

Computer Architecture and Technology Area  
Universidad Carlos III de Madrid



## Artificial Intelligence

FINAL PROJECT ASSIGNMENT PROBABILISTIC MODELING FOR THE  
PAC-MAN GAME AND OTHER SCENARIOS

BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING  
Year 2017

Skúli Arnarsson – 100369706

Jonas Ehn – 100360759

Group 89

## Table of Contents

1	Introduction.....	1
2	Development.....	1
2.1	Methods Implemented.....	1
2.1.1	ValueIterationAgents.py .....	1
	computeQValueFromValues.....	1
	computeActionFromValues .....	2
	doValueIteration .....	2
2.1.2	pacmanMDP.py.....	2
	Init.....	2
	updateTransitionFunction .....	2
	getTransitionStatesAndProbabilities.....	3
3	Test Evaluation .....	4
3.1	Section 1.....	4
3.2	Section 2.....	8
3.2.1	Transition Table with Ghost .....	13
3.3	Section 3.....	14
4	Conclusions.....	18
5	Personal Comments .....	19

# 1 Introduction

The purpose of this project was to familiarize ourselves with reasoning techniques under uncertainty using Markov decision processes. Given a modified version of Pac-Man we implemented the missing functions crucial for the correct functionality of the decision process. We then tested those implemented functions with the given autograder to confirm that our implementation works as expected. We then implemented the transition function and observed as the Pac-Man managed to learn how to win the games based on calculated values from previous training sessions. Finally, we saved the calculated result from the training sessions to a file so that the Pac-Man could learn from previous experiences and then adjusted the necessary features for the most successful Pac-Man agent with respect to time complexity.

## 2 Development

The development of the project has closely followed the outline provided in the instructions for the project. The task was to implement and write a number of methods. These methods are described in the section below.

### 2.1 Methods Implemented

In order to finish the project, several methods were implemented with the help of the project description. Below follows a short description of each of the methods.

#### 2.1.1 ValueIterationAgents.py

##### computeQValueFromValues

The function `computeQValuesFromValues` computes the Q-value of a combination of a state and an action using the following formula:

$$V^t(s) = \arg \max_a Q(s, a) \quad (1.)$$

$$Q(s, a) = \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V^{t-1}(s')] \quad (2.)$$

where R is the reward for reaching the next state from the current state by applying an action, P is the probability of ending up in a certain state by applying an action. The parameter gamma is called discount factor and describes how much future rewards are worth compared to immediate rewards. A low gamma gives preference to immediate rewards.

The function works by calculating the sum in equation (2.), over all the possible future states from a given state by taking a certain action.

### **computeActionFromValues**

The function `computeActionFromValues` is called for a certain state, and it returns the action that has the highest Q-value of all the allowed actions in said state. First it checks if the state is a terminal state, if it is a terminal state, no action is returned since no actions are allowed nor needed in a final state.

From the given state we find all possible actions that can be executed in that state. For all those actions we calculate their Q-values. The function then returns that action that corresponds to the highest Q-value.

### **doValueIteration**

The `doValueIteration` function performs value iteration for the number of iterations it is given.

If a state is terminal the reward is 0. If it is not, we check the possible actions for that state. For all the possible actions, we check the Q-value and store the highest Q-value. That is, we get the highest possible Q-value for a given state, by taking some legal action. Then we repeat this for all possible states.

At this point we have found the highest Q-value for each state and we repeat this for the number of given iterations. This iteration makes the value for each state converge to the actual V-value for the state. The V-value of the state thus corresponds to the Q-value of the optimal action in each state.

Delta is the largest difference between two calculated in the last iteration.

## 2.1.2 `pacmanMDP.py`

### **Init**

First we initialize the transition table to an empty dictionary. Then we make a call to `getTransitionTable` as per instruction by the teacher to make it compatible with the reading of a transition table from a file.

### **updateTransitionFunction**

Given an initial state, an action and a next state we first check if there exists an entry in the transition table with the given state and the given action.

If not, this means we have not been in this state and taken action before. Therefore, we must update the transition table by making a new dictionary that keeps track of the frequency of entering a certain “next” state by making an action from the original state. A new entry is then added in the sub-dictionary with the “next” state and the frequency 1.

In the case that the entry does exist in the transition table with the given state and the given action we first need to check if we have previously obtained the same next state by performing said action in the given state. If the result is new, we have to add an entry with this next state to the sub-dictionary and assign the next state with the frequency 1.

If we have seen the next state before by being in the same state and performing the same action, we only increase the frequency by 1.

### **getTransitionStatesAndProbabilities**

The function receives a state and an action. We begin by checking if we have any history of doing this action in this state, that is, if there exists an entry in the transition table with the given state and action as key. If it doesn't, the state have no successors and we return an empty list.

If the entry does exist, we sum the frequency of all the observed next states by performing said action in said state. That means that we get the number of times we've been in the state given to the function and performed the action given.

The sub-dictionary in the transition table that corresponds to this combination of action and state, contains the frequency of each next state that the combination of state and action gives. We make a copy of this list of tuples with the next states and the frequencies. We divide the frequency with the total number of times that we've seen this combination of state and action. This gives a list with the tuples of next states and the ratio of the number of times that we've been in this combination of state and action and ended up in this next state.

## 3 Test Evaluation

### 3.1 Section 1

Testing the `computeQValueFromValues`, `computeActionFromValues` and `doValueIteration` functions in the `ValueIterationAgents.py` file.

We tested our implementation by Using the default settings of the GridWorld scenario (BookGrid) by executing the following given command:

```
python gridworld.py -a value -i <iterations> -k <executions>
```

`i` represents the number of iterations to compute the policy and `k` is the number of executions performed by the agent applying that policy. Assuming that Figure 1 in the assignment description shows the expected V-value, we execute the command stated above with 5 iterations and 5 executions. As can be seen in Figure 1 the computed V-values after 5 iterations are the same as stated in Figure 1 of the assignment description.

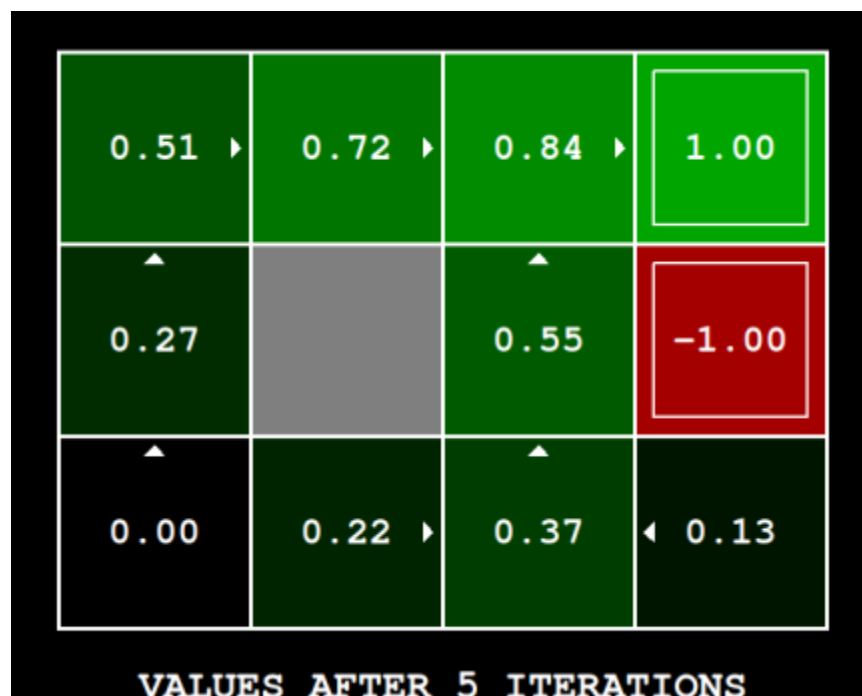


Figure 1: The V-values after 5 iterations

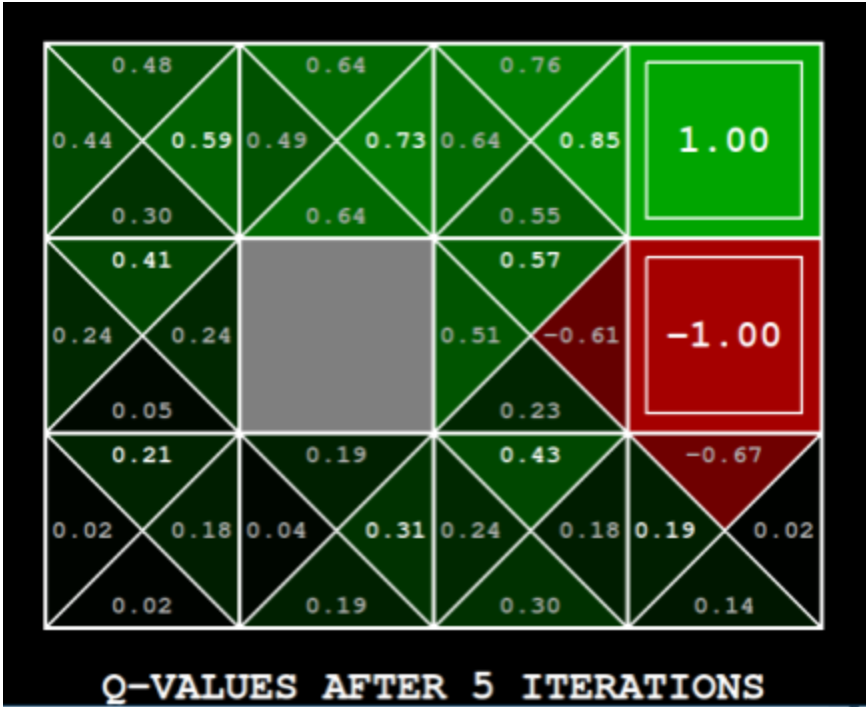


Figure 2: The  $Q$ -values of every action in each state

The output to the terminal is as follows:

```
Iterations: 5
Discount: 0.9
Number of states considered: 12
Last Delta between iterations: 0.26873856
---Omitted state transitions and return value from each execution.
AVERAGE RETURNS FROM START STATE: 0.53262198
```

To test the correctness of our solution we execute the given autograder with the following command:

```
python autograder.py -q q1
```

The the resulting output from that command is:

```
*** PASS: test_cases/q1/4-discountgrid.test
```

### Question q1: 6/6 ###  
Finished at 11:19:33

## Provisional grades

=====

Question q1: 6/6

Total: 6/6

\*\*\*

Testing of our implementation using the BridgeGrid scenario. The goal is to move the agent to the right-hand side without falling into the water. We manage this by executing the command with the noise parameter set to zero.

```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9
--noise 0.0
```

In Figure 3, we can see the V-value for each cell. In Figure 4, we can see the V-value for each action, moving from each cell.



Figure 3: The V-values for each cell in the bridge scenario



Figure 4: The Q-values for each action for each state in the bridge scenario



The output to the terminal is as follows:

```
Iterations: 100
Discount: 0.9
Number of states considered: 18
Last Delta between iterations: 0.0

--State transitions omitted.
Ended in state: TERMINAL_STATE
Got reward: 10
EPISODE 1 COMPLETE: RETURN WAS 5.9049
AVERAGE RETURNS FROM START STATE: 5.9049
```

To confirm that our implementation for question 2 of the first part is correct we test it using the autograder by executing the following command.

```
python autograder.py -q q2
```

The following is the output to the terminal after running the command:

```
Starting on 5-9 at 4:06:58
Question q2
=====
Iterations: 100
Discount: 0.9
Number of states considered: 18
Last Delta between iterations: 0.0
*** PASS: test_cases/q2/1-bridge-grid.test

### Question q2: 1/1 ###
Finished at 4:06:58
Provisional grades
=====
Question q2: 1/1
-----
Total: 1/1
```

The next part of section 1 involves changing the values of discount, noise and livingReward so that the agent reaches the goal by means of different behaviour. The behaviours the agent should express are:

- Reach the exit (+1) choosing the less safe but shorter path (-10)
- Reach the exit (+1) choosing the safer and longer path (-10)
- Reach the exit (+10) choosing the less safe but shorter path (-10)
- Reach the exit (+10) choosing the safer and longer path (-10)
- Avoiding both exits and the cliff (this option should never end)

For executing the behaviours we set the value of the variables for each scenario as follows:

- Discount = 0.1, Noise = 0.0, LivingReward = 0.0
- Discount = 0.3, Noise = 0.3, LivingReward = 0.2
- Discount = 0.9, Noise = 0.0, LivingReward = 0.0

d) Discount = 0.9, Noise = 0.2, LivingReward = 0.0

e) Discount = 0.0, Noise = 0.0, LivingReward = 1

To test if we manage to express these behaviours with the value of our variables we execute the autograder with the following command:

```
python autograder.py -q q3
```

The output of the autograder to the terminal is as follows:


```
Starting on 5-9 at 4:18:00
Question q3
=====
Iterations: 100
Discount: 0.1
Number of states considered: 23
Last Delta between iterations: 0.0
*** PASS: test_cases/q3/1-question-3.1.test
Iterations: 100
Discount: 0.3
Number of states considered: 23
Last Delta between iterations: 0.0
*** PASS: test_cases/q3/2-question-3.2.test
Iterations: 100
Discount: 0.9
Number of states considered: 23
Last Delta between iterations: 0.0
*** PASS: test_cases/q3/3-question-3.3.test
Iterations: 100
Discount: 0.9
Number of states considered: 23
Last Delta between iterations: 3.33510996597e-13
*** PASS: test_cases/q3/4-question-3.4.test
Iterations: 100
Discount: 0.0
Number of states considered: 23
Last Delta between iterations: 0.0
*** PASS: test_cases/q3/5-question-3.5.test

### Question q3: 5/5 ###

Finished at 4:18:01
Provisional grades
=====
Question q3: 5/5
-----
Total: 5/5
```

## 3.2 Section 2

Running the training is tedious and, especially for the bigger layouts, a lot of training sessions have to be performed. Pacman is well-behaved if it has seen all the possible states. However, there exists no easy way to see if all the possible states have been explored during the training. It becomes obvious sometime to see when Pacman comes to a state that it hasn't explored, since the behaviour becomes random, but there is no way to see if a situation like that can be



avoided. As far as we have seen, the only way to avoid erratic behaviour of the Pacman is to perform a very large number of training sessions.

To test our implementation for section 2 we run training sessions over a simple scenario so that the agent learns the transition function. Then we run a number of games to observe the behaviour of the agent and estimate how well the agent performs

```
python pacman.py -p EstimatePacmanMdpAgent -x <train> -n <total>
-k <ghosts> -l <scenario> -a "iterations=<number of
iterations>,discount=<discount - factor>"
```

We begin testing by executing the above command with 50 training sessions, 55 total games, 0 ghosts and in the superSmallGrid scenario by executing the following command.

```
python pacman.py -p EstimatePacmanMdpAgent -x 50 -n 55 -k 0 -l
superSmallGrid -a "iterations=50,discount=0.8"
```

The output to the terminal is as follows:

```

Beginning 50 episodes of Training
Training finished
Training Done (turning off epsilon and alpha)
-----
Executing Value Iteration
Iterations: 50
Discount: 0.8

Executing MDP with transition table:
(1, 1, 7, 0) Stop [((1, 1, 7, 0), 1.0)]
(1, 1, 7, 0) East [((2, 1, 6, 0), 1.0)]
(2, 1, 6, 0) West [((1, 1, 7, 0), 1.0)]
(2, 1, 6, 0) Stop [((2, 1, 6, 0), 1.0)]
(2, 1, 6, 0) East [((3, 1, 5, 0), 1.0)]
(3, 1, 5, 0) West [((2, 1, 6, 0), 1.0)]
(3, 1, 5, 0) Stop [((3, 1, 5, 0), 1.0)]
(3, 1, 5, 0) East [((4, 1, 4, 0), 1.0)]
(4, 1, 4, 0) West [((3, 1, 5, 0), 1.0)]
(4, 1, 4, 0) Stop [((4, 1, 4, 0), 1.0)]
(4, 1, 4, 0) East [((5, 1, 3, 0), 1.0)]
(5, 1, 3, 0) West [((4, 1, 4, 0), 1.0)]
(5, 1, 3, 0) Stop [((5, 1, 3, 0), 1.0)]
(5, 1, 3, 0) East [((6, 1, 2, 0), 1.0)]
(6, 1, 2, 0) West [((5, 1, 3, 0), 1.0)]
(6, 1, 2, 0) Stop [((6, 1, 2, 0), 1.0)]
(6, 1, 2, 0) East [((7, 1, 1, 0), 1.0)]
(7, 1, 1, 0) West [((6, 1, 2, 0), 1.0)]
(7, 1, 1, 0) Stop [((7, 1, 1, 0), 1.0)]
(7, 1, 1, 0) East [((8, 1, None, None), 1.0)]
End of MDP transition table

Number of states considered: 8
Last Delta between iterations: 0.0

Pacman emerges victorious! Score: 507
Training finished
Average Score: 507.0
Scores:      507.0, 507.0, 507.0, 507.0, 507.0
Win Rate:    5/5 (1.00)
Record:      Win, Win, Win, Win, Win

```

From these results we may infer that the agent has indeed learned to win from the 50 training sessions in a reasonable time, given the case that the scenario is the superSmallGrid and that no ghosts exist on the map. We should then try to execute the same command on a moderately larger grid. To do so we execute the following command:

```
python pacman.py -p EstimatePacmanMdpAgent -x 50 -n 55 -k 0 -l
smallGrid2 -a "iterations=50,discount=0.8"
```

The output to the terminal is as follows:

```

Beginning 50 episodes of Training
Training finished
Training Done (turning off epsilon and alpha)
-----
Executing Value Iteration
Iterations: 50
Discount: 0.8

```

--Transition table is omitted here, it was moderately larger than the one from the previous test.

```

Number of states considered: 18
Last Delta between iterations: 0.0
-----
Average Score: 503.0
Scores:      503.0, 503.0, 503.0, 503.0, 503.0
Win Rate:    5/5 (1.00)
Record:      Win, Win, Win, Win, Win

```

From these results we can infer that the agent is also able to win the larger map in a reasonable amount of time given a small map with no ghosts on it. We should then add the variable of a ghost to our testing. To do so we could execute the following command:

```
python pacman.py -p EstimatePacmanMdpAgent -x 50 -n 55 -k 1 -l
smallGrid2 -a "iterations=50,discount=0.8"
```

The output to the terminal from that command, omitting the transition table, is as follows:

```

Beginning 50 episodes of Training
Training finished
Training Done (turning off epsilon and alpha)
-----
Executing Value Iteration
Iterations: 50
Discount: 0.8

Number of states considered: 142
Last Delta between iterations: 2.99225219325e-08
-----
Pacman died! Score: -524
Training finished
Pacman died! Score: -524
Training finished
Pacman died! Score: -508
Training finished
Pacman died! Score: -524
Training finished
Pacman died! Score: -508
Training finished
Average Score: -517.6
Scores:      -524.0, -524.0, -508.0, -524.0, -508.0
Win Rate:    0/5 (0.00)
Record:      Loss, Loss, Loss, Loss, Loss

```

These results indicate that the agent does not have enough knowledge to win the game when a ghost is involved. When the ghost variable is introduced the number of states increases by a lot.

In the previous execution the number of states considered was 142. We should then run at least the same amount of training sessions so that the agent is able to win the game. We run the same scenario as before but with 500 training sessions and 50 executions to observe, so that the sample space is larger. The output, with the transition table omitted is as follows:

```
Beginning 500 episodes of Training
Learning Status:
    Completed 100 out of 500 training episodes
    Average Rewards over all training: -500.39
    Average Rewards for last 100 episodes: -500.39
    Episode took 0.80 seconds
Learning Status:
    Completed 200 out of 500 training episodes
    Average Rewards over all training: -500.05
    Average Rewards for last 100 episodes: -499.71
    Episode took 0.75 seconds
Learning Status:
    Completed 300 out of 500 training episodes
    Average Rewards over all training: -503.43
    Average Rewards for last 100 episodes: -510.18
    Episode took 0.80 seconds
Learning Status:
    Completed 400 out of 500 training episodes
    Average Rewards over all training: -505.13
    Average Rewards for last 100 episodes: -510.25
    Episode took 0.79 seconds
Training finished
Learning Status:
    Completed 500 out of 500 training episodes
    Average Rewards over all training: -503.98
    Average Rewards for last 100 episodes: -499.39
    Episode took 0.78 seconds
Training Done (turning off epsilon and alpha)
-----
Executing Value Iteration
Iterations: 50
Discount: 0.8
```

```
Number of states considered: 271
Last Delta between iterations: 1.16058639783e-05
```

```

Training finished
Average Score: 112.12
Scores:      -519.0, -519.0, 501.0, 485.0, 501.0, -515.0,
-519.0, 501.0, 501.0, 501.0, -519.0, -515.0, 501.0, 501.0,
501.0, -515.0, -515.0, 485.0, -515.0, 501.0, 501.0, 501.0,
485.0, 501.0, -515.0, 485.0, -515.0, 501.0, -519.0, 501.0,
-515.0, -515.0, 501.0, 501.0, -519.0, -515.0, 501.0, -515.
0, 485.0, 501.0, 501.0, 485.0, 485.0, 501.0, 501.0, -519.0,
501.0, 501.0, -515.0, 501.0
Win Rate:    31/50 (0.62)
Record:      Loss, Loss, Win, Win, Win, Loss, Loss, Win,
Win, Win, Loss, Loss, Win, Win, Win, Loss, Loss, Win, Loss,
Win, Win, Win, Win, Win, Loss, Win, Loss, Win, Loss, Win,
Loss, Loss, Win, Win, Loss, Loss, Win, Loss, Win, Win, Win,
Win, Win, Win, Win, Loss, Win, Win, Loss, Win

```

These results confirm our assumptions that the number of training sessions in the previous test were not enough for the introduction of the ghost variable. The number of states considered for this example is 271 compared to 142 in the previous one. This is because of the lack of training in the previous test. With more training sessions more states are encountered.

### 3.2.1 Transition Table with Ghost

The manual features an example of the transition table when the command

```

Python pacman.py -p EstimatePacmanMdoAgent -x 50 -n 55 -k 1 -l
smallGrid2 -a "iterations=50,discount=0.8"

```

is run. Executing this command gives the following output:

```

(2, 5, 1, -2, -1, -4) West [((1, 5, 2, -2, 1, -4), 0.375), ((1, 5, 2, -2, 0, -3), 0.625)]
(2, 5, 1, -2, -1, -4) Stop [((2, 5, 1, -2, -1, -3), 0.4782608695652174), ((2, 5, 1, -2, 0, -4), 0.5217391304347826)]
(2, 5, 1, -2, -1, -4) East [((3, 5, 0, -2, -2, -3), 0.25), ((3, 5, 0, -2, -1, -4), 0.75)]

```

This part of the transition table can be compared with the one in the lab manual. Which looks like this:

```

(2, 5, 1, -2, -1, -4) West [((1, 5, 2, -2, 1, -4), 0.466), ((1, 5, 2, -2, 0, -3), 0.533)]
(2, 5, 1, -2, -1, -4) Stop [((2, 5, 1, -2, -1, -3), 0.555), ((2, 5, 1, -2, 0, -4), 0.444)]
(2, 5, 1, -2, -1, -4) East [((3, 5, 0, -2, -2, -3), 0.411), ((3, 5, 0, -2, -1, -4), 0.588)]

```

The layout of the two tables are comparable, both gives two next states from each combination of state and action. The difference between the next states possible for each combination of state and action is in the position of the ghost relative to Pacman. In each time step the ghost can move either left or right, because of the layout. Since this movement of the ghost is random and independent from the actions of Pacman, each combination of state and pacman-action can result in two different next states, depending on what the ghost does.

However, the probabilities of the next states are different in the two tables. The reason for that is that the “probabilities” are not real probabilities but approximations. What they actually are is the number of times the system has seen the combination of state and action resulting in that next state divided by the times it has seen the specific combination of state and action. Since the movement of the ghost is totally random, this frequency observed could be any value. But since we know that there are two possible movements for the ghost, the approximation of the

probability should go towards a half for an infinite number of training sessions, if the seed for the randomness is perfect. The difference between the numbers is therefore that 50 is a rather rough approximation of infinity.

### 3.3 Section 3

In this section we initiate the game like in section 2, whereas we run the game with the same command but add an extra parameter *table* that allows us to use an external file to save the solution of a set of training rounds.

Let's test if the file is created when the command is following command is executed by running the following command:

```
python pacman.py -p EstimatePacmanMdpAgent -x 50 -n 55 -k 1 -l
smallGrid2 -a"iterations=50,discout=0.8,table=smallGrid2.p"
```

The file smallGrid2.p was successfully created and is 325 KB in size. The results from the game after running the command for the first time is as follows:

```
Number of states considered: 159
Last Delta between iterations: 6.56289947454e-05
-----
MDP transition table saved to file smallGrid2.p
Pacman died! Score: -512
Training finished
Pacman emerges victorious! Score: 488
Training finished
Pacman died! Score: -508
Training finished
Pacman died! Score: -508
Training finished
Pacman died! Score: -508
Training finished
Average Score: -309.6
Scores: -512.0, 488.0, -508.0, -508.0, -508.0
Win Rate: 1/5 (0.20)
```

The Pac-Man does seem to need more than 50 training sessions to be able to win the game more reliably. We try to run the same command again with the same table file and watch as the file size should increase and the number of games won should increase as more states will be explored and added to the file.

After running the command a second time, the size of the file has increased from 325 KB to 420 KB. We can assume that the Pac-Man is exploring more states and adding them to the table file causing it to grow in size. The result from running the command the second file is as follows:



```

Number of states considered: 198
Last Delta between iterations: 2.311653776e-05
-----
MDP transition table saved to file smallGrid2.p
Pacman died! Score: -510
Training finished
Pacman died! Score: -528
Training finished
Pacman emerges victorious! Score: 485
Training finished
Pacman emerges victorious! Score: 499
Training finished
Pacman emerges victorious! Score: 499
Training finished
Average Score: 89.0
Scores: -510.0, -528.0, 485.0, 499.0, 499.0
Win Rate: 3/5 (0.60)
Record: Loss, Loss, Win, Win, Win

```

We can see that the number of states considered has increased from 159 to 198 and the win rate from 0.20 to 0.60.

After running the same command the sixth time we were able to get the Pac-Man to win every game, with these results:

```

Number of states considered: 240
Last Delta between iterations: 1.7082052131e-05
-----
MDP transition table saved to file smallGrid2.p
Pacman emerges victorious! Score: 499
Training finished
Pacman emerges victorious! Score: 499
Training finished
Pacman emerges victorious! Score: 503
Training finished
Pacman emerges victorious! Score: 503
Training finished
Pacman emerges victorious! Score: 503
Training finished
Average Score: 501.4
Scores: 499.0, 499.0, 503.0, 503.0, 503.0
Win Rate: 5/5 (1.00)
Record: Win, Win, Win, Win, Win

```

The size of the table file at this point is 553 KB after 300 training sessions. For a grid as small as this one, 300 training sessions is enough to get a relatively smart Pac-Man agent. Changing the position of the Pac-Man and the food for the grid smallGrid2 does not affect the win-rate of the games. We change to a different grid using the same table file as before to test if the results that he has learned for smallGrid2 are applicable to other scenarios. We change to the smallGrid scenario. The table file has grown to 1920 KB at this moment, the results are as follows:

```

Number of states considered: 417
Last Delta between iterations: 2.0875393858e-07
-----
MDP transition table saved to file smallGrid2.p
Pacman died! Score: -511
Training finished
Pacman died! Score: -511
Training finished
Pacman died! Score: -514
Training finished
Pacman died! Score: -501
Training finished
Pacman died! Score: -515
Training finished
Pacman died! Score: -535
Training finished
Pacman died! Score: -502
Training finished
Pacman died! Score: -505
Training finished
Pacman died! Score: -502
Training finished
Pacman died! Score: -508
Training finished
Average Score: -510.4
Scores:      -511.0, -511.0, -514.0, -501.0, -515.0, -535.0, -502.0, -505.0, -502.0, -508.0
Win Rate:    0/10 (0.00)
Record:      Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss

```

The size of the table file at this point is 55.1 KB. This is small as this one 300 training sessions is a small sample. Changing the position of the Pac-Man and the win-rate of the games. We change to a new scenario to test if the results that he has learned for the smallGrid scenario. We change to the smallGrid scenario. The results are as follows:

Possible tests:

- Computing a policy in a particular scenario
- Computing a policy in a particular scenario with a particular number of ghost(s)
- Computing a policy with a particular number of training sessions

From these results we see that new states become available when a new scenario is introduced and the Pac-Man is not trained enough for these new scenarios.

Let's test if the Pac-Man can train for a larger grid and be able to learn how to win a moderate amount of games within a reasonable time. We try a new table file with a new grid testClassic and execute the following command.

```
python pacman.py -p EstimatePacmanMdpAgent -x 1000 -n 1100 -k 1 -l testClassic -a "iterations=50,discount=0.8,table=testClassic.p" -q
```

The results are as follows:

```

Training finished
Learning Status: Completed
Completed 100 test episodes
Average Rewards over testing: -491.18
Average Rewards for last 100 episodes: -491.18
Episode took 327.80 seconds
Average Score: -491.18
Scores:      -518.0, -503.0, -469.0, -508.0, -495.0, -470.0, -525.0, -487.0, -501.0, -508.0, -489.0, -525.0, -496.0, -485.0, -483.0, -467.0, -479.0, -509.0, -499.0, -513.0, -497.0, -665.0, -477.0, -494.0, -552.0, -471.0, -544.0, -493.0, -568.0, -465.0, -475.0, -533.0, -572.0, -496.0, -482.0, -508.0, -491.0, -486.0, -490.0, -475.0, -481.0, -509.0, -469.0, -493.0, -493.0, -476.0, -473.0, -466.0, -502.0, -479.0, -477.0, -473.0, -472.0, -488.0, -487.0, -463.0, -539.0, -499.0, -466.0, -477.0, -469.0, -488.0, -474.0, -468.0, -461.0, -482.0, -459.0, -468.0, -477.0, -492.0, -494.0, -484.0, -473.0, -494.0, -467.0, -522.0, -483.0, -477.0, -473.0, -485.0, -500.0, -471.0, -487.0, -521.0, -465.0, -489.0, -458.0, -460.0, -476.0, -474.0, -490.0, -461.0, -518.0, -474.0, -480.0, -487.0, -472.0, -499.0, -481.0, -520.0
Win Rate:    0/100 (0.00)

```

We can then assume that for a larger grid the Pac-Man agent needs to have a far greater amount of training sessions to explore all the possible states.

Even running over 20.000 training sessions over the same file, that is 18.837 KB and having explored 4299 many states yields a 0.00 win rate for 100 samples. In order to be able to win the game for a grid like testClassic we may try to modify the simplified (high-level) states. The

problem we have is that the number of variables are too many to be able to get the Pac-Man agent to learn how to win within a reasonable timeframe. To solve this problem, we can reduce the number of features in the high-level state. We should then be able to run a larger number of training sessions in less time. To try this, we can change the features used from

```
['posX', 'posY', 'IncFoodX', 'IncFoodY', 'IncGhostX', 'IncGhostY' ]
```

```
to ['posX', 'posY', 'IncFoodX', 'IncFoodY', 'ClosestGhostDist'].
```

Therefore, eliminating one variable. This results in the training sessions taking much less time to execute. This however is not useful whereas after over 10.000 training sessions the Pac-Man is not able to win a single game and his behaviour is such that he ends up staying still in a corner until a ghost approaches and then moves away from it, only to end up again in the same corner and never eating the food unless by chance and then ultimately being killed by the ghost.

```
Learning Status:
Completed 100 test episodes
Average Rewards over testing: -529.92
Average Rewards for last 100 episodes: -529.92
Episode took 157.56 seconds
Average Score: -529.92
Scores: -534.0, -541.0, -501.0, -515.0, -536.0, -507.0, -513.0, -509.0, -520.0, -512.0, -515.0, -501.0, -539.0,
-531.0, -563.0, -499.0, -497.0, -523.0, -504.0, -499.0, -524.0, -564.0, -525.0, -573.0, -598.0, -499.0, -522.0, -521.
0, -520.0, -509.0, -520.0, -527.0, -506.0, -545.0, -535.0, -510.0, -607.0, -632.0, -517.0, -599.0, -515.0, -527.0, -49
7.0, -512.0, -586.0, -531.0, -507.0, -561.0, -523.0, -543.0, -647.0, -541.0, -526.0, -558.0, -503.0, -507.0, -513.0, -
499.0, -503.0, -589.0, -519.0, -517.0, -564.0, -504.0, -569.0, -569.0, -497.0, -503.0, -532.0, -511.0, -562.0, -501.0,
-509.0, -510.0, -523.0, -579.0, -539.0, -513.0, -508.0, -575.0, -503.0, -515.0, -507.0, -497.0, -545.0, -550.0, -571.
0, -511.0, -504.0, -513.0, -504.0, -524.0, -506.0, -500.0, -527.0, -571.0, -587.0, -507.0, -497.0, -549.0
Win Rate: 0/100 (0.00)
```

## 4 Conclusions

The original list of features

```
['posX', 'posY', 'IncFoodX', 'IncFoodY', 'IncGhostX', 'IncGhostY']
```

is the best one. It takes a long time for it to learn but it wins the highest amount of games after a fewer amount of training sessions compared to when using

```
['posX', 'posY', 'IncFoodX', 'IncFoodY', 'ClosestGhostDist']
```

or

```
['posX', 'posY', 'IncFoodX', 'IncFoodY', 'GhostDist']
```

In the last two cases, the number of states are reduced, which makes the training much faster and more reliable since it is possible to explore all possible states in a reasonable number of time. On the other hand, it doesn't contain enough information of the position of the ghost to be useful. It only knows the distance to the ghost, not the direction. This causes the Pacman to keep a distance as large as possible to the ghosts. This causes it to get stuck on the other side of the board from the ghost and only changes the side when the ghost moves to the other side of the board, the one that Pacman occupies. This means that this simplification can be done if the goal is to stay alive, but not if the goal is to eat all the food.

As the table file grows to a large size as when calculating a decent transition table for a medium-sizes grid. Even executing a single training session takes an unreasonably long time because reading the table file takes considerably longer than if it only contained results for a small grid. We assume that having a single table file for each scenario would yield the best efficiency because it would reduce the size of the file, as it only contains information relevant to the specified scenario. Also as we learned from testing, training a Pac-Man agent in a specific scenario gives little knowledge for another scenario.

After a long time training our Pac-Man agent in multiple scenarios multiple times we conclude the best performance we achieved for a few small scenarios.

For superSmallGrid we were able to get a win rate of 1.00 for 1000 test samples with one ghost on the grid.

For smallGrid we were able to get a win rate of 0.91 for 1000 test samples with one ghost on the grid.

For smallGrid2 we were able to get a win rate of 1.00 for 1000 test samples with one ghost on the grid.

For testClassic we were not able to win a single game within a reasonable timeframe.

## 5 Personal Comments

We both worked on the project on our personal computers that run Windows. However, we got it working on a virtual machine and used it to develop our solution. However, using a virtual machine to do the training turned out to be extremely slow, since the training require significant performance for large training sessions, sessions with a huge number of possible states. We therefore strongly recommend another group making the same project to use Linux running directly on the hardware.

There was sometimes hard to understand exactly what was expected of us in each part of the project. It took a lot of discussion within the project group, and usually with the teacher as well, to understand what had to be done. The classes therefore became necessary in order to understand the task, for which it was good that there were many classes assigned to the project. However, the project instruction could maybe be updated with the hints that were given during class and emailed to the lab group.

We had difficulties running the Pac-Man game with the graphical animations, our virtual machine crashed every time the Pac-Man had to turn. We solved the problem by executing the game without the graphical animations and using the text representation instead by adding `-t` at the end of a command executing the Pac-Man game.

We learned how to implement value iteration, computing actions from values and calculating q-values in code. We have a better understanding of the functionality of the Markov decision process and how it is used in practice. We learned how the different values for discount, noise and living reward influence the behaviour of the agent. We observed directly how the time complexity of the program grew exponentially with a greater number of variables. This became really obvious in the third part, when the training sessions were written to a file and it was possible to see how the file grew with the number of possible states.

The fact that the project is based on a large given code was something that was new to both of us. It was hard to understand to know how to implement the functions correctly, since it was hard to see how they interact with the rest of the code. However, we do understand that the possibilities of doing a project like this a limited given the time space for it.