East London Pub Crawl

IN3016 Software Agents

Q-learning algorithm for reinforcement learning

Tomas Pilvelis BSc Computer Science

City, University of London

Table of Contents

[Domain and Task 3](#_Toc511291264)

[State Transition Function 4](#_Toc511291265)

[Reward Function 4](#_Toc511291266)

[Policy 5](#_Toc511291267)

[Representation of problem with Graph 5](#_Toc511291268)

[R-matrix 6](#_Toc511291269)

[Set parameter values 7](#_Toc511291270)

[Q-Matrix Updates per learning episode 8](#_Toc511291271)

[Episode 1: 9](#_Toc511291272)

[Episode 2: 10](#_Toc511291273)

[Representing performance vs episodes 11](#_Toc511291274)

[Trained Q Matrix 13](#_Toc511291275)

[Optimal Paths 14](#_Toc511291276)

[Analysis of results quantitatively 14](#_Toc511291277)

[Analysis of results qualitatively 15](#_Toc511291278)

[Appendix 1: Code 15](#_Toc511291279)

# Domain and Task

Here we will determine the best path of generating the ‘best’ pub crawl from the East of London (Windsor Fenchurch, Fenchurch Street) towards the City of London University’s favourite local pub (The Angel, Angel) The best route will be determined by implementing a reinforcement learning algorithm, Q-Learning.

This works by changing states and performing actions, through the actions rewards can be gained. It will go through each state determining the optimal action to take. The goal is to maximise the total reward gained in the optimal shortest path.

In this domain we will place an agent onto a map of pubs defined as states beginning in state 1 (Windsor Fenchurch), ending in state 20 (The Angel). The goal of the agent is to locate a route that get to the last pub through the 20 pubs whilst keeping the time taken to a minimum.

See Fig 1. This displays all the states (pubs) and all the possible routes that can be taken. When gathering the states and edges I excluded routes that are 10 mins or over walking distance because in real world application a mixture of cold weather and lack of warmth of a pub, morale of a group would slowly dissipate and in extreme cases cause hypothermia.

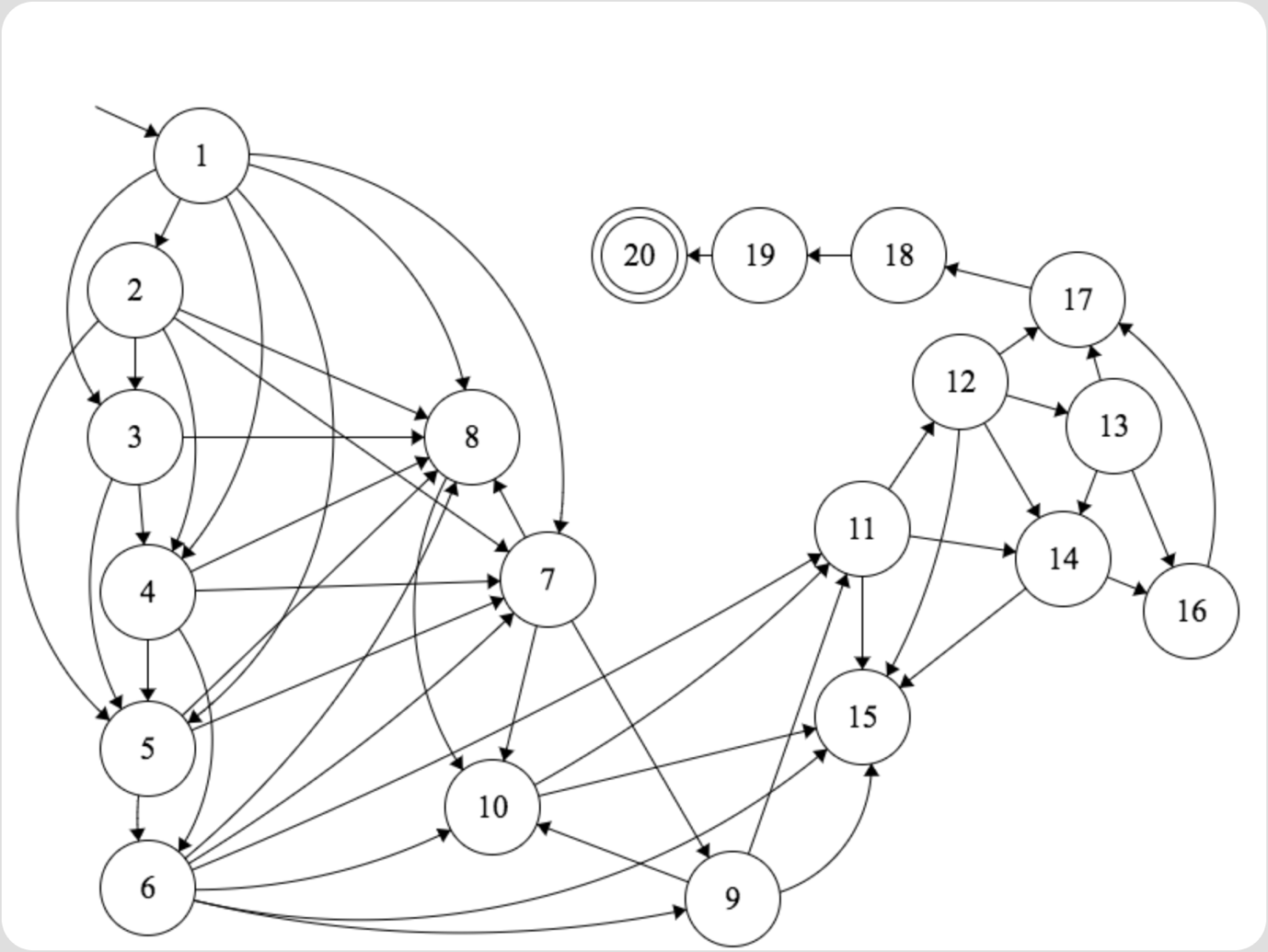


Fig 1

# State Transition Function

The following shows where left hand states can go to on the right hand side. The right hand side is displayed as a set meaning that from the left hand side it can go to any of the states in that set. E.g. State 14 can transition to state 16 or state 17.

Fig 2 shows all possible transitions.

Fig 3 shows the state number and their respective pub allocated.

st+1= δ(st, at )

1 -> {2, 3, 4, 5, 7, 8}

2 -> {3, 4, 5, 7, 8}

3 -> {4, 5, 8}

4 -> {5, 6, 7, 8}

5 -> {6, 7, 8}

6 -> {7, 8, 9, 10, 11, 15}

7 -> {8, 9, 10}

8 -> {10}

9 -> {10, 11, 15}

10 -> {11, 15}

11 -> {12, 14, 15}

12 -> {13, 14, 15, 17}

13 -> {14, 16, 17}

14 -> {16, 17}

15 -> {}

16 -> {17}

17 -> {18}

18 -> {19}

19 -> {20}

20 -> {20}

1. Windsor Fenchurch

2. The Swan Tavern

3. Corney & Barrow Lime Street

4. The Swan

5. The Crosse Keys

6. Golden Fleece

7. The Vintry

8. Cock & Woolpack

9. One New Change, Rooftop Bar

10. Williamson's Tavern

11. The Viaduct Tavern

12. The Sutton Arms

13. The Jerusalem Tavern

14. Grand Union Farringdon

15. Lord Raglan

16. Betsey Trotwood

17. The Slaughtered Lamb

18. The Blacksmith & The Toffeemaker

19. Old Red Lion Theatre Pub

20. The Angel

fig 2 fig 3

# Reward Function

The reward function will work by collecting as much rewards as possible in order to determine the most pubs visited route. Positive values for rewards are given dependant on time taken to walk to next transition. There is no reward given if a transition is over 10 mins because there will be no edge. +2 for a transition that takes less than 5 mins and +1 for 6-9 mins.

rt+1 = r(st , at )

# Policy

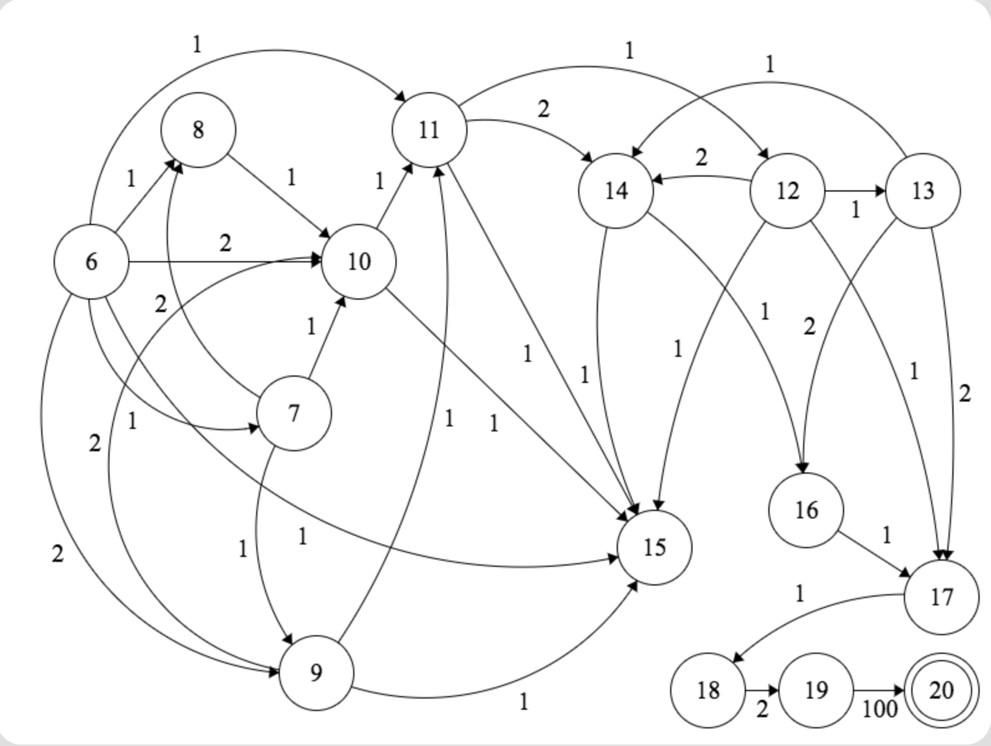
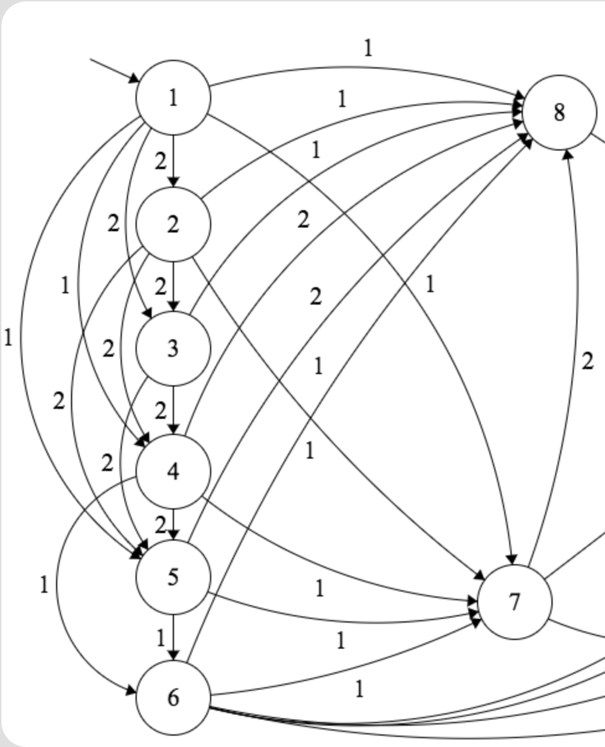
In this problem I will explore using the epsilon greedy policy. This being because I want the agent to go towards the final state and also maximise the rewards.

Epsilon greedy will use the actions with the highest reward value within the Q matrix of the state it is currently in, after the whole matrix has been calculated.

From here we can select random action and select an action with 1 epsilon that will give maximum reward in the current state. In another case if 0 will choose greedily. Because the agent will have multiple states it can go to there are actions that provide the same reward so its best option is to randomly explore a new path. The epsilon probability of an even can occur.

# Representation of problem with Graph

This graph has been divided into multiple parts due to the size of edges and states. The lack of tools to draw a legible FSM diagram were limited. This is the same graph as fig 1 just with the reward values. For a more clearer diagram without the edge values please see Fig. 1



# R-matrix

The R matrix below displays the rewards given.

Empty cells means that a move from that state is not possible to accomplish.

A value of 1 or 2 shows that this is a valid action from the state the agent is in ‘to go to state’

2 will be rewarded if a pub is <= 5 mins of the current pub (state)

1 will be rewarded if time to walk from current state to go to state is > 5 up to 9 mins

100 means that a reward has been granted of the value 100. This is the max possible value that can be gained.

This is to encourage less time outside.

Agent to go to state

Agent in state now

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 1 |  | 0 | 0 | 0 | 0 |  | 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  | 0 | 0 | 0 |  | 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  | 0 | 0 |  |  | 0 |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  | 0 | 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  | 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  | 0 | 0 | 0 | 0 | 0 |  |  |  | 0 |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  | 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  | 0 |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  | 0 | 0 |  |  |  | 0 |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  | 0 |  |  |  | 0 |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |  |  |  | 0 |  | 0 | 0 |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |  |  |  |  | 0 | 0 | 0 |  | 0 |  |  |  |
| 13 |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |  | 0 | 0 |  |  |  |
| 14 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 | 0 |  |  |  |
| 15 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 16 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |  |  |  |
| 17 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |  |  |
| 18 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |  |
| 19 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 100 |
| 20 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |

Fig 5

# Set parameter values

The alpha and the gamma values are the parameters for Q-Learning.

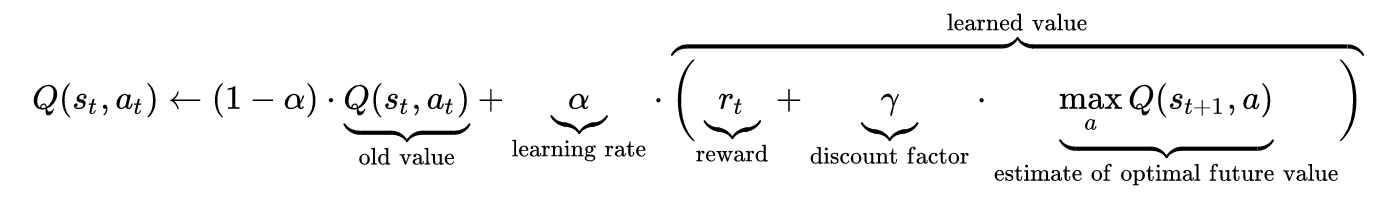
Alpha α is the learning rate of the algorithm. Gamma γ is the discount factor.

The discount factor for the transition rule equation will determine the important of the future rewards. 0 will only consider current rewards. Whereas 1 will aim for long-term rewards. In cases it is documented that starting with a low discount factor and subsequently increasing the value towards the final value accelerates learning. **François-Lavet, Vincent;** **“Fonteneau, Raphael; Ernst, Damien (2015-12-07). "How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies"**

The alpha value or the learning rate is like the gamma rate where it must be between 0 and 1. When set to 0 will encourage no learning so will exploit however setting to the other extreme of 1 will encourage exploration. This will ignore prior knowledge.

Alpha α = 0.7

Gamma γ = 0.8



# Q-Matrix Updates per learning episode

In order to demonstrate the Q Learning algorithm in action I will demonstrate how it functions through a learning episode.

Once the R Matrix has been set up to display immediate rewards for the agent in the corresponding state-action pair. Our agent is set with the parameter values of Alpha α = 0.7

Gamma γ = 0.3 as above.

We will initialise a Q matrix. This will be the same table as the R Matrix however all state action pairs will be set to 0.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 18 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Now looking back at the R matrix (fig 5) we see there are 6 possible routes our agent can take. The possible states are 2, 3, 4, 5, 7 or 8. By a random selection we choose to go to state 3. Now by going to state 3 we have 3 possible actions we can take to go to state 4, 5 or 8.

Equation:

Q(state, act) = R(state, act)+α\*{R(state, act)+ γ\* Max[Q(next state, all act)–Q(state, action)}

## Episode 1:

Q(18,19) = R(18 , 19) + 0.7 \* {R(18, 19) + 0.3\* Max[Q(19, 20)] – Q(18, 19)}

At the current episode:

Q(18,19) = 0

R(18, 19) = 2

Max[Q(19, 20)] equal to 0

Therefore,

Q(18, 19) = 0 + 0.7 \* {2 + 0.3 \* Max[0] - 0}

Q(18, 19) = 1.61

Updated Q Matrix following episode:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 13 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 15 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 16 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 17 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 18 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1.61 |  |
| 19 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## Episode 2:

State 1 Possible actions are 2,3,4,5,7,8.

The policy will randomly explore to state 2.

The following states would then be possible 3,4,5,7,8

Q(1, 2)=R(1, 2)+0.7\*{R(1, 2) + 0.3\* Max[Q(2, 3),Q(2, 4),Q(2, 5),Q(2, 7) ,Q(2, 8)] – Q(1, 2)}

At the current episode:

Q(1,2) = 0

R(1,2) = 2

Max[0, 0, 0, 0, 0]

Therefore,

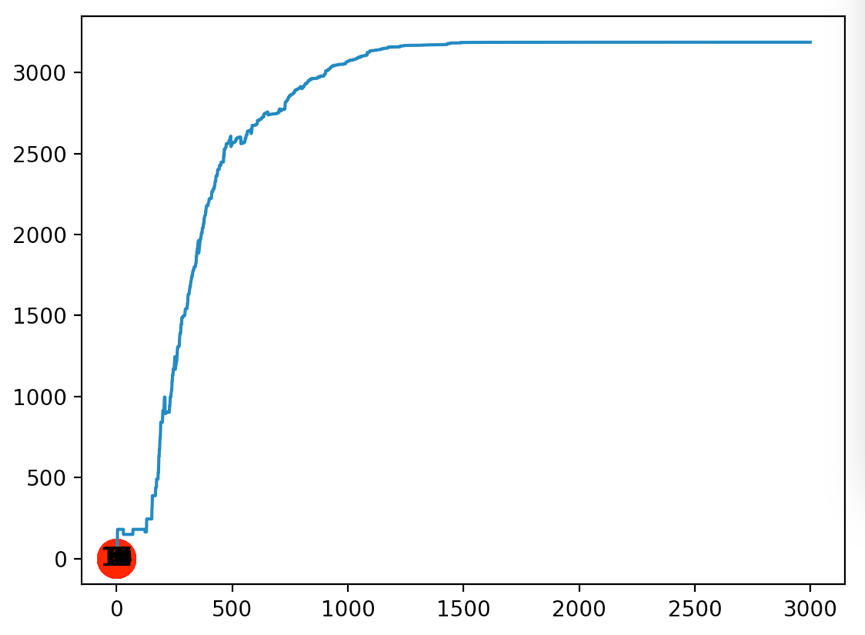
Q(1, 2) = 0 + 0.7 \* {2 + 0.3 \* Max[0] - 0}

Q(1, 2) = 1.61

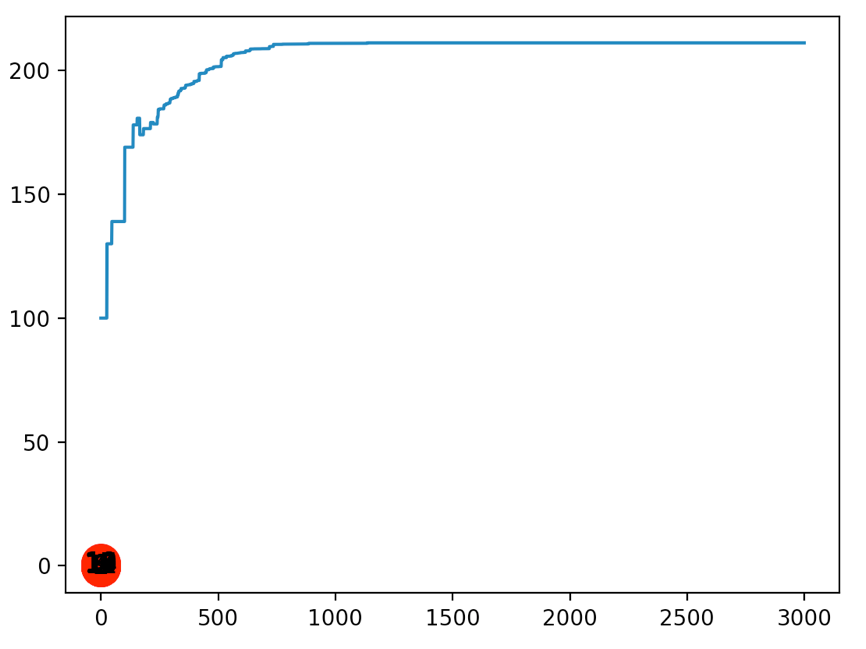
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 1 |  | 1.61 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 13 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 15 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 16 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 17 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 18 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1.61 |  |
| 19 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

# Representing performance vs episodes

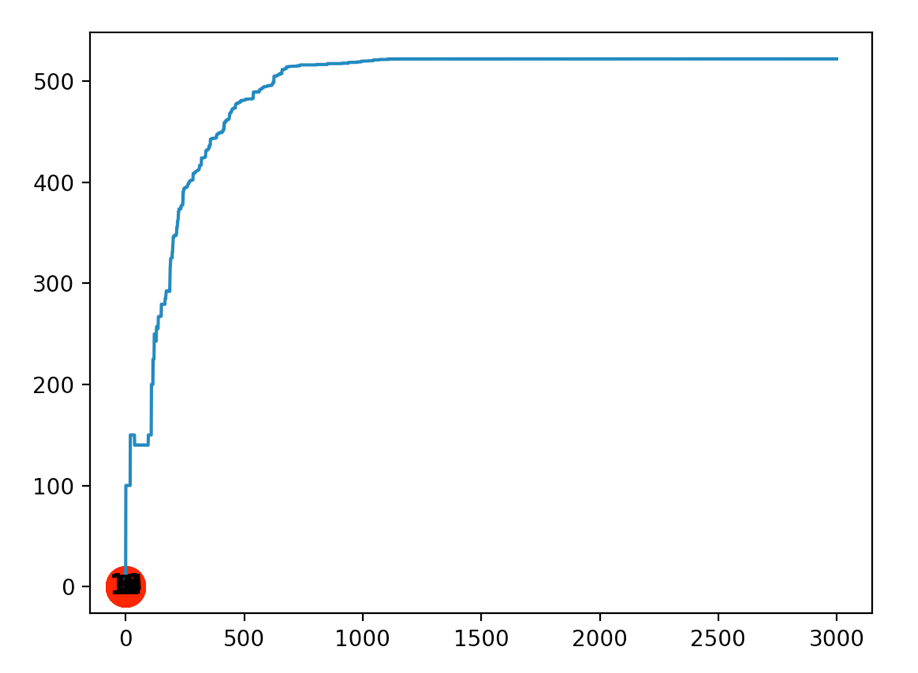
These tables show the amount of episodes on the X axis and then on the Y axis shows the steps



Gamma 0.8



Gamma = 0.3

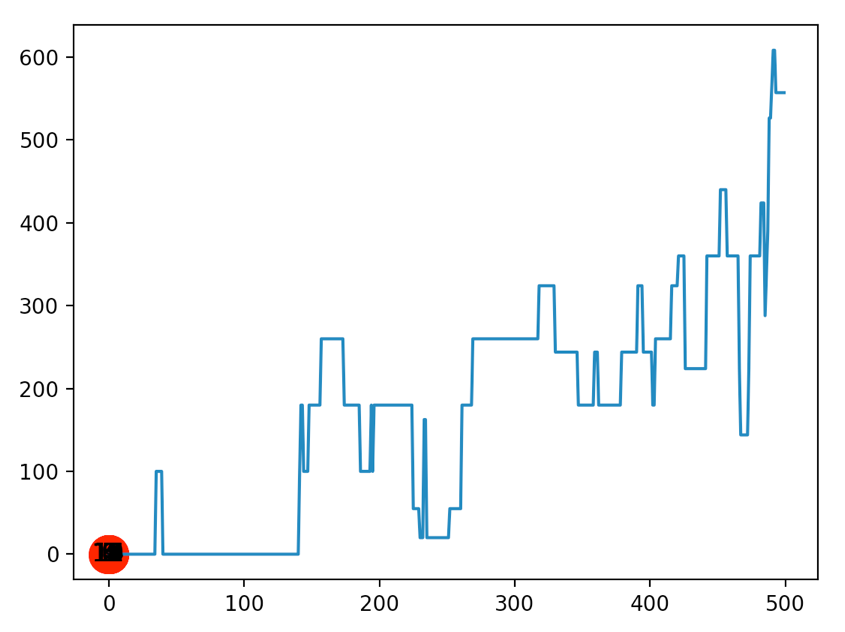


Gamma 0.5

With the alpha value now in the equation it has been difficult to allow the application to run through more episodes so that has been a limitation in gathering information

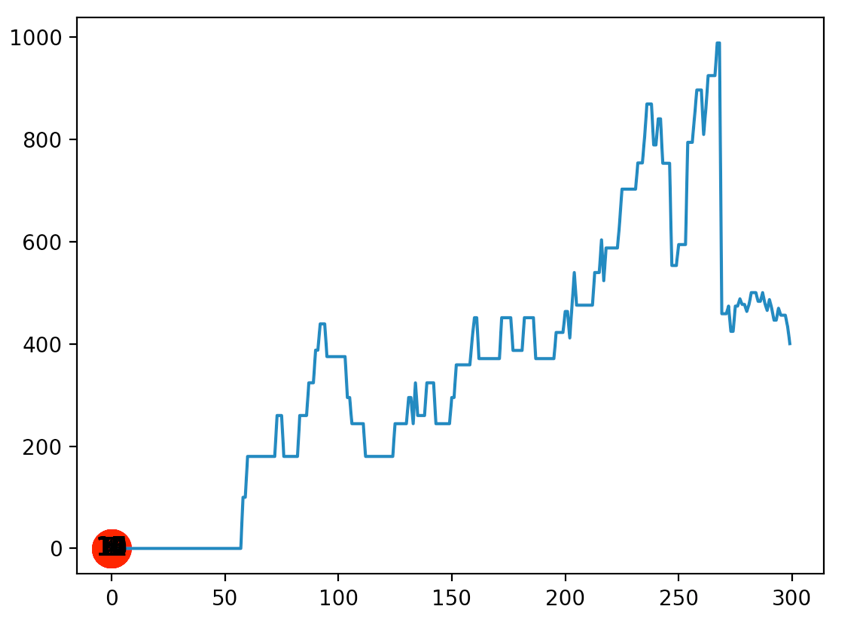
Gamma = 0.8

Alpha = 0.2



Gamma = 0.8

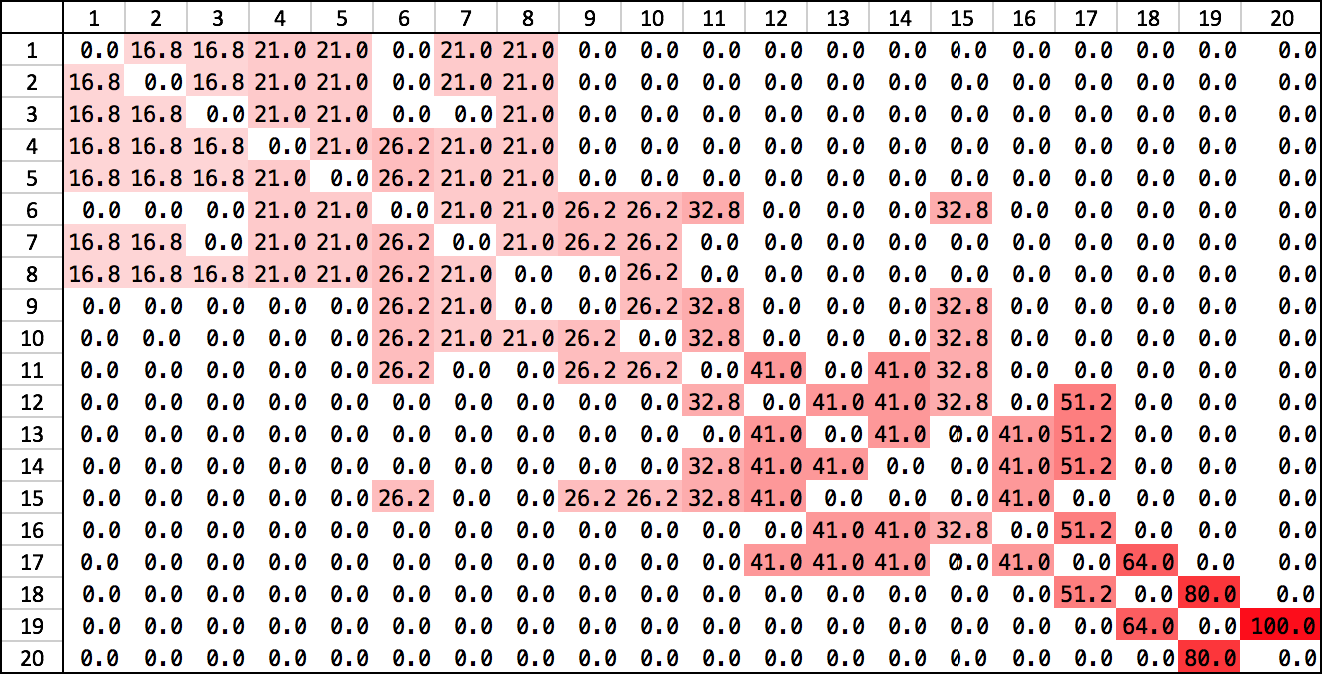
Alpha = 0.5



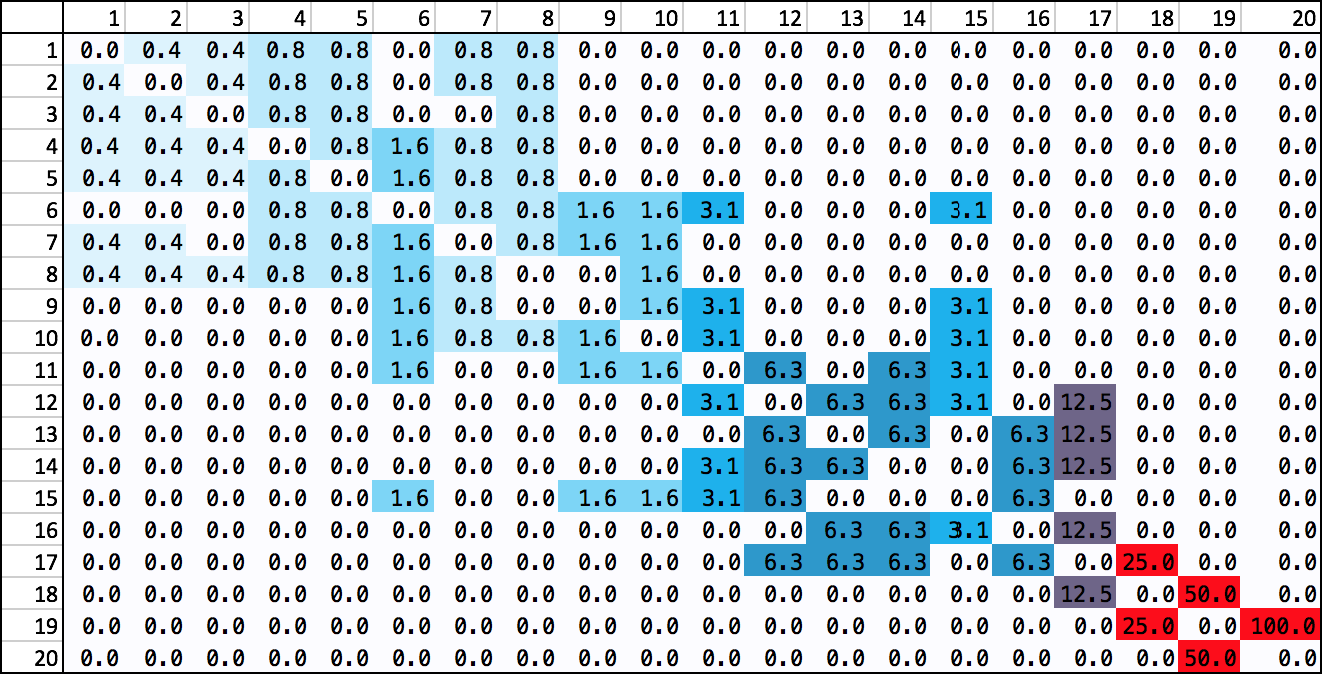
# Trained Q Matrix

Values will be rounded to 2 decimal places because when the Q matrix was generated left over 10 decimal places

Gamma = 0.8



Gemma = 0.5



Placing the gamma value at lower values such as 0.4 and less creates numbers to the power of at decimal values however the trend is still quite similar in values.

Both converge at the 1000 mark no matter the gamma value

Most Efficient Path at Gamma 0.8

# Optimal Paths

Gamma = 0.8

1st Run [0, 6, 5, 10, 13, 16, 17, 18, 19]

2nd Run [0, 3, 5, 14, 11, 16, 17, 18, 19]

3rd Run [0, 3, 5, 14, 15, 16, 17, 18, 19] -> Optimal

4th Run [0, 3, 5, 10, 13, 16, 17, 18, 19] -> Optimal

5th Run [0, 3, 5, 10, 11, 16, 17, 18, 19] -> Optimal

6th Run [0, 7, 9, 10, 13, 16, 17, 18, 19] -> Optimal

7th Run [0, 7, 5, 10, 13, 16, 17, 18, 19]

8th Run [0, 6, 5, 14, 11, 16, 17, 18, 19]

9th Run [0, 6, 8, 10, 11, 16, 17, 18, 19] -> Optimal

10th Run [0, 7, 5, 10, 11, 16, 17, 18, 19]

# Analysis of results quantitatively

Looking at 3 different gamma values produced different types of results.

Due to my lack of programming skill I had to adapt my states and actions.

# Analysis of results qualitatively

Looking at the information gathered between different parameter values I have noticed that a lower gamma value provided better results. When the gamma was lower at 0.5 going back to a state would provide approx. 50% less of a reward. Whereas a gamma of 0.8 would only lower the Q reward by approx. 25%. It was more difficult to gain data for results working with the full formula which included the alpha value because my algorithm using pylab would not generate a scatterplot if there were high iterations or when I attempted to place the alpha at a higher value than 0.5. I presume this would be because the computation took too long.

# Appendix 1: Code

import numpy as np

import pylab as plt

# Fenchruch Street to Angel

# In order of 1st by fenchurch street

# If Above 10 list with \*

# Name

# 0. Windsor Fenchurch,

# 1. The Swan Tavern,

# 2. Corney & Barrow Lime Street,

# 3. The Swan

# 4. The Crosse Keys

# 5. Golden Fleece

# 6. The Vintry

# 7. Cock & Woolpack

# 8. One New Change, Rooftop Bar,

# 9. Williamson's Tavern

# 10. The Viaduct Tavern

# 11. The Sutton Arms

# 12. The Jerusalem Tavern

# 13. Grand Union Farringdon

# 14. Lord Raglan

# 15. Betsey Trotwood

# 16. The Slaughtered Lamb

# 17. The Blacksmith & The Toffeemaker

# 18. Old Red Lion Theatre Pub

# 19. The Angel

#

# Graph with minutes from state to

#   /   0   1   2   3   4   5   6   7   8   9   10 11 12 13 14 15 16 17 18 19

#   0   /   2,  3,  6,  7,  ,   ,   9,  ,   , , , , , , , , , , ,

#   1   ,   /   3,  4,  5,  ,   9,  7,  ,   ,   , , , , , , , , , ,

#   2   ,   ,   /   4,  5,  ,   ,   6,  ,   ,   , , , , , , , , , ,

#   3   ,   ,   ,   /   2,  9,  6,  4,  ,   ,   , , , , , , , , , ,

#   4   ,   ,   ,   ,   /   9,  7,  3,  ,   ,   , , , , , , , , , ,

#   5   ,   ,   ,   ,   ,   /   7,  8,  4,  2,  9, , , , 8, , , , , ,

#   6   ,   ,   ,   ,   ,   ,   /   5,  9,  8,  , , , , , , , , , ,

#   7   ,   ,   ,   ,   ,   ,   ,   /   ,   9,  , , , , , , , , , ,

#   8   ,   ,   ,   ,   ,   ,   ,   ,   /   3,  6, , , , 6, , , , , ,

#   9   ,   ,   ,   ,   ,   ,   ,   ,   ,   /   9, , , , 7, , , , , ,

#   10  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   / 8, , 5, 7, , , , , ,

# 11 , , , , , , , , , , , / 8, 5, 7, , 8, , , ,

# 12 , , , , , , , , , , , , / 6, , 5, 5, , , ,

# 13 , , , , , , , , , , , , , / , 9, 8, , , ,

# 14 , , , , , , , , , , , , , , / , , , , ,

# 15 , , , , , , , , , , , , , , , / 8, , , ,

# 16 , , , , , , , , , , , , , , , , / 9, , ,

# 17 , , , , , , , , , , , , , , , , / 5, ,

# 18 , , , , , , , , , , , , , , , , , / 6,

# 19 , , , , , , , , , , , , , , , , , , /

# 1 -> {2, 3, 4, 5, 7, 8}

# 2 -> {3, 4, 5, 7, 8}

# 3 -> {4, 5, 8}

# 4 -> {5, 6, 7, 8}

# 5 -> {6, 7, 8}

# 6 -> {7, 8, 9, 10, 11, 15}

# 7 -> {8, 9, 10}

# 8 -> {10}

# 9 -> {10, 11, 15}

# 10 -> {11, 15}

# 11 -> {12, 14, 15}

# 12 -> {13, 14, 15, 17}

# 13 -> {14, 16, 17}

# 14 -> {16, 17}

# 15 -> {}

# 16 -> {17}

# 17 -> {18}

# 18 -> {19}

# 19 -> {20}

# 20 -> {}

# map State to action

state\_action\_list = [(0,1), (0,2), (0,3), (0,4), (0,6), (0,7), (1,2), (1,3), (1,4), (1,6), (1,7), (2,3),

(2,4), (2,7), (3,4), (3,5), (3,6), (3,7), (4,5), (4,6), (4,7),

(5,6), (5,7), (5,8), (5,9), (5,10), (5,14),

(6,7), (6,8), (6,9),(7,9), (8,9), (8,10), (8,14), (9,10), (9,14), (10,11), (10,13), (10,14),

(11,12), (11,13), (11,14), (11,16), (12,13), (12,15), (12,16), (13,15), (13,16), (14,15),

(15,16),(16,17),(17,18),(18,19)]

#Goal state initialised

goal = 19

#Import used to create network diagram

#https://networkx.github.io/documentation/networkx-1.9.1/tutorial/tutorial.html?highlight=graph

import networkx as nx

#Create an empty graph

G=nx.Graph()

#Adds multiple edges. edges defined in the state\_action\_list

G.add\_edges\_from(state\_action\_list)

#Positions nodes in specific algorithm

pos = nx.spring\_layout(G)

#Draw nodes/states of graph G in positions

nx.draw\_networkx\_nodes(G,pos)

#Draw edges of graph G in positions

nx.draw\_networkx\_edges(G,pos)

#Draw state labels

nx.draw\_networkx\_labels(G,pos)

#Uncomment to show the generated G graph

# plt.show()

# Matrix size, used for R and Q matrices

MATRIX\_SIZE = 20

# create matrix x\*y

R = np.matrix(np.ones(shape=(MATRIX\_SIZE, MATRIX\_SIZE)))

#Set all values in matrix R to -1

R \*= -1

#Uncomment to display R matrix before setting values

#print(R)

# assign zeros to paths and 100 to goal-reaching state\_action

for state\_action in state\_action\_list:

if state\_action[1] == goal:

#If the first part of tuple is the goal value of 19 set to 100

R[state\_action] = 100

else:

#If a state\_action exists from the state\_action\_list assign 0

#Edge exists

R[state\_action] = 0

#Inversed operations occour here

if state\_action[0] == goal:

R[state\_action[::-1]] = 100

else:

# reverse of state\_action

R[state\_action[::-1]]= 0

#Uncomment to print R matrix following population

#print(R)

#create Q matrix with 0 values matrix\_size x\*y

Q = np.matrix(np.zeros([MATRIX\_SIZE,MATRIX\_SIZE]))

#Learning parameter Gamma

gamma = 0.8

#Learning parameter Aplha

alpha = 0.1

# Episodes

episodes = 300

#Start state

initial\_state = 1

#Function to retrun all possible actions

def available\_actions(state):

current\_state\_row = R[state,]

av\_act = np.where(current\_state\_row >= 0)[1]

return av\_act

available\_act = available\_actions(initial\_state)

#function to return a random action from the avaliable actions

def sample\_next\_action(available\_actions\_range):

next\_random\_action = int(np.random.choice(available\_act,1))

return next\_random\_action

action = sample\_next\_action(available\_act)

def update(current\_state, action, gamma):

max\_index = np.where(Q[action,] == np.max(Q[action,]))[1]

if max\_index.shape[0] > 1:

max\_index = int(np.random.choice(max\_index, size = 1))

else:

max\_index = int(max\_index)

max\_value = Q[action, max\_index]

#Q-Learning Function for Gamma Only

#Q[current\_state, action] = R[current\_state, action] + gamma \* max\_value

#Q-Learning Function for Gamma and Alpha

Q[current\_state, action] = R[current\_state, action] + (alpha \* R[current\_state, action] + gamma \* max\_value) - Q[current\_state, action]

#print('max\_value', R[current\_state, action] + gamma \* max\_value)

if (np.max(Q) > 0):

return(np.sum(Q/np.max(Q)\*100))

else:

return (0)

update(initial\_state, action, gamma)

# Training

#Initialise scores list to empty

scores = []

#For each iteration of episodes

for i in range(episodes):

#current state is random number between 0 and 19

current\_state = np.random.randint(0, int(Q.shape[0]))

#store avaliable actions in available\_act from the current state

available\_act = available\_actions(current\_state)

#next action stored in action

action = sample\_next\_action(available\_act)

#score is the return of update

score = update(current\_state,action,gamma)

#Add to end of scores list

scores.append(score)

#print ('Score:', str(score))

#Uncomment to Print the trained Q matrix

#print("Trained Q Matrix")

#print(Q/np.max(Q)\*100)

# Testing

#Start state

current\_state = 0

steps = [current\_state]

#while the current state is not the accept state

while current\_state != 19:

next\_step\_index = np.where(Q[current\_state,] == np.max(Q[current\_state,]))[1]

if next\_step\_index.shape[0] > 1:

next\_step\_index = int(np.random.choice(next\_step\_index, size = 1))

else:

next\_step\_index = int(next\_step\_index)

#Add next\_step\_index to end of steps list

steps.append(next\_step\_index)

#set the current state to the next step

current\_state = next\_step\_index

#display line chart of scores list

plt.plot(scores)

#show

plt.show()