Introduction to Intelligent, Cognitive, and Knowledge-Based Systems

Final Project

David Levin

June 19, 2022

1 Introduction

The final project involved building and describing a complete general agent that operates in multiple domains and carries on a variety of tasks in these domains. The problems formulated in the syntax of PDDL[1], and the agent needs to plan, learn, and execute in discrete space environments, handling deterministic and probabilistic action models – in environments with partial information.

When I approached the solution to the problem, I reviewed the various topics and approaches we learned, and while reading various articles on RL subject – I formulated for myself the way of the solution that I will present below. As a result, I decided to focus on the following:

1. Separation between deterministic and non-deterministic problems
2. Using the *avoid the past* technique
3. Dealing with a new world without learning
4. Building a reward function with an emphasis on the transition from model based to model free
5. How to proceed after reaching the goal - that is, how to make the most of the learning time

In the following pages I will address each topic in detail and present the agent's learning process.

[1] PDDL is a standard encoding language for classical planning tasks ; Based on STRIPS notation.

2 Architecture

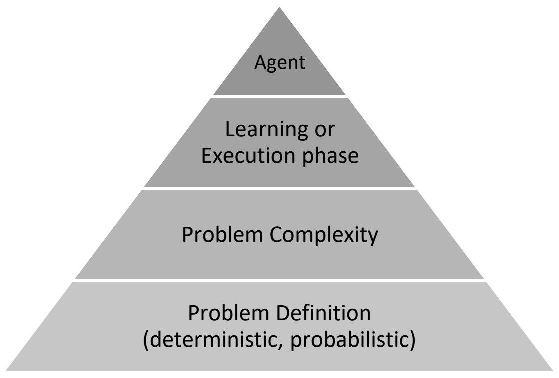
2.1 Initialization

In order for the agent to know how to solve the problem in the best way, he performs a number of actions before approaching the problem:

- Is the domain deterministic?
  Seemingly, this question can be answered very easily, with simple examination of the given domain file. The problem is that we may get a model that presents probabilistic actions – even though in practice they are not. Thus, such cases are also addressed in this model examination.
- Are we missing information to complete the task?
  As before, ostensibly the examination is very simple: parsing the domain file and searching for the *reveal*[2] keyword. Again – the information will not always be presented in this way – I will discuss later how the agent reacts in such situations.
- What is the approximation complexity of the given problem?
  Because at this point of time, the agent doesn't know the scale of the problem, we make an approximation of the complexity of the given problem.
  We perform an initial separation into 2 types of problems: **Easy** and **Hard** problems, when the separation is based on the following values:
  - Number of goals
  - Number of objects*
  - Number of actions*
- Is the agent in the learning or execution
  This information is provided by the user, using a dedicated flag.

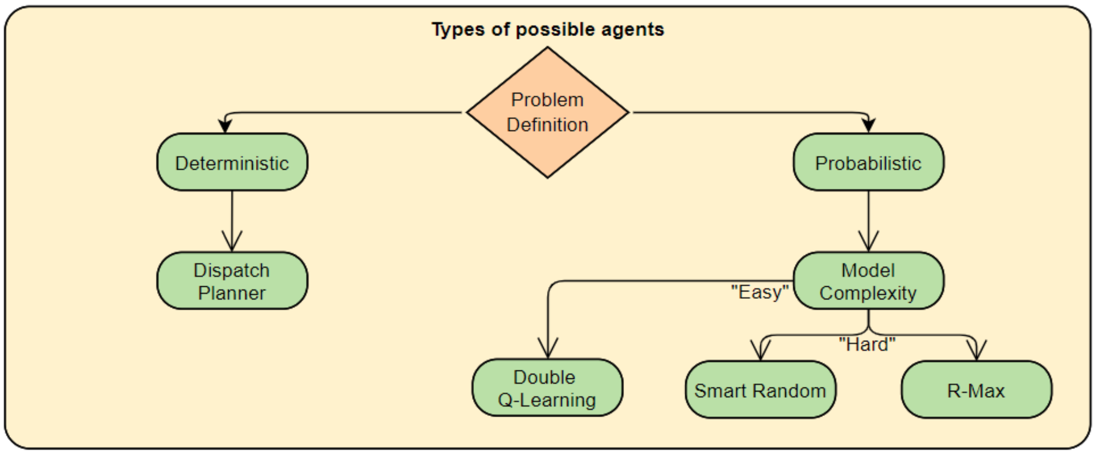
2.2 The Agent

The agent is built in a gradual process, where at each stage he builds on the conclusions of the previous stage, according to which he understands how he must "evolve". The process can be described as follows:


Agent
Learning or Execution phase
Problem Complexity
Problem Definition (deterministic, probabilistic)

[2] Used to describe an expression from the form <condition>-<result>. If the agent has completed the condition, the domain will reveal new information that it did not know before.
* This is an initial value that can be obtained from parsing the given domain and problem files.

For example, the decision-making process regarding the identity of the agent is carried out when we are in the process of learning (I will expand on the execution process later):



2.3 The Agent purpose

The main goal of the agent is to solve problems as quickly as possible. When it comes to deterministic problems, he will use outside services that will help him solve the problem. On the other hand, if it is probabilistic problems, he is trying to construct himself up gradually (as I presented earlier).

When the agent is exposed to a new problem, his goal will be to study it comprehensively, so that he can solve it in the future (if requested) quickly. The learning process of the agent performs using RL techniques and algorithms we saw in the course throughout the semester, extensions from various academic papers and more improvements based on my personal experience while writing the work.

3 Learning process

3.1 Deterministic world .vs. Probabilistic world

Before we define the type of agent, we examine whether it is a deterministic or probabilistic world.

Assuming this is a deterministic world – the agent exits the program with code 128 (as we were asked to do in case we do not want to use learning). In such a situation I preferred to give up the learning process and use directly a pddlsim agent called Dispatch Planner.

Assuming we are in a probabilistic world, we would like to test the level of complexity of the model and define the type of agent accordingly. As I mentioned earlier, there are 2 main definitions for the level of complexity of a model: easy and hard.

3.2 "Easy" .vs. "Hard"

In case of an easy problem, we will use a RL model from called Double Q-Learning.

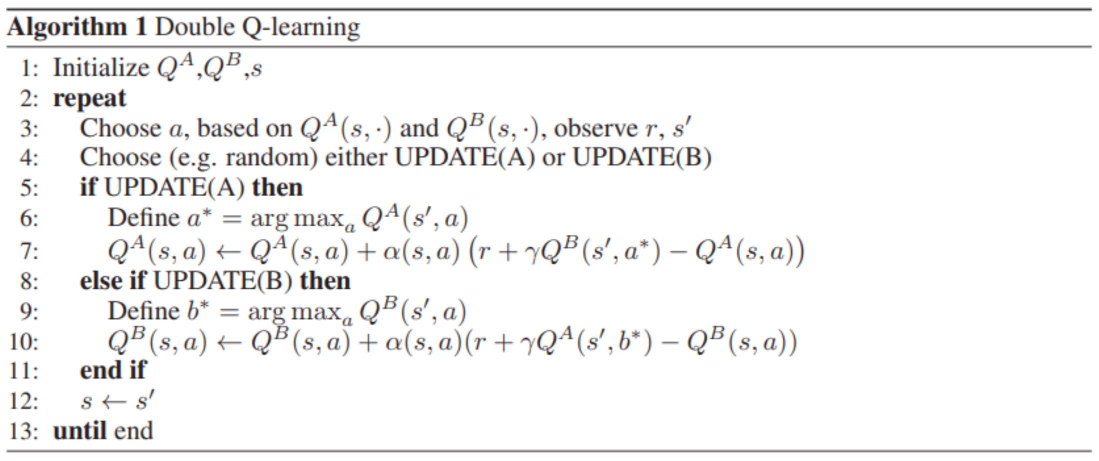
According to the agent, a problem will be defined as a "easy" problem when it meets one of the following parameters:

1. Number of goals is 1
2. Number of goals is 2/3 and number of actions times number of objects is less than 300[3].

The need to limit the size of the problem, which is expressed, among other things, by the number of operations and the number of objects, stemmed from a fundamental problem that the Vanilla Q-Learning algorithm has: memory shortage and policy unable to converge.

After realizing an agent based on the classic algorithm, I was exposed to an article by Hado Van Hasselt[4], which explained an improved algorithm called Double Q-Learning. The article shows that in certain stochastic environments, the classical algorithm malfunctions, caused by large overestimations of action values. According to Hado, the problem with the decision stems from the use of the maximum action value as an approximation of the expected maximum action value – which can sometimes lead to incorrect and irrational learning.

Using the Double Q-Learning algorithm, the decision may "hurt" the expected maximum value, just as it can "increase" the expected maximum value. Such an action may lead to balancing and reducing the problem. Pseudo code of the algorithm:



Algorithm 1 Double Q-learning

```
1: Initialize $Q^A$,$Q^B$,$s$
2: repeat
3:   Choose $a$, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe $r$, $s'$
4:   Choose (e.g. random) either UPDATE(A) or UPDATE(B)
5:   if UPDATE(A) then
6:     Define $a^* = \arg\max_a Q^A(s',a)$
7:     $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
8:   else if UPDATE(B) then
9:     Define $b^* = \arg\max_a Q^B(s',a)$
10:    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
11:  end if
12:  $s \leftarrow s'$
13: until end
```

Note that we use the familiar equation of Q-Learning:

$$\hat{Q}(x,a) \leftarrow \beta \cdot \left(r + \gamma \hat{V}(y)\right) + (1-\beta) \cdot \hat{Q}(x,a)$$
$$\hat{V}(x) \leftarrow \max_a \hat{Q}(x,a)$$

[3] After several attempts with different values, the value that resulted in the most consistent and good results is 300 (tested using the *grid search* method)
[4] There is a reference to the article in the end of this work.
* Based on the explanation I presented at the end of the previous page.

In case of a hard problem, we will use a RL model from called R-Max.

According to the agent, a problem will be defined as a "hard" problem when it meets one of the following parameters:

1. Number of goals is higher than 3.
2. Number of goals is 2/3 and number of actions times number of objects is higher than 300*.

In problems whose space of situations and actions is defined as large by the agent, the agent will prefer to use a model-based algorithm. I chose to use the R-Max algorithm which can attain near-optimal average reward in polynomial time. During the learning phase, the agent maintains a complete model of the given world environment. With this data, in the future (execution time), the agent will act based on the optimal derived from this data.

3.3 Reward Function

While writing the reward function, it was important for me to focus on 2 key elements: providing compensation based on the rate of achievement of the goals and punishment according to the principle of avoid the past.

In order to implement the above idea, after each action of the agent, the agent performs the following actions:

- Save the last action the agent performed
- Save the last state that the agent was in
- Number of goals the agent has achieved in each unit of time
- The level of the agent's progress in achieving the overall goal

The final reward function is:

Reward (s, a, s'):

```
if s == s':
        return -3
if s was visited in the last 5 actions:
        return -20
if s was the last goal to achieve:
        return 100
if remaining_goals(s) > remaining_goals(s'):
        return -50
if remaining_goals(s) < remaining_goals(s'):
        return 100
else:
        return -1
```

3.4 Export the learning results

At the end of the learning phase, the agent saves the learning outcomes in a dedicated file that will assist him in the execution phase. The form of file saving varies depending on the type of agent, and is defined as follows:

- Dispatch Planner:
  As explained above, if the problem is defined as a deterministic, then we skip the learning phase. Therefore, in such a case we will not save any file.
- Double Q-Learning:
  In the Vanilla algorithm, we create a single file in which we determine the reward values calculated throughout the learning phase. In the "Double method", we use 2 Q-tables, so we will create 2 files (each table in a file).
- R-Max:
  We create one file that contains the probabilities of performing each action. Recall that we run a 'smart random' agent, and throughout the learning phase we calculate the probabilities of performing a certain action from a certain situation. This is how we manage to obtain more information about the model.

  In the last two cases, we will save the data in a JSON file, which will contain information in the following form:

```
{
      "H($state_t$)":
            [{"$action\ a_i$": $\alpha$},
             {"$action\ a_j$": $\beta$},
              …
            ],
      "H($state_k$)": …
}
```

Before the agent begins to execute the learning phase, he will check whether there is a file/s suitable for the problem and the environment described in the given problem. If so – the agent will parse the data and continue to learn from where he last stopped. Otherwise, he will start the learning process from "zero".

3.5 Multiprogramming

Python has a mechanism called GIL. Each process running in Python has its own interpreter that "locks" when performing an operation. This means that one process can only run one command on the processor, even if the process consists of different threads and there are several cores in the processor. Because the program in which the agent is acting is CPU bounded, if we choose to use some threads, it will not lead to parallelism, and may even impair the effectiveness of the program.

Hence, the need arose to use a number of processes. For learning probabilistic tasks, the program's principal agent creates a parallel system as follows:

```
1.  Initialize multiprogramming.manager
2.  While TRUE:
        2.1.  Create $\alpha$ new processes
        2.2.  Set the target function of each process as the *learning* function
        2.3.  Execute all the processes
        2.4.  Wait until all the processes finished the *learning* process
        2.5.  Parse execution results
3.  Save the relevant data in JSON file
```

In this way, the main agent creates sub-agents who perform the learning process (each one independently), and at the end of each iteration[5], when all sub-agents complete the learning process – we use the run results to determine which sub-agent performed the best learning. Thus, we increase the chance that the main agent will get a better and safer policy.

4 Execution process

4.1 Deterministic world .vs. Probabilistic world

As I explained earlier, if we are in a deterministic world, we will use a pddlsim agent called Dispatch Planner.

Otherwise, if the world is not deterministic, we distinguish between 2 different situations:

1. If a learning process has been performed for the specific problem – we will use the learning file we saved for the problem. We will define an agent according to the type of learning performed for the problem.
2. Otherwise, suppose we get a new problem that the agent has not seen before, so we'll run an "upgraded random" agent:
    - Avoid the past – The agent will keep in queue the last 10 actions he has performed, and he will prefer to perform actions that are not in the queue.
    - Avoid circles – In case of multiple failure for a specific action, the agent will avoid performing this action again – in order to avoid a situation where he will enter an infinite loop.

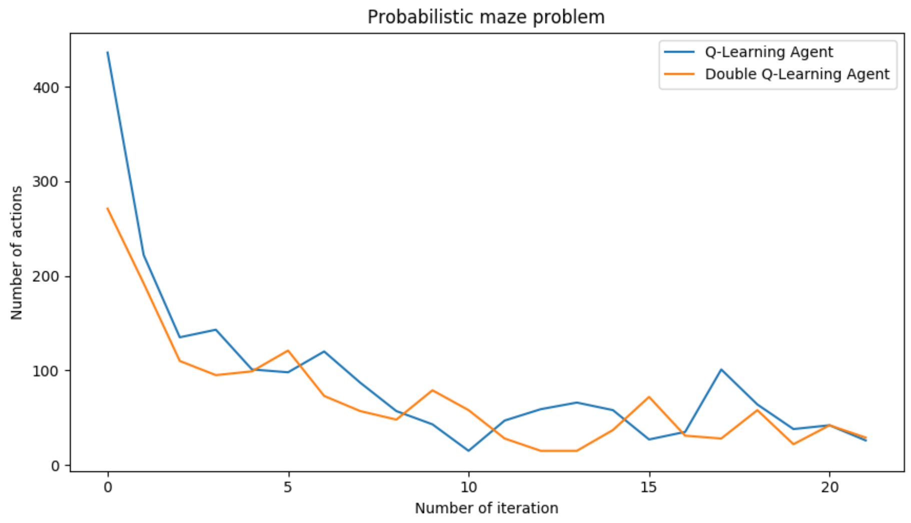
5 Performance Analysis

5.1 Vanilla Q-Learning .vs. Double Q-Learning

The decision to use an agent based on the Double Q-Learning method instead of a Vanilla Q-Learning agent is based, among other things, on comparing a number of different domains and problems – which led to an unequivocal conclusion (except in a few cases) that this method is preferable. It is important to note that as the problem became larger (average number of actions required to achieve the goal) – the differences became more noticeable and significant.

[5] Each iteration starts in stage 2 in the pseudo code description.

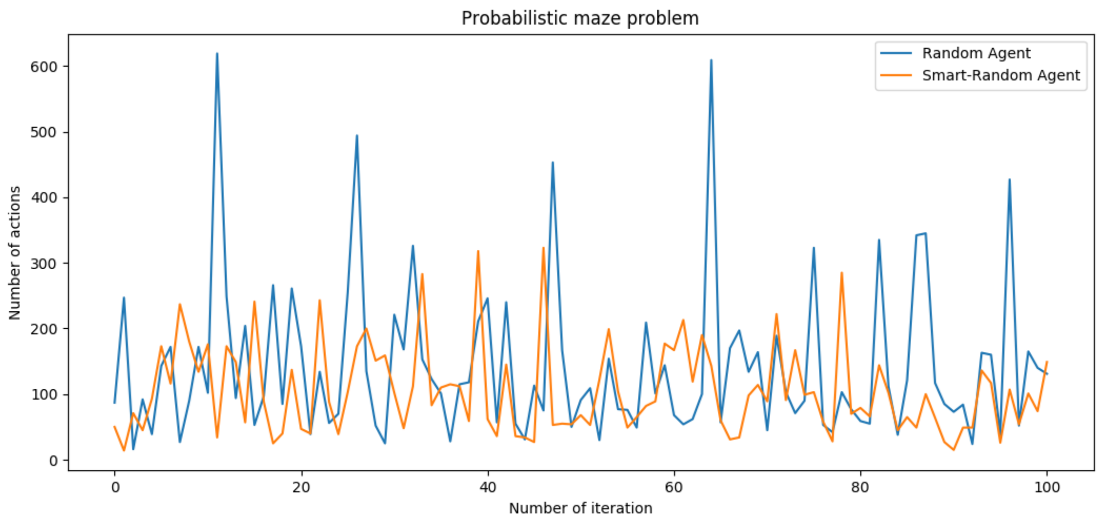
To illustrate this, I performed a full run of the entire learning process (as described in section 3.5) and chose to present the results of the run on an easy problem[6]:



5.2 Avoid the past

The decision to use a random agent based on *avoid the past* principle stemmed from a number of situations in which the agent found himself in an infinite. As a result, the agent took unnecessary actions that led to slower-scale solutions.

In order to illustrate the difference, we ran the maze problem[7] in its probabilistic model using two types of agents: a regular random agent and a "smart" random agent (based on avoid the past). It can be clearly seen that the number of actions that a "smart" random agent is required to perform in order to solve the problem is lower than the number of actions that a regular random agent performs:



[6] Domain name: $maze\_domain\_multi\_effect\_food.pddl$ * Problem name: $t\_5\_5\_5\_food.pddl$
[7] Domain name: $maze\_domain\_multi\_effect\_food.pddl$ * Problem name: $t\_5\_5\_5\_food.pddl$

6 Summary

6.1 Additional attempts

Throughout the agent writing process, I have used additional techniques that I have chosen not to leave in the lead agent. I will expand on them now:

- Normalize learning values:
  When the agent is in the learning phase, he uses parallel running to gain as much study time as possible. In each "learning iteration", it defines α new processes ($= \alpha$ new agents) each of them goes out to learn the task. At the end of each "study iteration" (as described in 3.5), there will be an agent who has managed to solve the problem faster and better than the other agents. Hence, I wanted to perform a normalization for the rest of the agents according to the best agent. The normalization process is calculated as follows:

```
1.    For each learning iteration:
      1.1.  Save the values of each agent (total action & failures & total run-time)
      1.2.  x ← best agent
      1.3.  For each agent (except x), do:
            1.3.1.    factor ← $\frac{\#x\ actions - \#x\ failures}{\#total\ actions - \#total\ failures}$
            1.3.2.    define agent new learning value ←
                        $\left((factor \cdot agent\ current\ values) + (1 - factor \cdot x\ values\right)$
```

  The results did improve over time, but they were inconsistent: there were runs where it led to significant damage to run time and solution quality, so I eventually decided to omit it from the agent's final solution[8].
- Initial values:
  An important and main issue in RL is the improvement of the learning process. One of the options for improving learning processes, is to define the initial values that the agent will use in a smart way. In the Q-Learning algorithm for example, it gets even more meaningful (as a "free model" based). I have tried several approaches regarding initializing learning values:
  - Initialize a table with arbitrary values
  - Initialize a table with arbitrary values and normalized the table
  - Draw an arbitrary value, initialize the entire table with this value

  In none of these forms have I seen an improvement over the "classical" method of initializing a zero table.

6.2 Conclusions and summary

I can say that I am satisfied with the results that my agent presents: he knows how to approach new problems, learn, and deduce information from them efficiently and with quality, and bring this to the fore in the future execution process if he is asked.

[8] The functions can be seen in the end of *BasicAgent.py* file.

Also, we can see a significant improvement in the agent's abilities compared to the agent I built in Ex.4 – where we were required to implement an agent to solve similar problems. The improvement is due, among other things, to the combination of new RL techniques, the construction of a high-quality reward function, the running of a number of agents in parallel, the use of multithreading principles, the integration of RL algorithms and the strict attempt of the agent to solve the problem.

I believe that if I had decided to change the planner to know how to respond to probabilistic problems, the results would have been significantly better. However, my goal among other things was to experiment and specialize in the RL world, to read academic articles and bring them into practice into code – things that as a bachelor's student I did not think I would do.

6.3 References

Throughout the writing of the work, I used the following sources of information:

- ***Double Q-Learning:*** https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf
- ***R-Max Algorithm:*** https://www.jmlr.org/papers/volume3/brafman02a/brafman02a.pdf
- ***Comparing immediate Reward:*** https://arxiv.org/ftp/arxiv/papers/1009/1009.2566.pdf
- ***Reward function and initial values:*** https://hal.archives-ouvertes.fr/hal-00331752v2/document
- ***Algorithms for determining tournament payout structure:*** https://www.chrismusco.com/dfsPresentation.pdf