Q-Learning Project Report

Robert Sato

March 2021

Abstract

This project explores Q-learning, a reinforcement learning algorithm for model free environment exploration. Q-learning is applied to a problem of finding optimal paths from any location to a single destination. This project implements four version of Q-learning: Python, Java, C, and C++. The differences, similarities, and performance of all four are considered.

1 Introduction

The motivation for this project was to gain a deeper understanding of the Q-learning algorithm and to experiment with multiple programming languages. Completion of an implementation in another language involved creating a program that would read in the environment generated in a CSV file, train a Q-table with the input hyper-parameters, and output the correct path and path cost. All four implementations are capable of this with varying levels of performance. Some implementations required numerous helper functions and abstract data types to keep the code clear and concise. Others relied on many built-in functionalities of the language while carrying lots of baggage that slowed down performance.

2 Problem Description

The problem our Q-learning algorithm attempts to solve is the "Robots in a Warehouse" problem. In this problem, we want to train a robot to bring objects from within a warehouse to a particular packaging location.

2.1 Reinforcement Learning Environment

The warehouse consists of three states. The green square is the packaging location and can be thought of as a goal state. The black squares are item storage locations. Because we do not want the robot crashing around making a mess, these are terminal states. The white squares are aisles where the robot can move.

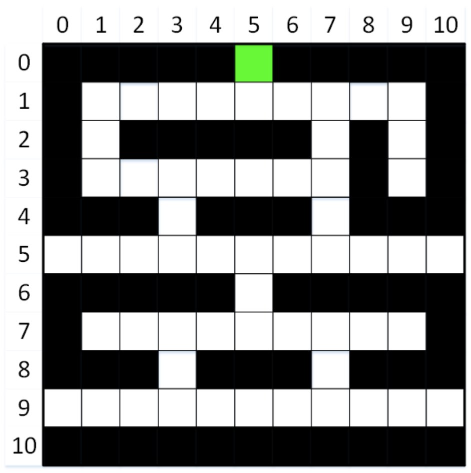
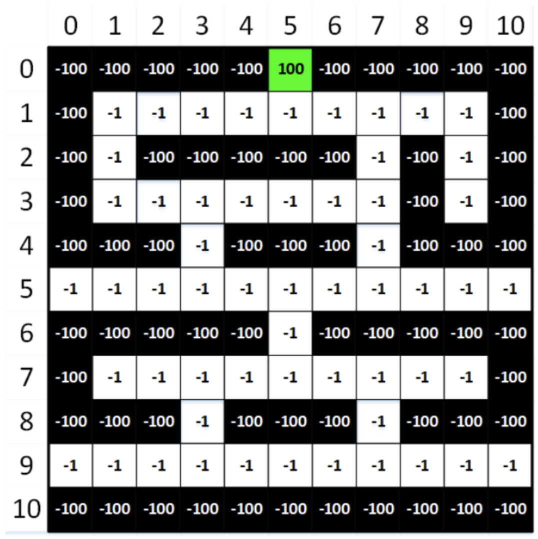


Figure 1: Environment states

2.2 Rewards and Actions

The goal state has a reward (value) of 100. Crashing into an item storage location ends the training session and gives a reward of -100. The aisle states have a reward of -1. This is to deter infinite paths; the longer the robot takes to get to the goal state, the more negative reward it collects.



	0	1	2	3	4	5	6	7	8	9	10
0	-100	-100	-100	-100	-100	100	-100	-100	-100	-100	-100
1	-100	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100
2	-100	-1	-100	-100	-100	-100	-100	-1	-100	-1	-100
3	-100	-1	-1	-1	-1	-1	-1	-1	-100	-1	-100
4	-100	-100	-100	-1	-100	-100	-100	-1	-100	-100	-100
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	-100	-100	-100	-100	-100	-1	-100	-100	-100	-100	-100
7	-100	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100
8	-100	-100	-100	-1	-100	-100	-100	-1	-100	-100	-100
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100

Figure 2: Environment with rewards

There are only four actions the robot can take; up, down, left, and right.

These actions are restricted at the edges and corners as the robot can not leave the warehouse.

3 Q-Learning

In order to better understand this project, a short discussion of the Q-learning algorithm is included. Q-learning employs a "Q-table" which saves the expected "Q-value" of a state and action pair. The Q-table is a 3 dimensional array of height and width equal to that of the environment. Each state in the environment receives a dimension for actions in the Q-table. The Q-table saves the Q-value (or expected value) of each action from the given state.

The Q-learning algorithm chooses random start states, takes stochastic walks until it reaches a terminal state, and updates the Q-table as it goes. Once it is done training, we use the Q-table to take the estimated best action from each individual state. The critical step of the Q-learning algorithm is Q-value updates using temporal difference.

$$Q^{new}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t,a_t)}_{\text{old value}}\Bigg)}^{\text{temporal difference}}$$

Figure 3: Temporal difference formula used for Q-learning

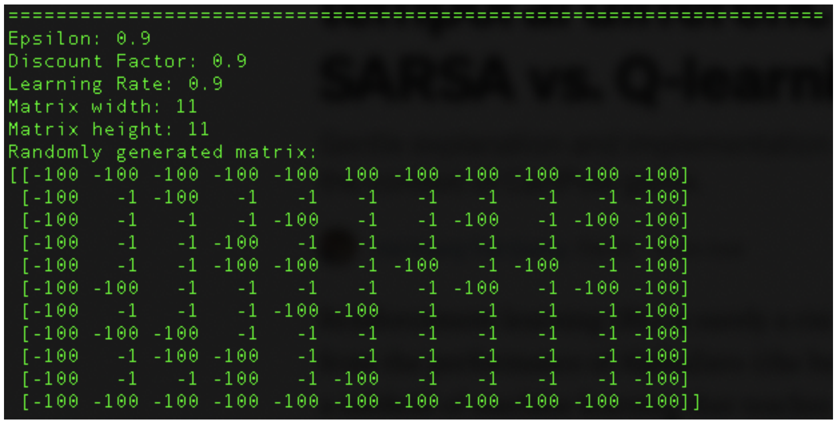
This update is repeated throughout the random walk. Each random walk constitutes one training episode.

4 Implementation

The environment is generated with the generatematrix.py script. This script creates a CSV file that has all of the training hyper-parameters and stores the values of each state in a matrix. This file is then read in by each of the different implementations and trained on.

4.1 Hyper-parameters

There are four hyper-parameters of interest;epsilon, discount factor, learning rate, and training episodes. Epsilon controls how greedy the algorithm is when choosing the next best action. A larger epsilon (closer to 1) means the algorithm is less likely to explore the environment and will only take greedy actions. A small epsilon (closer to 0) ignores what we have learned so far and explores supposedly less optimal actions. Discount factor, as seen in Figure 3, is how much we discount future reward. Because we do not receive future reward immediately, it is discounted in comparison to the reward of the current state. Learning



```
===========================================================================
Epsilon: 0.9
Discount Factor: 0.9
Learning Rate: 0.9
Matrix width: 11
Matrix height: 11
Randomly generated matrix:
[[-100 -100 -100 -100 -100  100 -100 -100 -100 -100 -100]
 [-100   -1 -100   -1   -1   -1   -1   -1   -1   -1 -100]
 [-100   -1   -1   -1 -100   -1   -1 -100   -1 -100 -100]
 [-100   -1   -1 -100   -1   -1   -1   -1   -1   -1 -100]
 [-100   -1   -1 -100 -100   -1 -100   -1 -100   -1 -100]
 [-100 -100   -1   -1   -1   -1   -1 -100   -1 -100 -100]
 [-100   -1   -1   -1 -100 -100   -1   -1   -1   -1 -100]
 [-100 -100 -100   -1   -1   -1   -1   -1   -1   -1 -100]
 [-100   -1 -100 -100   -1   -1   -1   -1   -1   -1 -100]
 [-100   -1   -1 -100   -1 -100   -1   -1   -1   -1 -100]
 [-100 -100 -100 -100 -100 -100 -100 -100 -100 -100 -100]]
```

Figure 4: 11x11 generated values matrix environment

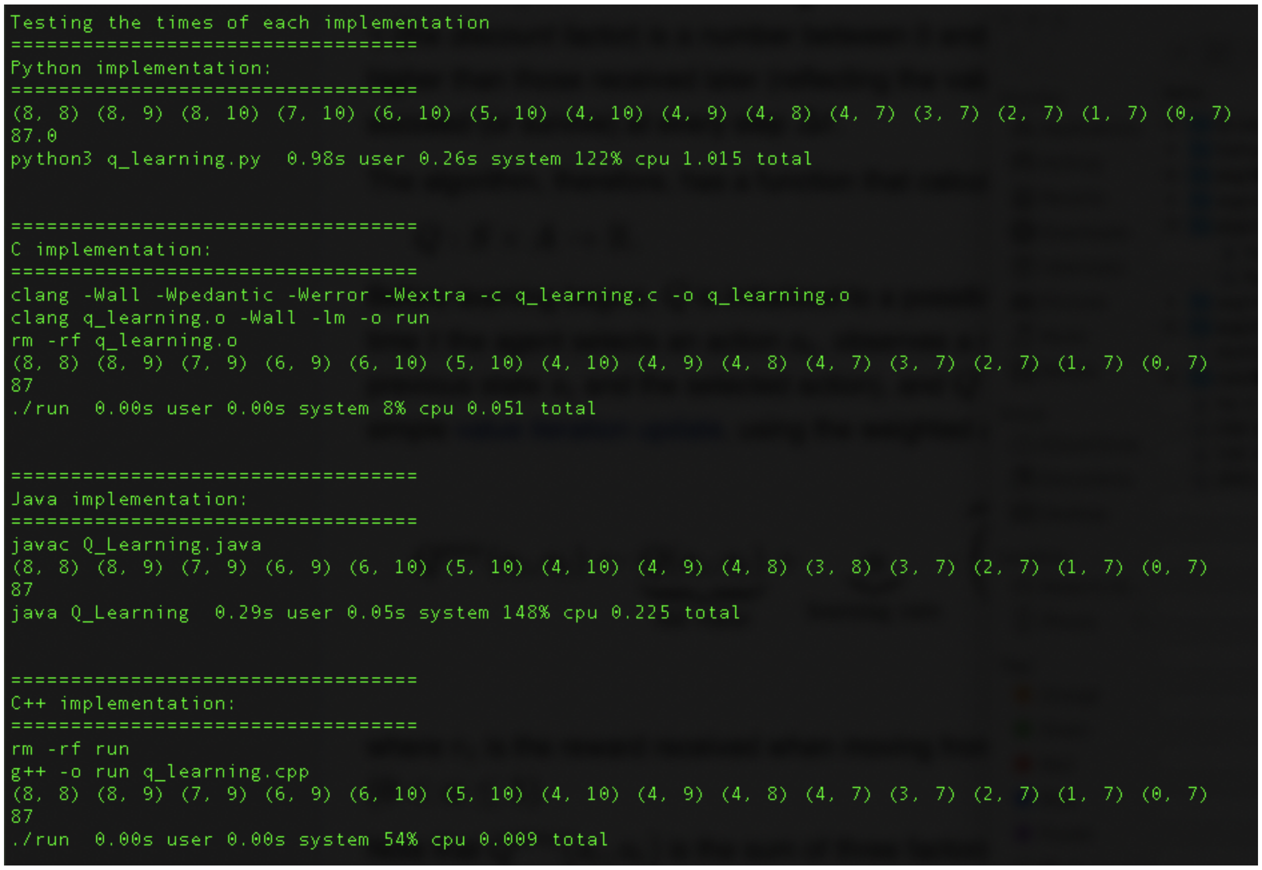
rate, also seen in Figure 3, influences how quickly we update our Q-table. A large learning rate causes large updates. Training episodes, which denote how many times we allow the algorithm to restart and explore the environment, is kept at 1000. For all tests, epsilon = 0.9, discount factor = 0.9, and learning rate = 0.9 are used.

4.2 Implementation Specifics

Numerous helper functions were created for the different implementations. While generic helper functions, such as for getting a new random start state and checking if a state is terminal, were implemented in all four languages, some structures (pairs, lists) had to be implemented in others. C, C++, and Java all required a new structure for storing states.

5 Results

C and C++ had the best overall performance. This performance is measured as total execution time of the executable files. As Python is interpreted, we can see why it has much longer execution times. Execution time also includes the reading of the CSV input. I/O and the Q-table training are the main suspects in taking up execution time. These measurements (Figure 6), were done across varying matrix sizes. The generatematrix.py script generated pseudo-random environments as input and each of the different implementations were run. These were all run sequentially using a bash script. Each implementation was isntructed to find the best path starting from square (8,8). Both the path and the path cost is printed. This is shown in Figure 5.



```
Testing the times of each implementation
==================================
Python implementation:
==================================
(8, 8) (8, 9) (8, 10) (7, 10) (6, 10) (5, 10) (4, 10) (4, 9) (4, 8) (4, 7) (3, 7) (2, 7) (1, 7) (0, 7)
87.0
python3 q_learning.py  0.98s user 0.26s system 122% cpu 1.015 total


==================================
C implementation:
==================================
clang -Wall -Wpedantic -Werror -Wextra -c q_learning.c -o q_learning.o
clang q_learning.o -Wall -lm -o run
rm -rf q_learning.o
(8, 8) (8, 9) (7, 9) (6, 9) (6, 10) (5, 10) (4, 10) (4, 9) (4, 8) (4, 7) (3, 7) (2, 7) (1, 7) (0, 7)
87
./run  0.00s user 0.00s system 8% cpu 0.051 total


==================================
Java implementation:
==================================
javac Q_Learning.java
(8, 8) (8, 9) (7, 9) (6, 9) (6, 10) (5, 10) (4, 10) (4, 9) (4, 8) (3, 8) (3, 7) (2, 7) (1, 7) (0, 7)
87
java Q_Learning  0.29s user 0.05s system 148% cpu 0.225 total


==================================
C++ implementation:
==================================
rm -rf run
g++ -o run q_learning.cpp
(8, 8) (8, 9) (7, 9) (6, 9) (6, 10) (5, 10) (4, 10) (4, 9) (4, 8) (4, 7) (3, 7) (2, 7) (1, 7) (0, 7)
87
./run  0.00s user 0.00s system 54% cpu 0.009 total
```

Figure 5: Output of testing script stored in logs.txt

6 Conclusion

This project solidified many of my pre-existing notions of the languages that were implemented. While being the fastest, C has the least built in help in the forms of structs, polymorphism, and memory management amounting to the largest line count. C++ is a close contender with many similarities. Java, while being object oriented, still requires some tedious work being put into basic structs. Python, which was the easiest and fastest to implement, was by far the slowest. Q-learning was found to be a simple but effective algorithm at exploring environments. This model free approach implements low dimensional arrays and basic algebra to come up with a fast path finding solution. In conclusion, the project greatly helped me explore the different programming languages and understand Q-learning.

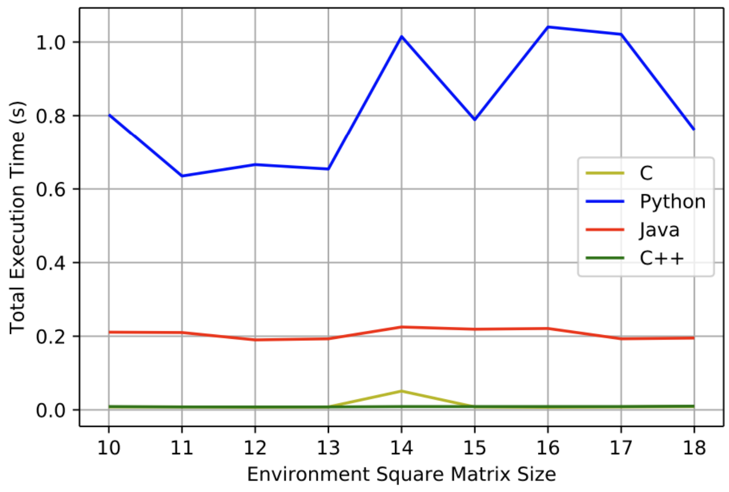


Figure 6: Execution times

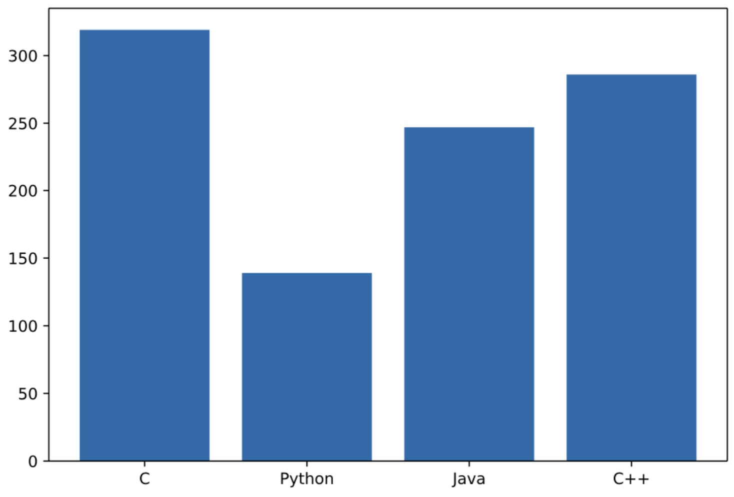


Figure 7: Lines of code by language

7 Future Work

There are many other types of analysis that can be done on these implementations. The hyper-parameters were not tuned in this project and we could see some interesting results when changing those. By reducing the learning rate, increasing training episodes, etc., we may find the different implementations coming to different conclusions on a best path. This is because when the environment is large, the algorithm does not have time to explore the entire environment and come up with an optimal solution. Therefore, much larger environments could be interesting to look into. Finally, changing the problem itself into an environment where actions are stochastic would incur some uncertainty that could be interesting. There are also many other reinforcement learning algorithms that could be applied to this problem.

8 References

[1] "Q-Learning: A Complete Example in Python" YouTube, uploaded by Dr. Daniel Soper, 24 Apr. 2020, https://www.youtube.com/watch?v=iKdlKYG78j4t=158s

[2] Viet Hoang Tran Duong (2021, February 23) Intro to reinforcement learning: temporal difference learning, SARSA vs. Q-learning https://towardsdatascience.com/intro-to-reinforcement-learning-temporal-difference-learning-sarsa-vs-q-learning-8b4184bb4978

[3] Baijayanta Roy (2019, September 12) Temporal-Difference (TD) Learning https://towardsdatascience.com/temporal-difference-learning-47b4a7205ca8