P4 – SmartCab Project Write-up

Implement a basic driving agent

The initial code base makes decisions randomly as to which way the Smartcab should drive. This of course means that it rarely reaches its intended destination within the allotted time. The available actions for the cab are none (no action), forward, turn left and turn right. Picking these directions at random also means that the cab doesn't pay attention to other cars or whether lights are red or green. It therefore also drives erratically and if this was a real car, it would cause many accidents!

In this simulation, there are times when the smart cab reaches its destination within the allotted time, but this is a very low percentage of times.

Deleted: See code base.

Formatted: Normal

Identify and update state

State could be defined as the inputs into the agent at this time. An example of this is:

State = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}

We also have an input which is the direction that the Planner is trying to send the Smart Cab for the next waypoint. This can be:

Left, right, forward, none

A combination of these could be used to create the state for the cab, for example:

State = {'light': 'green', 'oncoming': None, 'right': None, 'left': None, 'waypoint': 'Left'}

However, in order to work well with Python dictionaries, the format of this state has been modified use tuples, such as:

State = (('lights', 'red'), ('oncoming', None), ('right', None), ('left', None), ('waypoint', 'forward'))

The rationale for including these in the state are:

i) The state of the lights indicates whether it is safe to move from the current location, and which cars are important to pay attention to. For example, in a green light it is safe to drive forwards irrespective of other cars, but we should check for cars coming from ahead if we want to turn left).
ii) 'oncoming, 'left' and 'right' indicate whether there are other cars approaching this location, and if so which direction they are travelling or likely to travel. This, in combination with the status of the lights, gives us an indication as to whether it is safe to move from this location.
iii) The waypoint is the next direction that the planner is trying to send the smart cab. As the aim of this simulation is to get the cab to its destination, it is

Formatted: List Paragraph, Numbered + Level: 1 + Numbering Style: i, ii, iii, … + Start at: 1 + Alignment: Left + Aligned at: 0.25" + Indent at: 0.75" obviously important to bear in mind the direction that the cab needs to go. Without this, the cab would look to take the safest driving action, but it would still be directionless. In effect without this as part of the state, the cab would pick a random direction out of all of the driving actions with the best score (e.g. The safest actions)

Formatted: List Paragraph, Indent: Left: 0.75"

There are other possible environmental variables the could be included in the state, but these have been discounted for the following reasons:

Deadline: This is the time to the destination. This has not been used in the state because:

i) we do not wish for the agent to become reckless as the deadline approaches, for example running red lights
ii) it significantly increases the number of possible states making it difficult for the agent to learn the rules of the game in the typical duration of a game. In effect, the state-space would be too sparse and we would find it difficult to get q-values to converge within the timescales of the simulation
iii) The rules of the road, or the rules by which the agent has to drive, do not vary based on proximity to the deadline
iv) It could be possible to incentivise the agent to run a red light if the deadline is approaching, but this will likely increase the chance of an accident (see point i).

For the smartcab, I have allowed the state model to emerge over time. I have not set up an initial state space complete with default values as many of these may never happen (for example, we are unlikely in this simulation to see 3 cars at a particular junction).

Implement Q-Learning

The learnt reward for each state are held in a Python dictionary called 'q'. This has a key based on the state_id from the above table.

Each entry in the q dictionary contains 4 values based on the possible actions that the smart cab can take:

{'forward': -1.0, 'right': -0.5, None: 0.0, 'left': -1.0}

The function choose_action() determines if the state that the cab is in is one that has been seen before. If it isn't, then a random action is picked from the possible actions (None, forward, left, right), and if it is, then a choice is made from the actions with the highest q value. For example, if left and forward both have a q value of 2, then the agent will pick one of those at random.

What is obvious from implementing this functionality is that the cab reaches the destination much quicker and more purposefully than before. Its direction is generally towards the destination, whereas previous to this functionality, it would wander aimlessly around the environment.

Deleted:

This change in behaviour is due to learning about the rules of the environment and picking the action with the best reward. As the environment rewards for following the direction to the next waypoint, and provides a negative "reward" for not following the rules of the road, the cab learns that the best possible action is to follow the direction of the next waypoint while avoiding traffic accidents.

Enhancing the Agent

Initial runs of the code only took the immediate reward as the q-value. Although this enabled the agent to learn the rules of the game very quickly, it was only ever motivated by the short term reward of its actions (e.g. did it follow the right directions from the planner and avoid any traffic violations). Although this can be seen as providing a suitable outcome, it does not enable the agent to take action based on future potential rewards. Obviously there is a balance between an immediate reward (or penalty) and a longer term reward (or penalty).

The code was then modified to provide the following variables that can be tuned to affect the way the agent learns:

1) Alpha – This is the learning rate of the cab and determines how much of the reward for a particular action and any potential future rewards is taken into account. A value of 1 means that the entirety of this reward and future rewards are used, and a value of 0 means that the agent won't learn (as in won't update its q-values).
2) Gamma – This enables tuning of how much of a potential future reward is used in modifying the q-values. A value of 0 means only the current reward is used, whereas a value of 1 means a greater importance is put on future rewards.
3) Epsilon – This determines whether the agent goes for a random action all of the time, goes with learned actions (decisions based on q-values) only, or a mix of the 2. During early runs of the project it is typically useful to explore the world more thoroughly, and therefore a higher percentage of actions are chosen randomly. However, as the agent understands more of the rules of its environment and q-values converge, it is possible to reduce the percentage of actions that are chosen randomly.
4) Policy - I have used this variable as an easy way to compare tuned parameters with a completely random approach. This variable can be set to 'q' to use q-values (default) or any other value to force completely random actions

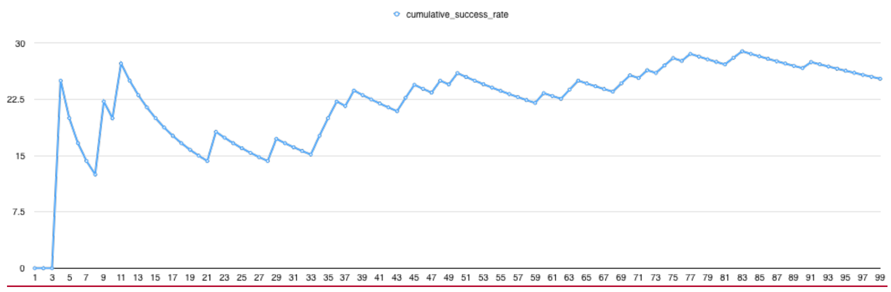
Formatted: List Paragraph, Numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 1 + Alignment: Left + Aligned at: 0.25" + Indent at: 0.5"

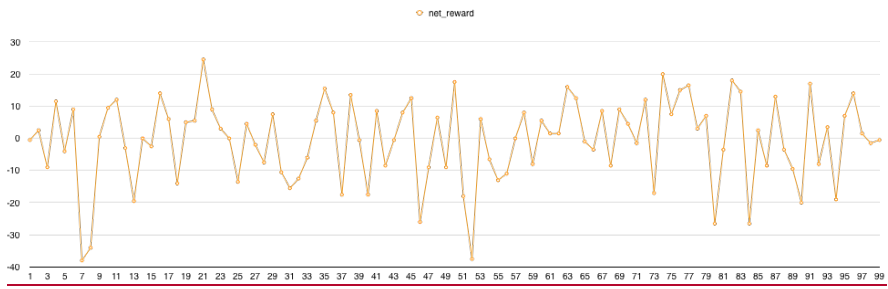
Deleted: ... [1]

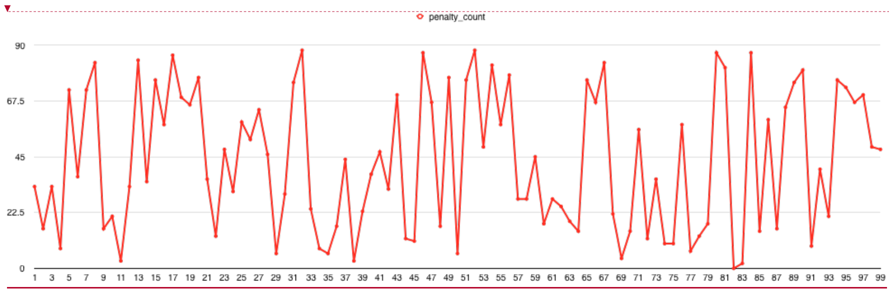
The following section describes the results of various testing runs

A Random Policy

An initial test was taken using a purely random approach to picking actions without any understanding of the environment (e.g. lights, other cars or the planner’s directions were not taken into account when picking a direction to go).

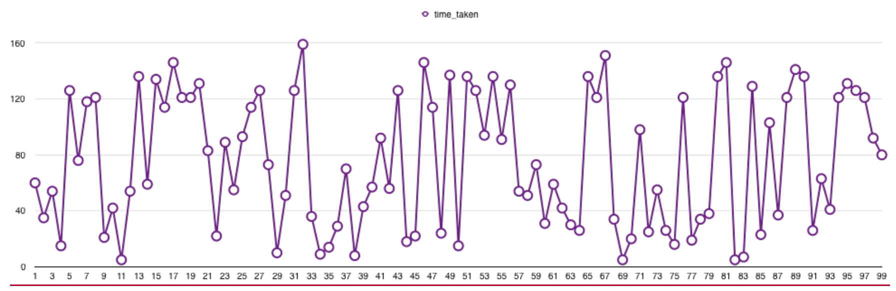






Formatted: Heading 3

Deleted: ... [2]



Deleted:

As can be seen from the graphs below, the cumulative success rate over the 100 trials is around 25%. What is more striking is that net reward over this run, with a significant number of trials receiving a negative net reward due to the high penalty count. This could be caused by not following the planner's directions or by traffic violations. The final graph shows the duration of each trial (so how long it takes the agent to reach the destination, if at all). There is no pattern to this, and it is obvious that the agent does not improve its performance over time.

For these reasons, it is obvious that a random approach is not a feasible policy.

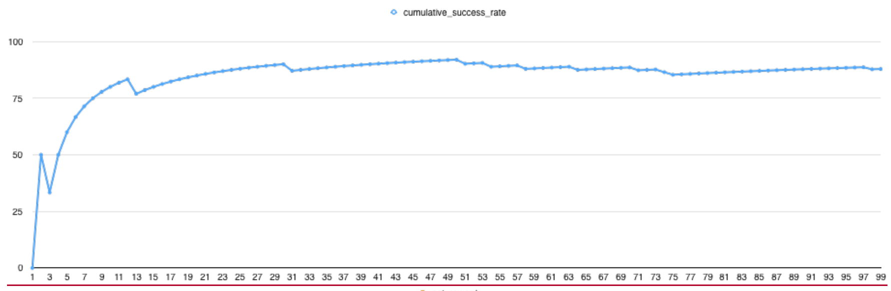
Using Q-Values

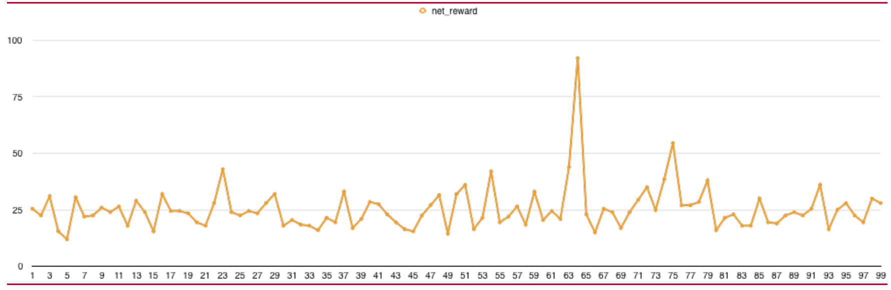
This run used the following approach to determine the action to take:

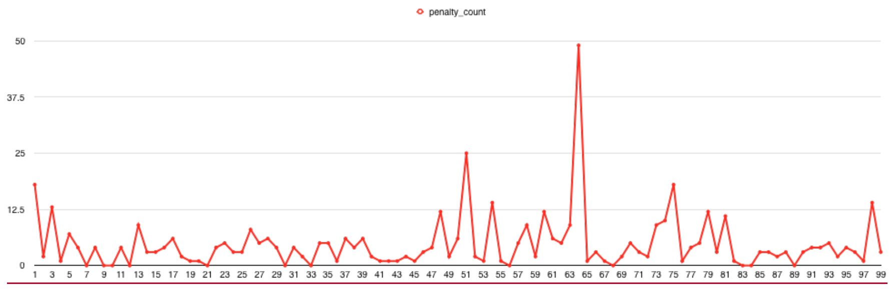
i) If we are in an unknown state, or 20% of the time (based on the value of epsilon), pick a random action
ii) If we know the state and we want to exploit our knowledge (80% of the time in this run), then pick an action from the best possible actions. If there is more than one action with the same max q-value then the action is picked at random.
iii) Alpha (0.75), gamma (0.5) and epsilon (0.8) remain unchanged during the run.

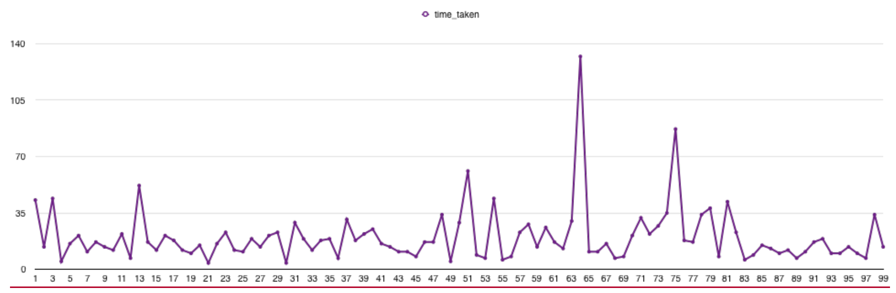
Formatted: Heading 3

Formatted: List Paragraph, Numbered + Level: 1 + Numbering Style: i, ii, iii, … + Start at: 1 + Alignment: Left + Aligned at: 0.25" + Indent at: 0.75"









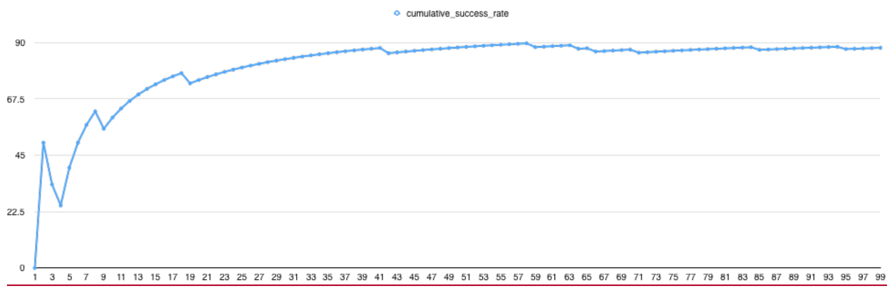
From these graphs it can be seen that the success rate across the 100 trials tends towards 100%. The net reward is also positive for the duration of the 100 trials, which is unusual as often the early trials (say the first 2-10) can produce a negative net reward as the agent learns the rules of the game.

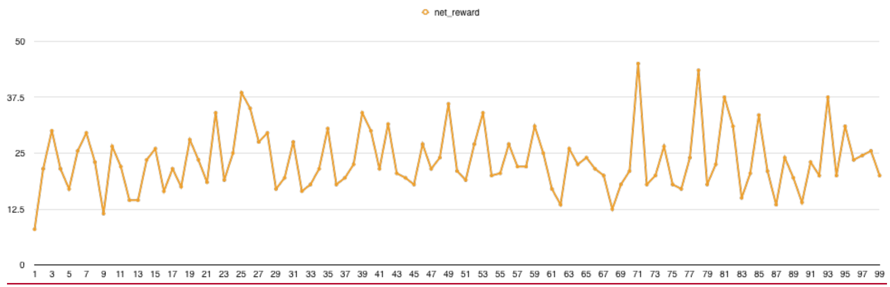
What is also evident though is that the time taken to reach the destination is much more consistent, and is typically around 20. There are a couple of outliers in this data (at steps 64 and 75) which based on the penalty count are likely due to the agent not following the planner's directions due to other traffic (at this stage in the run, we can assume that the agent understands the rules of the game in respect to traffic violations).

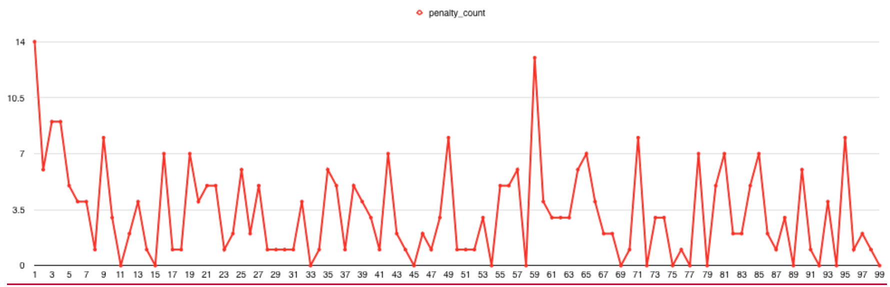
Decaying the learning rate over time

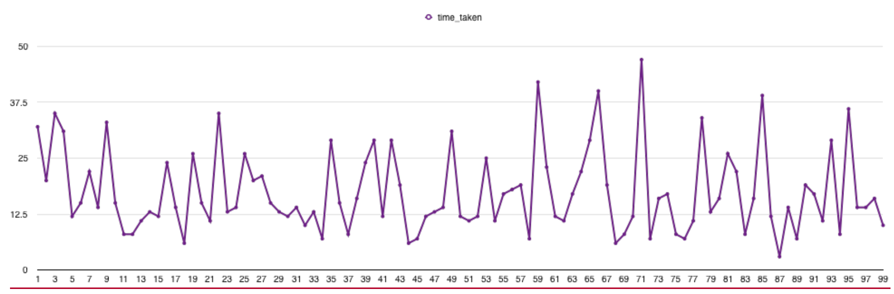
Formatted: Heading 3

In this run, the learning rate alpha is decayed over time using the formula alpha_start/trial_number. This means that in the first trial, the full initial value of alpha is used (0.75) and by the final trial we are only learning at a rate of 0.75/100. This is putting a much greater emphasis on what the agent already knows.







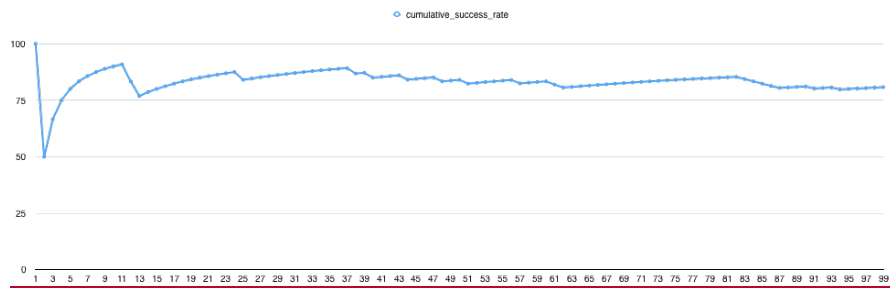


From these graphs it is obvious that the cumulative success over the 100 trials is not as good as when alpha was not decayed. This means that we are decaying alpha too fast (and therefore q-values converge too slowly) and therefore the agent doesn't get to understand the rules of the environment quickly enough.

Alpha decays each quarter

Formatted: Heading 4

In this subsequent run, alpha is decayed by 25%, 50%, 75% for each quarter of the run.



The expectations was that this would enable the agent to learn more in the first 25 trials but rely on its knowledge more over each subsequent quarter of the run. However, from this graph it can be seen that this hinders the learning of the agent.

Comparing this with the graph where alpha is not decayed, it is obvious that the agent continues to learn the environment up to trial 50.

Other values of alpha were explored, but any value less than 0.75 over the duration appears to have a negative effect on the cumulative success rate by the end of the run. Higher values of alpha (for example 0.9) did not appear to have any effect on the learning rate.

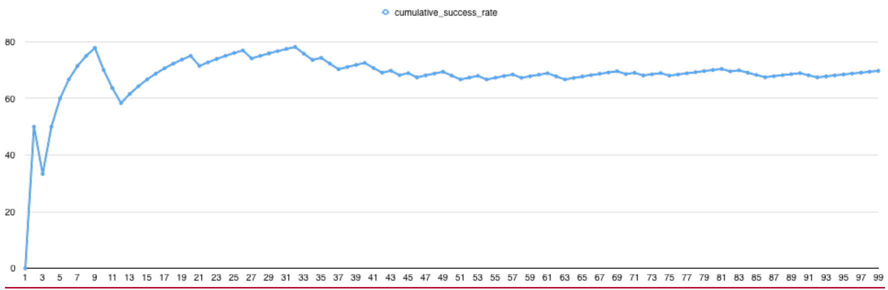
Changing the importance of future rewards

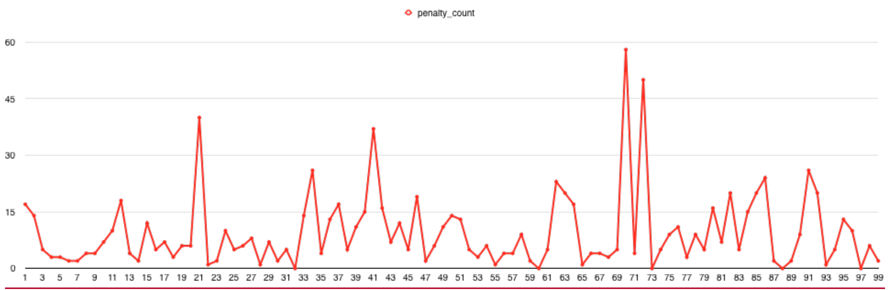
The value of gamma impacts how much of the q-value is based on the potential for future rewards, so for example running a red light means that the agent gets a bigger reward for reaching the target quicker.

Gamma = 0.75

The initial value was set to 0.5, and in this run I have increased the value to 0.75 so that the impact of future reward on the action taken for this particular state is greater.

Let's compare the cumulative success rate of this run:





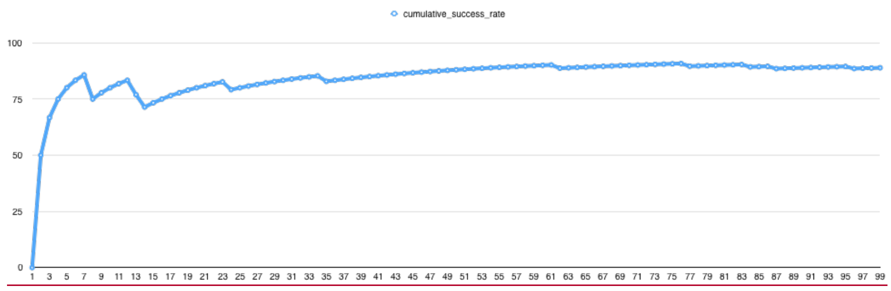
From this it can be seen that having a greater emphasis on the potential future reward decreases the cumulative rewards during the run. This also evidenced in the penalty count which is higher than the run where gamma = 0.5.

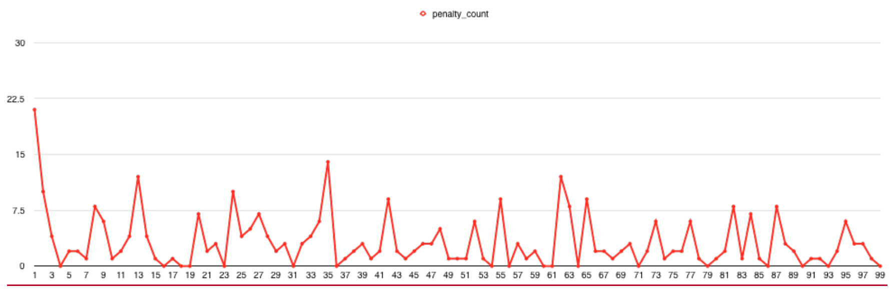
Formatted: Heading 4

Formatted: Heading 4

Gamma = 0.25

As a comparison, in this run gamma was set to 0.25, meaning less importance was put on potential future rewards.





From these results it can be seen that the number of penalties is much lower than previous runs and that the cumulative success rate of the 100 trials is around 90%.

Gamma = 0

When gamma was set to 0, the results seemed to vary wildly from a cumulative success rate of 88% up to 95%. What is obvious from this is using at least of the potential future reward when calculating the q-values reduces the reactiveness of the agent, so in effect it provides for more stable outcomes.

Formatted: Heading 4

Formatted: Heading 4

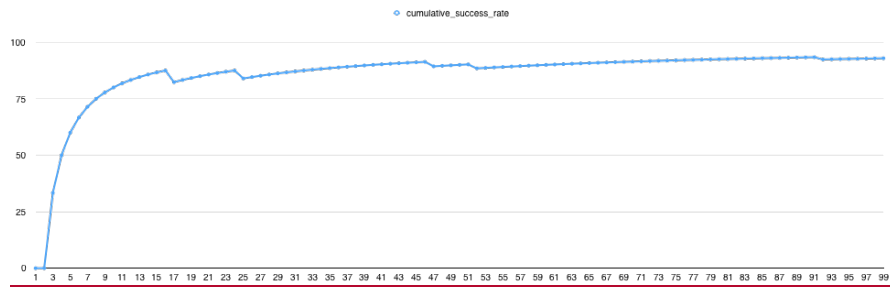
Epsilon-Greedy (Exploitation vs exploration)

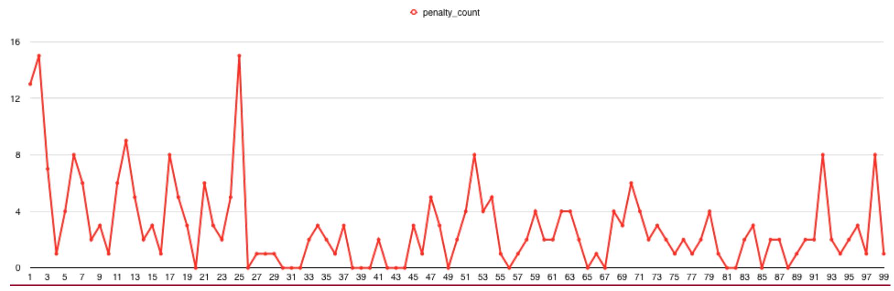
In these runs, the value of epsilon was varied over the trials to understand the impact of exploiting what has been learnt vs exploring new states.

Epsilon decays each quarter

Formatted: Heading 4

In this initial run, we look at decaying the value of epsilon (starting at 20%) by 25% each 25 runs (so it's values are 20%, 15%, 10%, 5%).



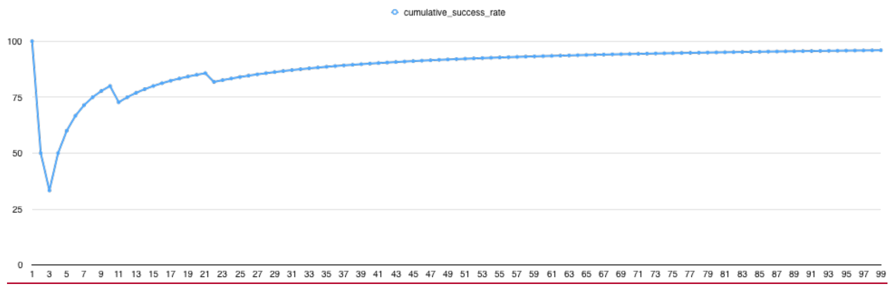


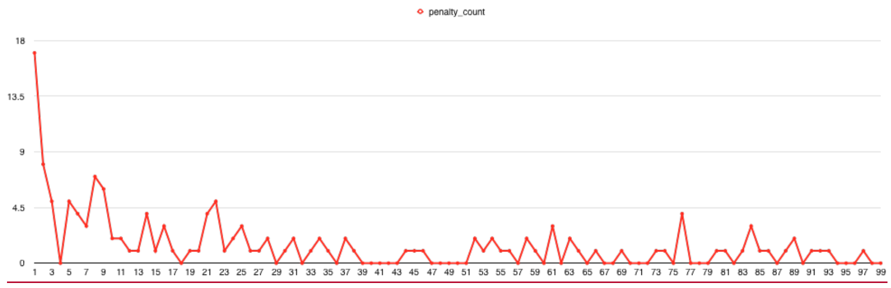
This improves the performance of the agent marginally over a constant value of 20%.

Formatted: Heading 4

Epsilon decays by 20%/trial

This test provides a faster initial decay of the value of epsilon, so in effect the learning from the initial couple of trials has a bigger impact on the later trials.





What can be seen here is that the penalty count tends to 0 after the initial runs. This has the advantage of being able to exploit existing knowledge, but there is little chance of the agent learning anything new over time. In an environment where the rules are static, this could potentially be a good thing, but the agent will not explore any states (or actions) that it doesn't learn about in the initial runs.

The Optimal Policy

There were only a few occasions when the agent went round in circles, or took a long time to reach the destination. These were in the situations where it encountered lots of traffic during the trial. It may be possible to overcome this with a better understanding of the environment (for example knowing where the final destination is), but this would also increase the state space. Some programmatic logic could be used to solve this, but this would mean that the code was specifically designed for this specific environment.

Therefore, from this project, I propose that an optimal policy would be:

i) Learning rate alpha set to 0.75 or higher. Higher values make a marginal difference but this gave consistent results over multiple runs

ii) Discounted rate set to between 0.25 and 0.5. Although this made little difference to the typical run, having some part of potential rewards introduced into the q-value provided more stable cumulative successes over 100 trials.

iii) Epsilon decaying by 25% for each quarter of the run. This reduces the likelihood of exploring new actions / states in favour of exploiting what is already known, and although the reducing epsilon much faster gave a better end result, I propose that keeping at least 5% of actions chosen randomly gives the agent chance to learn new things. There is a risk that this 5% leads to violation of traffic laws or moving away from the destination, but in a typical trial of 20-40 moves, this may result in 1-2 violations on average.

Formatted: Heading 3

Formatted: List Paragraph, Numbered + Level: 1 + Numbering Style: i, ii, iii, ... + Start at: 1 + Alignment: Left + Aligned at: 0.25" + Indent at: 0.75"

Page 3: [1] Deleted	**Mark Strefford**	**6/8/16 9:38:00 AM**

Page 4: [2] Deleted	**Mark Strefford**	**6/8/16 10:17:00 AM**