = the iteration of the ADP the agent has reached since the beginning of the program run. Updated within each step (before the decision on the next action) of ADP.

`shouldExploreLoopNumber` = `loopNumber` times the `shouldExploreProbability` (`loopNumber x shouldExploreProbability`). Updated when the the below parameter `shouldExplore` is true.

`shouldExplore` = true if `loopNumber` is equal or greater than the floor of `shouldExploreLoopNumber`. Updated in each step of the ADP.

So as we see in the above defined parameters, the algorithm explores (it chooses randomly an action, rather than the one determined by the policy) every time the `shouldExplore` parameter is true. This parameter is set according to the `loopNumber`, `shouldExploreLoopNumber` and `shouldExploreProbability.` As observed, the greater the `loopNumber` the agent is in the less likely is for it to explore. That happens since the factor of `shouldExploreProbability` is at maximum 1.009 or even less, for example:

`shouldExploreProbability` = 1.009

`loopNumber` = 10000

`shouldExploreLoopNumber` = 10000 x 1.009

`shouldExplore` = will become true at 10090, this is 90 iterations after the current one.

Reporting Functionality

In order to measure the overall algorithm’s performance and accuracy, some reporting functionality has been implemented.

For the generation of report files, *JFreeChart* and *itextpdf* libraries where used. All report files from every run in a machine can be found in the path specified in the configuration file variable name *REPORTING_PATH_FOR_FILES.* Inside this path a folder is created for each run. The name of each subfolder depends on the date and time of its run. For example, a run **started** in the 26th of June of 2019 and at local time 11:54:35 according to a 24hour clock, would generate the folder Run20190626115435.

Run Report

This is the most important and key report to validating the results. It can be found under *REPORTING_PATH_FOR_FILES* and under the subfolder of its run, named as RunReport_ and the date and time in a PDF file format.

Policy progress Report

This report contains all the calculated policies. More specifically, inside this file there is one policy for each PI loop. It can be found under *REPORTING_PATH_FOR_FILES* and under the subfolder of its run, named as Policies in a PDF file format.

Utilities progress Report

Contains the utilities for each state for each loop of PI in the following format:

*PI_RUN***=**[STATE1_ID]**:**[STATE1_UTILITY]**;**[STATE2_ID]**:**[STATE2_UTILITY]**;**… etc.

For example,

```
23=[4,4]:7909.810183434268;[3,7]:9151.310469566046;[0,8]:5425.84704100715
1;[13,4]:11218.99140871583;[12,10]:16185.64849809536;[2,2]:5271.754956769
583;[5,1]:5962.087136731263;[12,7]:14613.271668487436; etc.
```

It can be found under *REPORTING_PATH_FOR_FILES* and under the subfolder of its run, named as Utilities in a TXT file format.

General report

Contains information on every run made in this machine. It can be found directly under *REPORTING_PATH_FOR_FILES* and named as GeneralReport in a TXT file format. Each line in this file represents a run and is a short form of describing the RunReport.

Results and Validation

The machine used to run the experiments is a laptop with:

- Operating System: Windows 10 64-bit
- Processor: Intel® Core™ i7-8750H CPU @ 2.20GHz
- Installed Memory (RAM): 16,0 GB

Factors affecting the running time are:

1. The number of obstacles in relation to the entire grid (object analogy)
2. The density of the obstacles in relation to the entire grid
3. The size of the grid (number of columns and rows)

These three factors do not have the same weight in the analogy of effectiveness to the performance. Having said that, factor 1 has little to no effect depending to factor 2 and factor 3 has as almost no effect to the performance if no obstacles exist. This happens because, even if the world were to have 10% of its size as obstacles but condensed to one area the algorithm would still be very quick since it would consider this area to be merely a big obstacle. Going around one object, even if that is big, is very simple since once it is overcome, it will be like nonexistent for the agent. However, if the density of the obstacles is low, meaning they are splattered all around the world, it makes the agent lose a lot of time to get out of a dead end or working its way around one obstacle after another.

Factor 1 is also connected to potential issues during run. For example, when the states-obstacles analogy gets smaller it is very easy for the generation of the environment algorithm to create dead ends and block the agent from moving at all (dead starts) as well.

Results from reporting functionality

Sample RunReports about the performance and accuracy of the overall algorithm are displayed and discussed in this section.

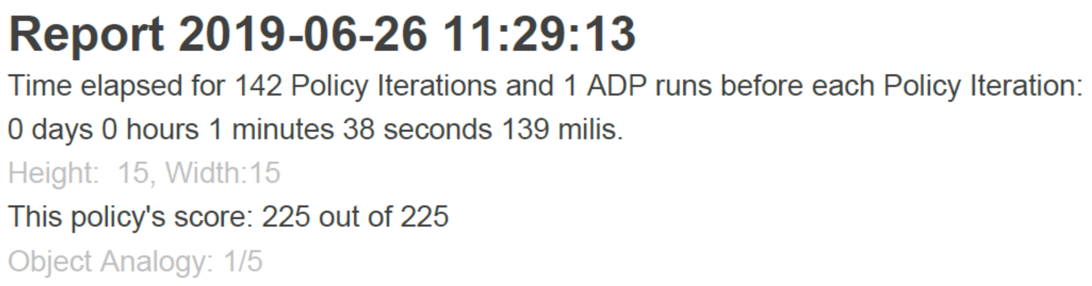
RunReport, contains information on the following metrics:

1. the time it took for the agent to converge to policy,
2. number of policies the agent considered before converging,
3. number of columns and rows of the grid
4. the policy the agent concluded to, as well as

5. diagrams with samples from throughout the run:
    a. the average of utilities of every state per loop
    b. progression of utilities for specific states (diagonal states of the grid) per loop
    c. policy's score in the end of each run of PI
    d. visit rates in the end of each run of PI in relation to starting from a particular state and making a certain action (state-action) and starting from a particular state, making a certain action and ending up in a specific state (state-action-state)
6. Policy's score is based on whether the actions point in obstacles, or whether action result to an oscillation between adjacent cells. It is implied that since some states are completely surrounded by obstacles and thus unreachable, they count as minus points if they too are pointing in an obstacle or the wall.

For the policy found, we indicate in red the actions-arrows that are somehow conflicting with the end-state-diamond, like pointing on an obstacle-rock or outside the grid, or at each other. The actual outcome of the implementation is in most parts as expected.

Let's start analyzing from the top:


Report 2019-06-26 11:29:13

Time elapsed for 142 Policy Iterations and 1 ADP runs before each Policy Iteration:
0 days 0 hours 1 minutes 38 seconds 139 milis.

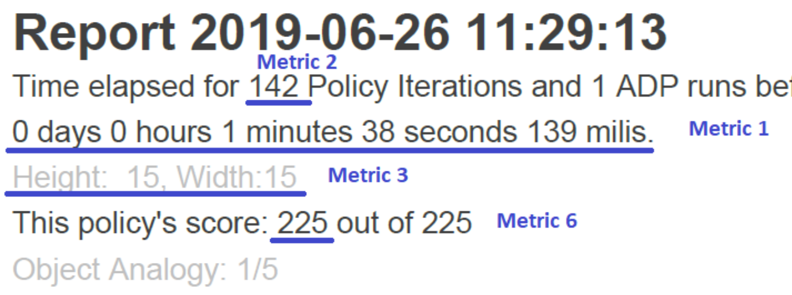
Height: 15, Width:15

This policy's score: 225 out of 225

Object Analogy: 1/5

Figure 9 Original top of report

These are the metrics 1,2,3 and 6:



Report 2019-06-26 11:29:13

Time elapsed for 142 Policy Iterations and 1 ADP runs before each Policy Iteration:

0 days 0 hours 1 minutes 38 seconds 139 milis.

Height: 15, Width:15

This policy's score: 225 out of 225

Object Analogy: 1/5

Figure 10 Commented top of report

It is a small world 15 x 15 and the obstacles are sparse but not too many, this is an average run with such specifications. The maximum time for such a world has been about 3 minutes whereas the minimum about 55 seconds.

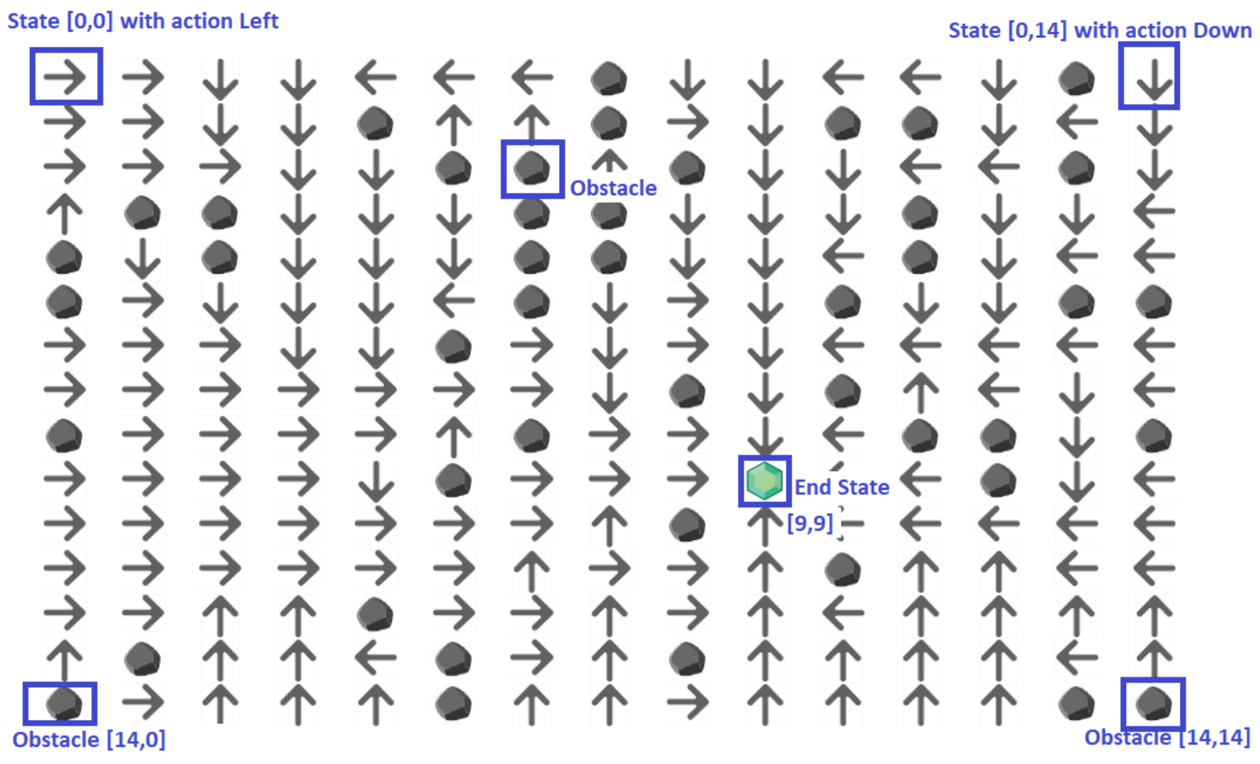


Figure 11 Commented policy display in report

The above policy seems fine since no red arrows are presented. We shall now discuss the performance of the algorithm and how it converged to the above policy. Below, in Figure 12, we present the evolution of the average utility gor all states, per loop. The Y axis shows the average utilitiy. where the horizontal X axis shows the number of loops of the PI algorithm.

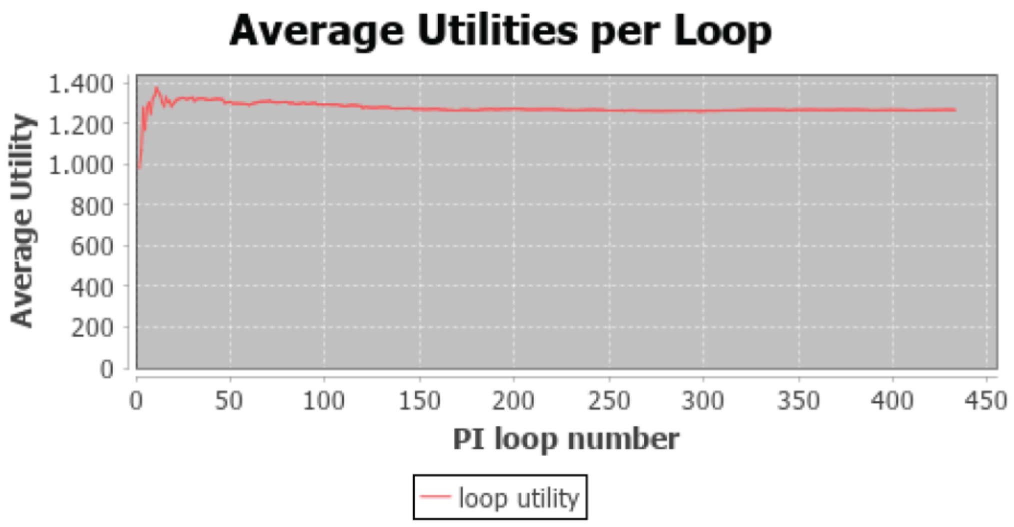


Figure 12 Average utility per PI loop Diagramm as seen in reports

As we observe little after the start the average utility seems to hit a pick, this is because the agent has yet to explore all the states and the discounting has just started for each utility. Each time a new state is observed and added to the agent's model of the environment the average utility gets bigger. The same applies in the following diagram, showing the evolution of utilities of specific states (in this case states on the diagonal of the grid).

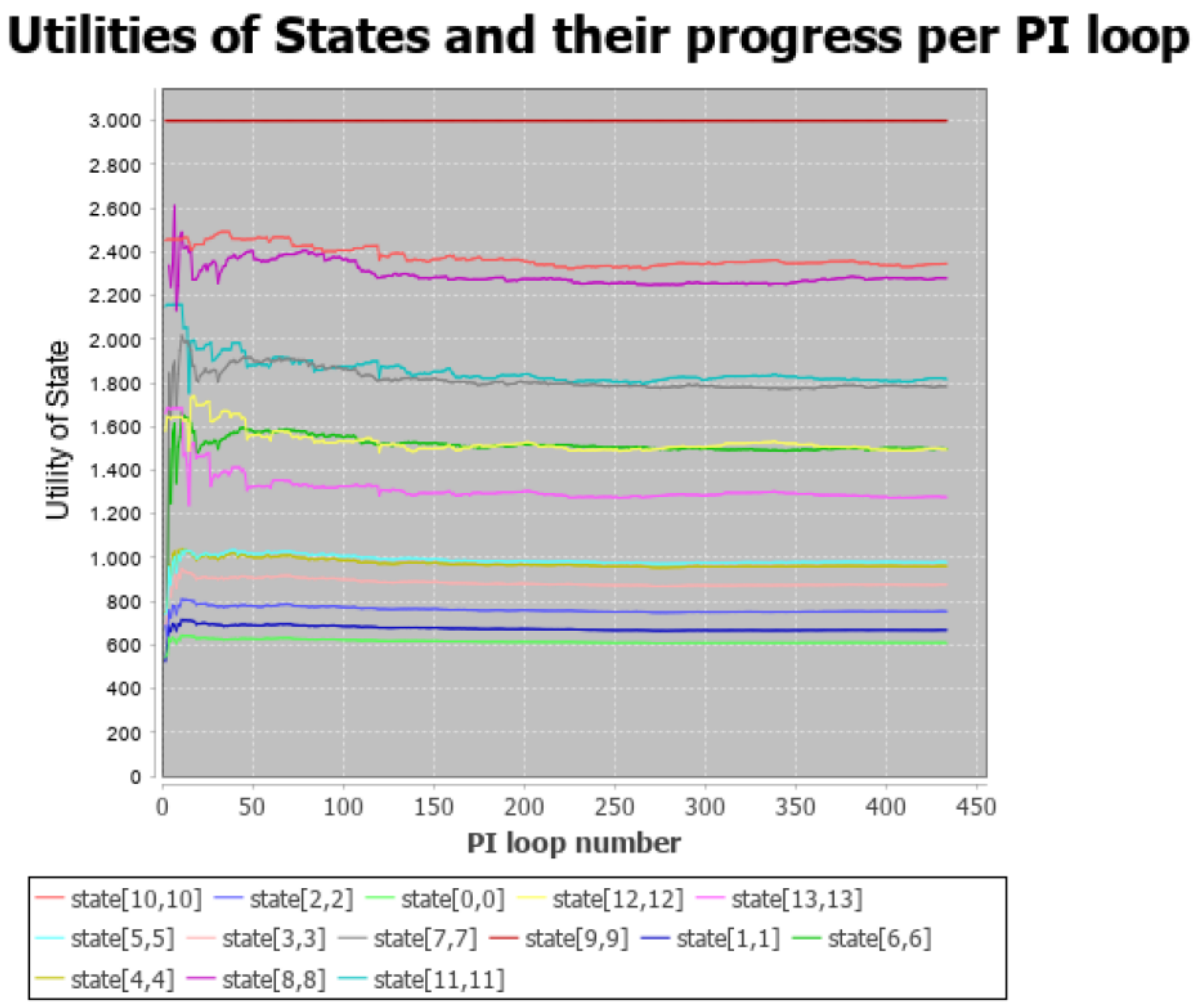
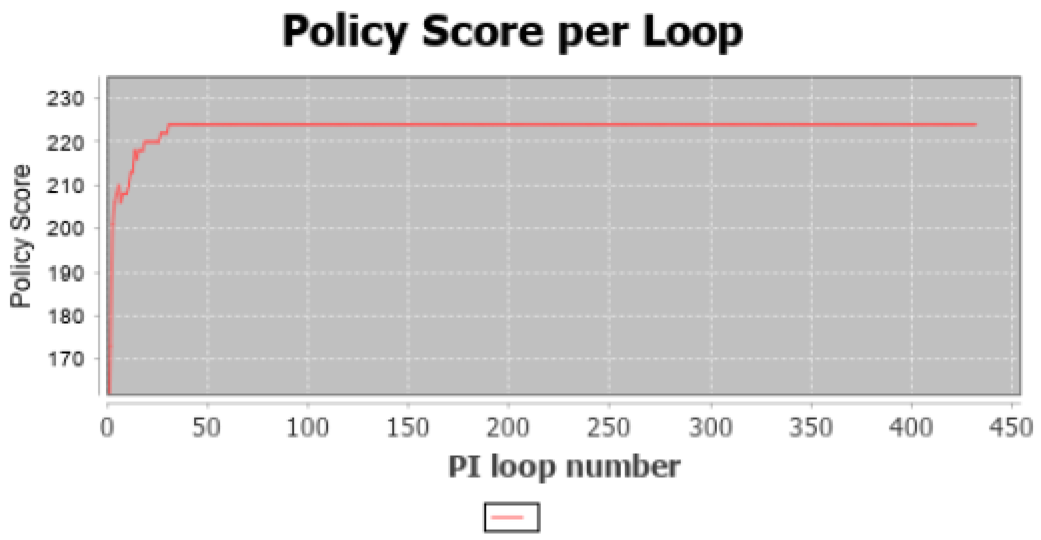


Figure 13 Sample state utility progress per PI loop

The red line on the top is of the state with ID [9,9] which was the configured *end state*. Naturally, we see that the further a state is from the end state the less utility it has, the closer the agent gets to the end state the better the utility. What is more, we see that state with ID [6,6] has greater utility that the [8,8], that happens because state [6,6] is a very key point for overcoming obstacles in the surrounding positions.

Figure 14 Policy score progress per PI loop



The policy score diagram helps us see the progress made, as far as the "quality" of the policy computed in each loop. The criteria used are very basic this is why the score is maxed out even before loop 50.

Last, the visit rate progress diagram in Figure 15 shows the average visits to state-action pairs and state-action-state triples, per PI loop. The state-action visits are about triple the

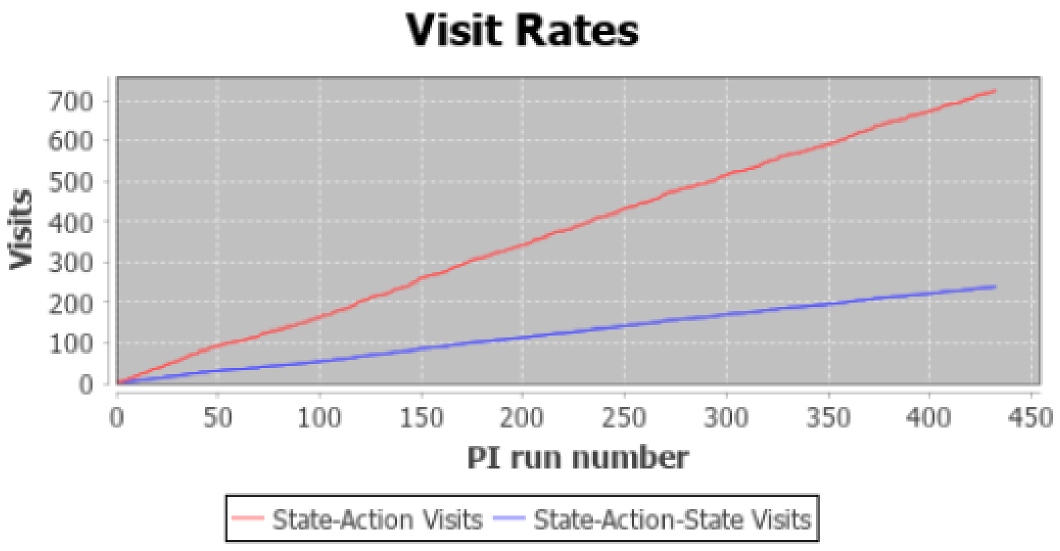


Figure 15 Visit rates diagram per PI loop

number of state-action-state rates and that is normal since the agent should choose between three actions in order to move closer to its goal.

Three and not four because stepping in the opposite direction than that of its goal/end state would lead it further away, but it may follow a side action in case in needs to overcome an obstacle..

In conclusion, the algorithm produces accurate results and this is validated by the report files as well. Firstly, the policy the agent concluded is correct and no state is left unvisited (that is because of exploration). Secondly, from the diagrams in the RunReport we see that it is persistent in the utilities comparing the average utilities with the specific state utility progress. What is more, in the last diagram we observe that the number of visits per state are proportional to the PI loop number.

Possible Improvements

As stated in The Goal section, the purpose of this thesis was to combine PI and ADP algorithms of RL, and in the process learn more about RL in general. From these, most have been achieved, but there is always space from improvement.

About the combination

The combined idea could be enhanced even more, by applying more thought in the combination of the two, or even the addition of more theory and algorithms. For example, introduce an algorithm to more accurately calculate rewards. Also, a more in-depth study can be made in the way these two algorithms are combined. The study as such could be based on various topics. One could be an introduction of an algorithm to calculate how many times the ADP should loop before going into PI, and not have it be a constant number in configuration.

About the implementation

Of course, there are uncountable improvements to be made in the implementation.

To start with, the addition of more than one agent, the communication and collaboration of the ones in sight. In order to do this, an introduction of new features would be needed, like the blocks an agent can see, the angle of its sight (should it not be 360^o), the type of communication the agents would have with one another. Also, additional thoughts on this would be if they were to be collaborators or competitors, will the environment treat them equally, and if not under what circumstances and in what matter.

Another improvement can be to make the generation of the environment “smarter”. By smarter we mean create the obstacles not randomly but in a more structured way. For example, the creation of random labyrinths, with dead-ends and a couple of different paths leading to the end state.

What is more, we could implement the agent to start from various points, this might make the agent converge to a policy quicker, since it will not have to constantly go from one end to another in each iteration and still quickly figure out the entire MDP and make the connections from different angles.

As an overall experience it has been a great journey to the paths of AI and given the time and circumstances, I am happy with the result.

References

Isbell, C., Littman, M., & Pryby, C. (n.d.). *Reinforcement Learning, Offered at Georgia Tech as CS 8803*. Retrieved from Udacity, In Collaboration With Georgia Institute of Technology: https://www.udacity.com/course/reinforcement-learning--ud600

Busoniu, L., Babuska, R., Schutter, B. D., & Ernst, D. (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximators.* Boca Raton, FL : CRC Press (an imprint of Taylor and Francis Group, LLC, an informa business).

Farrukh, A. (2017). *Practical Reinforcement Learning.* Packt Publishing.

Garcia, F., & Rachelson, E. (2010). Markov Decision Processes. In O. Sigaud, & O. Buffet (Eds.), *Markov Decision Processes in Artificial Intelligence.* ISTE Ltd, John Wiley & Sons, Inc.

Kamalapurkar, R., Walters, P., Rosenfeld, J., & Dixon, W. (2018). *Reinforcement Learning for Optimal Feedback Control, A Lyapunov-Based Approach.* Springer International Publishing . Retrieved from https://link.springer.com/book/10.1007%2F978-3-319-78384-0

Katja, V., Ann, N., Peter, V., & Maarten, P. (2008). Multi-Automata Learning. In C. Weber, M. Elshaw, & N. M. Mayer, *Reinforcement Learning Theory and Applications.* Vienna, Austria: I-Tech Education and Publishing.

Kolter, J. Z. (2016, Feburary 29). *Markov Decision Processes.* Retrieved from Carnegie Mellon University, School of Computer Science: http://www.cs.cmu.edu/afs/cs/academic/class/15780-s16/www/slides/mdps.pdf

Levine, S. (2018, 08). *Deep Reinforcement Learning, CS 294 - 112*. (University of California, Berkeley) Retrieved from YouTube: https://www.youtube.com/watch?v=ml8wUkE0M6U&list=PLkFD6_40KJIxJMR-j5A1mkxK26gh_qg37&index=22

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of Machine Learning* (2nd ed.). (F. Bach, Ed.) Cambridge, Massachusetts: The MIT Press.

P., B. D. (n.d.). Dynamic Programming: Deterministic and Stochastic Models.

Russell, S., & Norving., P. (2003). *Artificial Intelligence: A Modern Approach.*

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: an introduction.* London, England: The MIT Press.

Szepesvari, C. (June 9, 2009). *Algorithms for Reinforcement Learning, Draft of the lecture published in the Synthesis Lectures on Artificial Intelligence and Machine Learning* . Morgan & Claypool Publishers.