Train a Smart cab to Drive

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- Next waypoint location, relative to its current location and heading,
- Intersection state (traffic light and presence of cars), and,
- Current deadline value (time steps remaining),

And produces some random move/action `(None, 'forward', 'left', 'right')`. Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.

Run this agent within the simulation environment with `enforce_deadline` set to `False` (see `run` function in `agent.py`), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

There are many observations about the world where this car moves.

- The goal and the start location changes in each episode
- The car location and destination can be obtained from the planner
- The planner gives waypoint suggesting the desired action
- Negative reward (-1) is given when the car violates traffic rule
- If the car moves without violating the traffic rule but does not follow the waypoint, it gets 0.5 reward point but if the car does not move (hence, also not following the waypoint), it gets 1 reward point. But I'd think the car may want to move in some cases in such scenario.
- If the car moves without violating the traffic rule and also follows the waypoint it gets 2 reward points (however, the planner sometimes does a poor job of suggesting wrong direction)
- Additional 10 reward points is given when the car reaches the destination within the time limit.
- Additional reward points are no more than 10. Reaching faster to the destination does not earn more points
- The car with random actions does eventually reach to the destination by chance since enforce_dealine is set to False.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Justify why you picked these set of states, and how they model the agent and its environment.

We want an optimal mapping function from states to actions so that we can use the function for many different episodes. Therefore, we want something static rather than dynamic.

The start location, the destination and the time limit all change in each episode. Therefore, we don't want to use them as our states.

The traffic rules are consistent across all episodes. In order to react to the traffic lights and other cars, we need the traffic inputs ('light', 'oncoming', 'left', 'right') values as part of the state. This is important to avoid negative rewards.

Waypoints are advising us to reach the destination. It may not be perfect navigator as such but it gives us better chance to reach the destination than completely random actions.

Waypoints are either 'forward', 'left' or 'right'. None is used when the car reached to the goal. Since we don't need to decide further action once the car reaches to the destination, we do not consider waypoint=None case for our state (If less number of states works as good as more number of states, then less number of states are preferable as it is quicker to learn (explore)).

As the planner is not perfect, we may be wandering off the destination by following it in some cases. However, improving the planner is an independent topic since the learning to follow the waypoints are not related to the quality of waypoints itself. Therefore, I'm assuming we are not expected to modify/swap the planner in this project.

In summary, we should incorporate the following into the state:

- Waypoints for reaching the destination
- Traffic Inputs (Light, Oncoming, Left, Right) for handling the traffic rules

For each state, there are 4 possible actions.

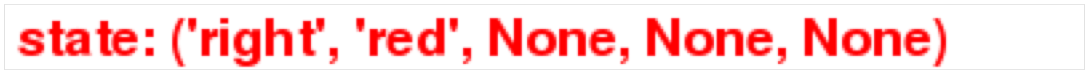
States	Values
Waypoints	'forward'
	'right'
	'left'
Traffic Lights	'green'
	'red'
Oncoming	'forward'
	'right'
	'left'
	None
Left	'forward'
	'right'
	'left'
	None
Right	'forward'
	'right'
	'left'
	None

Actions
'forward'
'right'
'left'
None

Therefore, there are 384 possible states (3 waypoints x 2 lights x 4 oncoming x 4 left x 4 right), each of which has 4 actions, giving 2048 possibilities for decision making. I'll combine state values into a tuple in the following tuple format:

```
(<Waypoint>, <Traffic Light>, <Oncoming>, <Right>, <Left>)
```

An example of a state shown in the GUI is as follows:



```
state: ('right', 'red', None, None, None)
```

Performance Measurement

Now that we have our states and actions are defined, I'd like to define our benchmarks for success rates and coverage (exploration) rates as follows:

Measurement	Formula
Success Rate(%)	$\frac{\#\ of\ times\ the\ car\ reached\ the\ destination}{\#\ of\ episodes}$
Coverage Rate (%)	$\frac{(\#\text{ of states visited at least once})}{\#\ of\ states}$

The success rate tells us how often the car reached the destination after the all episodes (for example, 95% means the car reached to the destination 95 times out of 100 episodes). I want to see how successful the car is in terms of reaching to the destination.

The coverage rate tells us how many states in Q-table is used (at least once) after the all episodes. If 38 states are visited by the car (out of 384 states), the coverage rate will be about 10%. I want to see how much exploration is done by the car using the coverage rate.

I keep track of running averages of 10 success rates for each episode to draw a graph to see how the success rate changes as more episodes run. Same goes for the coverage rates.

In this project, I will run 100 experiments (each of which has 100 episodes) for each driver type (Random, Greedy, etc, drivers) to compare the performance.

The last 10 episode defines the success rate of 1 experiment. The average of 100 experiment defines the success rate of a particular driver type. Also the standard deviations are examined. With these statistics, I can compare the performance of different driver types.

Random Driver

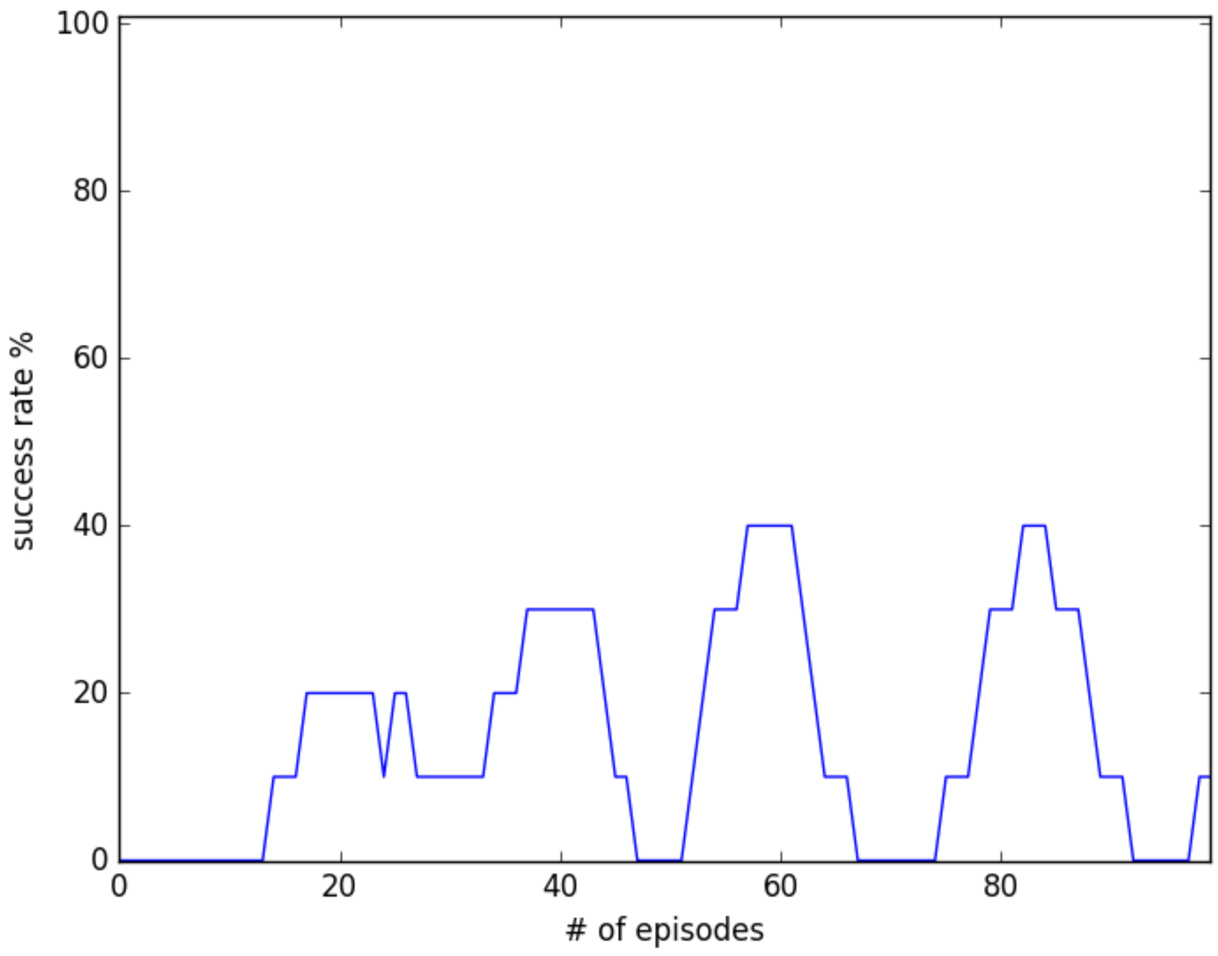
The outcome of the 100 experiments for Random Driver is as follows:

	Mean	**Std dev**	**Max**	**Min**
Success Rate	19.90%	12.67%	50.00%	0.00%
Coverage Rate	13.21%	0.80%	15.62%	11.46%

The success rate is about 20% on average. The maximum is 50%. It is very bad. Also, it is very slow to run Random Driver for 100 experiments (which includes 10000 episodes) as it often uses up the time limit.

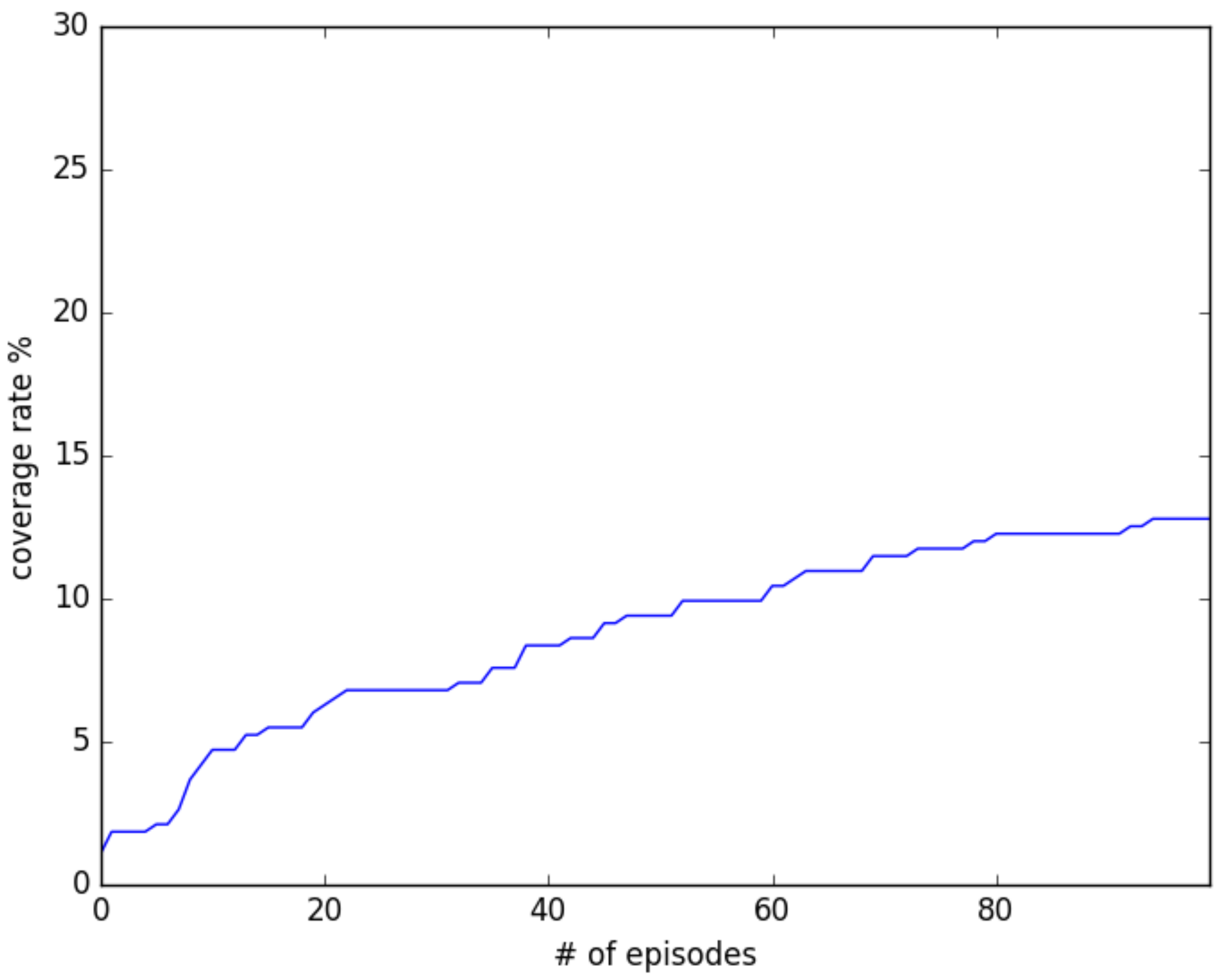
As it learns nothing from the experiment, it is a waste of time running so many episodes but this will give us the base case to compare with other driver types.

The following shows the success rates and coverage rates of one experiment randomly picked from the 100 experiments.



The success rate graph patter is very different from experiment to experiment.

The coverage improves over episodes. It explores by chance (but not learning).



All the report and image files are under ‘report/random’ folder.

File Name	**Description**
nnn.csv	The details of 100 episodes for each experiment
coverage_nnn.png	Coverage rage graph for each experiment
success_nnn.png	Success rate graph for each experiment
results.csv	The average success/coverage rates from the last 10 episodes of each experiment.
results.txt	Mean, std dev, max, min of coverage rates and success rates (the values used are the same as the success/coverage rates in results.csv)

To run the random driver, the following command is used:

```
python smartcab/agent.py random 100 100 10
```

Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

What changes do you notice in the agent's behavior?

The Q-table is implemented as a python dictionary, key of which is the state and value of which is another dictionary of (action, Q-value). All the Q-value is initially zero, and are updated using the following Q-learning formula as the car makes (state, action) decisions.

$$Q[s,a] \leftarrow (1-\alpha)Q[s,a] + \alpha(r+ \gamma\max_{a'} Q[s',a'])$$

The program has the following parameters to control the Q-learning algorithm.

Parameter	Range	Description
alpha	0.0 – 1.0	Learning rate Q-value is updated as the weighted average of the current value and the new value. Higher alpha means new Q-value has higher weight in the calculation.
gamma	0.0 – 1.0	Discount Rate Lower value means we discount delayed reward more.

Greedy Driver

Greedy Driver will always choose the best (highest Q-value) action. When there are multiple best values, it will randomly choose one of them which is to avoid any bias in decision making. If we do not randomize such case, it may always choose same action (say, 'right') based on the ordering handled by python dictionary structure.

The learning rate (alpha) is a decreasing function of time (which starts with 0 and increases) and it is calculated as below. The idea is that as the time goes, the car should have learned more and new information becomes less important (and Q value should converge into a certain value).

$$\alpha = \frac{1}{1 + time}$$

$$where\ time \geq 0$$

The discount rate (gamma) should be adjusted based on the life time. If we have longer life time, we should evaluate delayed reward lower. If we have shorter life time, we should care more about the immediate reward.

$$\gamma = \frac{1}{1 + deadline}$$

$$where\ deadline \geq 0$$

I observed each individual decision in nnn.csv files. Greedy Driver does learn the traffic rule fairly quickly since any violation of the traffic rule will update Q-value to negative value which is less than the initial Q-value of zero. So, once the car violates one of the traffic rules, it will ignore that for good.

However, Greedy Driver does not learn optimally. As it initially chooses one action randomly since all Q-values for each (state, action) pair is zero, it is possible that this decision was not the optimal one (not the global maximum). After this, the car always choses the best action and it may get stuck with that local maximum and does not perform optimally.

For example, when the waypoint is ‘forward’, the traffic light is ‘green’ and no other cars are around, if the car happens to choose right initially (reward 0.5), it will continue to choose that action in the same state whereas the optimal action should be ‘forward’ (reward 2).

Unlike Random Driver, Greedy Driver does not explore at all other than the initial choice of actions for each state (other than avoiding negative reward).

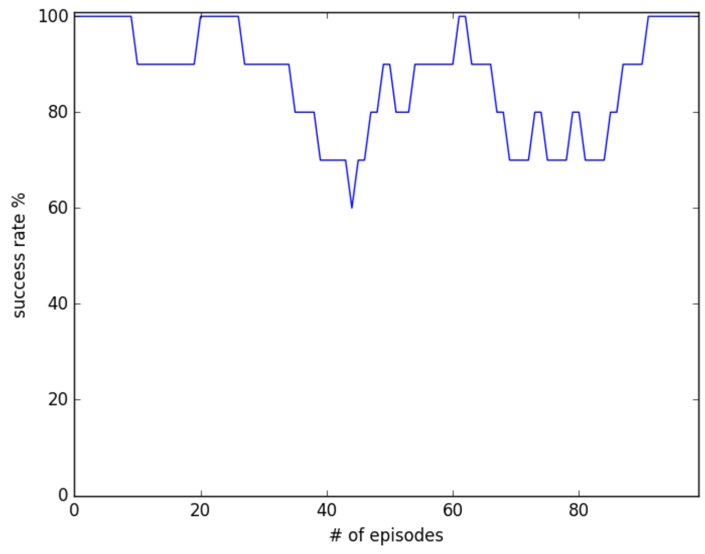
Random Driver did much more exploration (but without exploitation). Greedy Driver, on the other hand, did much more exploitation and almost no exploration.

The outcome of the 100 experiments for Greedy Driver is as follows:

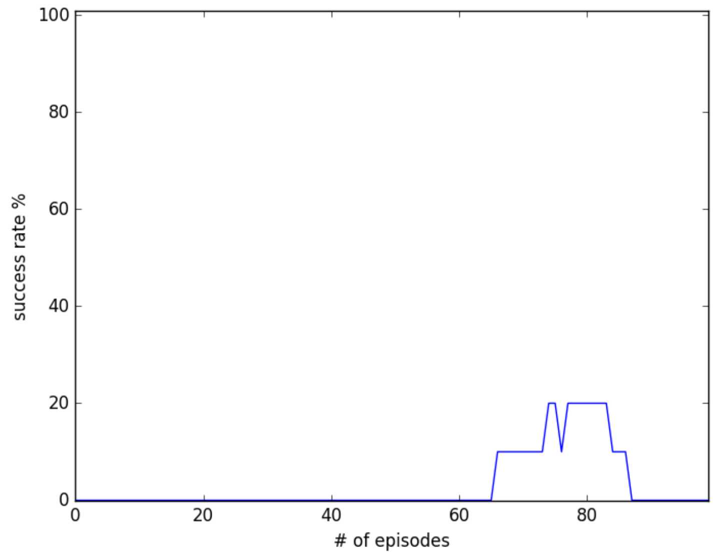
	Mean	**Std dev**	**Max**	**Min**
Success Rate	17.80%	23.03%	100.00%	0.00%
Coverage Rate	11.58%	1.80%	15.36%	7.29%

The performance of Greedy Driver is surprisingly worse than Random Driver on average and the standard deviation is much higher (almost twice). If the initial choices get lucky, it might perform very well. Otherwise, it performs very poorly. Too much exploitation but not much exploration.

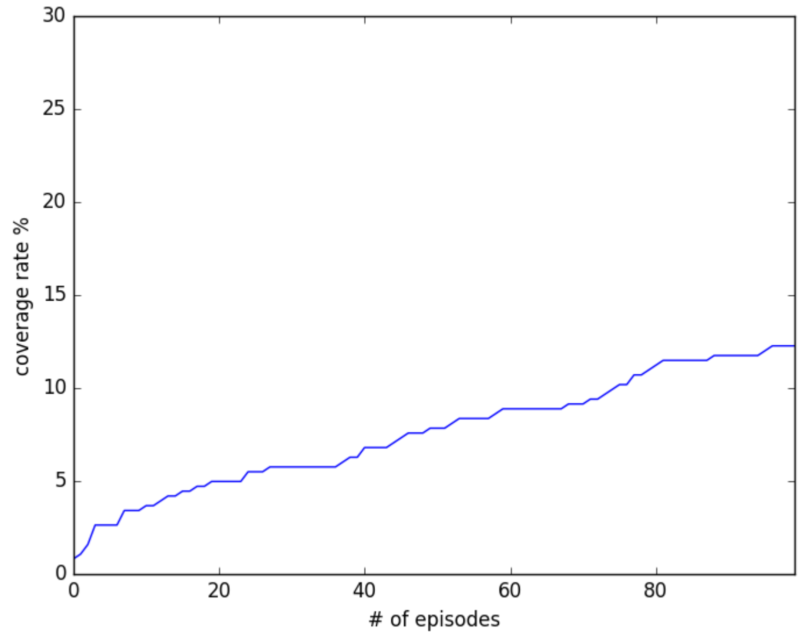
Success Rates (Lucky case)



Success Rates (No so lucky case)



The coverage rates are slightly worse than Random Driver, and the std dev is much tighter, meaning the coverage is quite similar between experiment and experiment than that of Random Driver. Again, this is the effect of not exploring.



All the report and image files are under 'report/greedy' folder.

To run the greedy driver, the following command is used:

```
python smartcab/agent.py greedy 100 100 10
```

Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

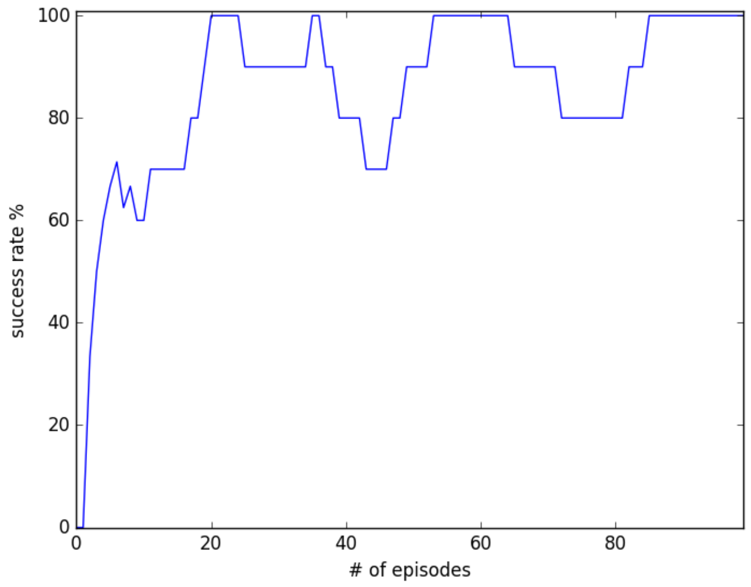
Greedy2 Driver

One problem with Greedy Driver was that it was not exploring. What we can do to force it explore more by setting the initial Q value to 1.5. The maximum reward is 2 except for the destination reward of 10. Any other rewards are less than 1.5 so that the driver will discarded it since it's always choosing a best value.

The outcome of the 100 experiments for Greedy2 Driver is as follows:

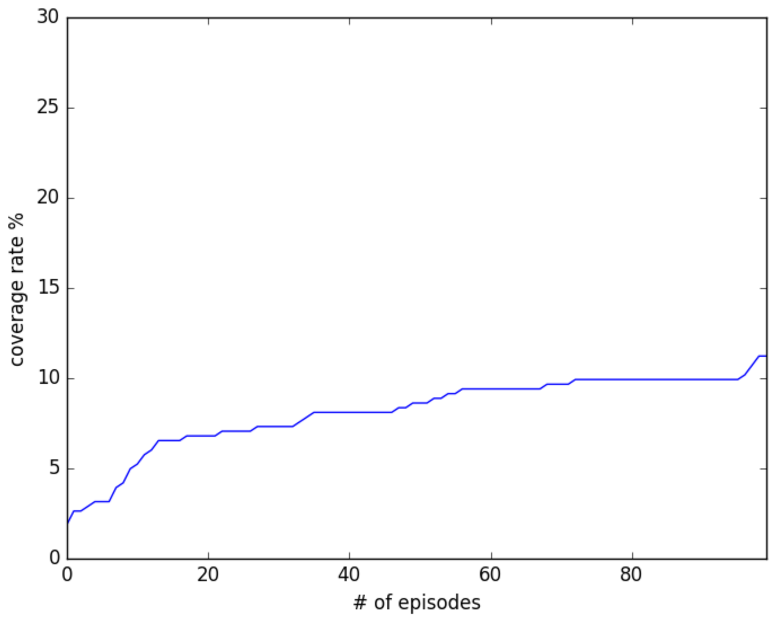
	Mean	**Std dev**	**Max**	**Min**
Success Rate	88.60%	9.85%	100.00%	60.00%
Coverage Rate	10.73%	0.86%	12.76%	8.85%

The mean success rate is far better than Greedy Driver with much less standard deviation. The minimum is 60% which is also much better. The below is a graph of one experiment.



Even though Greedy2 Driver uses the same greedy approach, initial Q-value change made such a big impact. It is exploiting and exploring at the same time. It stops exploring a state when it hit reward 2 by following both waypoint and the traffic light for that state.

The mean coverage rate is a bit less than Greedy Driver with less standard deviation. The difference of the mean coverage rate between Greedy Driver and Greedy2 Driver is within two standard deviation. They are not significant.



All the report and image files are under 'report/greedy2' folder.

To run the greedy2 driver, the following command is used:

```
python smartcab/agent.py greedy2 100 100 10
```

Having done the Greedy2 Driver experiment, I do feel this is a bit cheating. Who learned to set the initial value to 1.5? It's not the driver but me. In real life scenario, how often do we get such a lucky insight?

Smart Driver

The driver should be able to learn even if the initial Q value is zero.

We need a smarter driver that explores accessible (state, action) pairs as many as possible at early stage of learning (in the earlier episodes), learns traffic rules and rewards, and at later episodes, exploit that experience.

In other words, we don't really care how bad the performance is at earlier episodes as long as we make much less mistakes at later episodes. This is also why I'm measuring the performance of each experiment by the mean success rate of last 10 episodes.

Smart Driver explores as long as there is some (state, action) pair that it can access but has not done so. If there is no (state, action) pair that it can access immediately then switch to exploit the learned experience from that state and related actions.

Smart Driver uses the coverage information by exploring (state, pair) which has no coverage.

```
    def explore(self, state):
        return random.choice([action for action in self.coverage[state] if
self.coverage[state][action]==0])
```

As the coverage for each (state, pair) increases, the experience for that pair increases. Therefore, we can reduce the learning rate (alpha) for that (state, pair) and let it converge. As such, the formula for alpha calculation is no longer based on the time but it is based on the coverage at (state, action).

$$\alpha = \frac{1}{1 + coverage(state, action)}$$

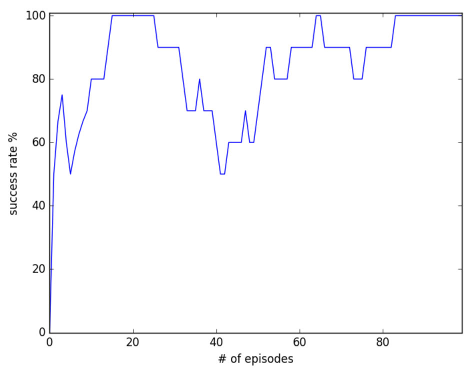
With this, the initial alpha will be 1 and learns the reward very quickly as the coverage (experience) increases the learning rate decreases at that (state, action). This way, we do not have to manually specify the initial Q to some value (like in the case of Greedy2 Driver) as the car will set it to the first reward for that (state, action).

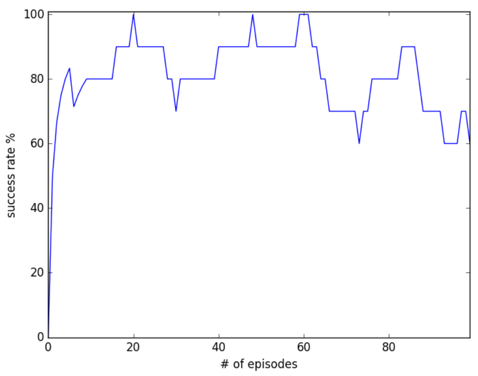
The outcome of the 100 experiments for Smart Driver is as follows:

	Mean	**Std dev**	**Max**	**Min**
Success Rate	87.20%	9.86%	100.00%	60.00%
Coverage Rate	10.76%	1.04%	13.28%	7.29%

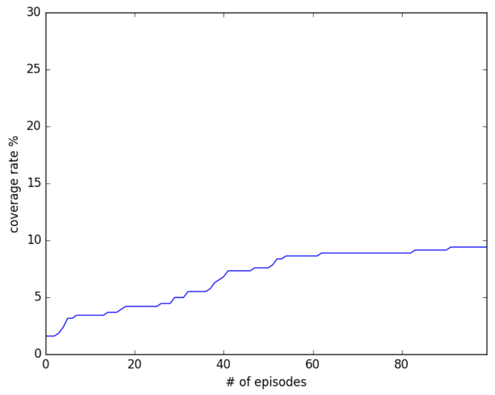
Comparing with Greedy2 Driver, the success rate and the coverage rate are almost the same.

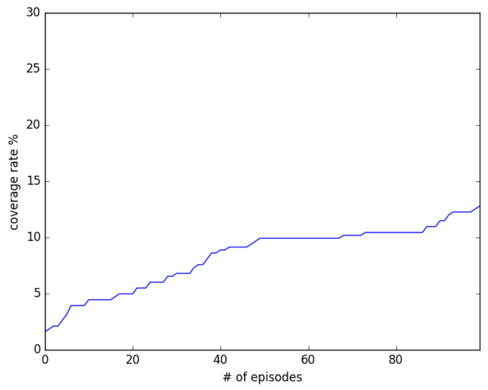
The following two charts compares the best success rate (100%) episode (on the left side) and the worst success rate (60%) episode (on the right side).





The following two charts are from the same episodes (above) but for the coverage rates. I noticed that the driver on the right side is still learning new states in the last 10 episodes. The left driver is more exploiting in the last 10 episodes.





To run the smart driver, the following command is used:

```
python smartcab/agent.py smart 100 100 10
```

Smart2 Driver

So, why the Smart driver on the right side chart still learning new states at such later stage? The details in the report/smart/nnn.csv file shows that the driver encounters other cars at the intersections. As the driver has not learned such (state, action) pairs before, therefore it has to explore it.

Suppose this is a real auto-driving car in public road, it should probably stop when it encounters a state that it has not learned previously.

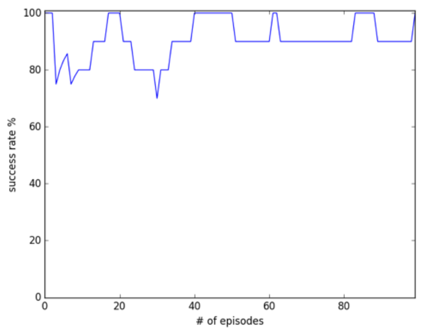
In the last 10 episodes, the car should stop exploring and simply exploit, treating the last 10 episodes as the validation period. When it encounters a state that it has not learned before in the validation period, it should do nothing and wait for the situation to change.

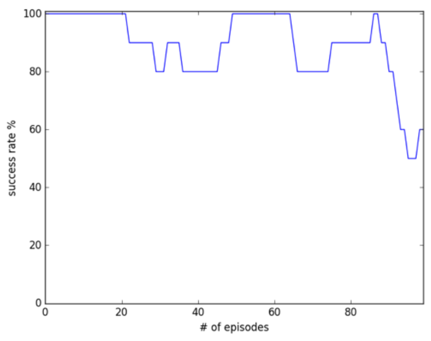
The outcome of the 100 experiments for Smart2 Driver is as follows:

	Mean	**Std dev**	**Max**	**Min**
Success Rate	87.30%	9.93%	100.00%	60.00%
Coverage Rate	10.90%	0.94%	12.76%	8.59%

The difference between Smart Driver and Smart2 Driver is not significant. However, Smart2 Driver is probably safer car.

The following two charts compares the best success rate (100%) episode (on the left side) and the worst success rate (60%) episode (on the right side). As seen with Smart Driver, Smart2 Driver occasionally performs bad (60%).





To run the smart2 driver, the following command is used:

```
python smartcab/agent.py smart2 100 100 10
```

Smart3 Driver

By looking at the details of both Smart Driver and Smart2 Driver, I found that the driver is optimally driving most of the time in the last 10 episodes of in every experiment. However, just following the waypoints and the traffic rules does not always get the car to the destination within the time limit. For example, if the waypoint is 'forward' and the traffic light is 'red', the driver waits until the traffic light changes to 'green'. However, if this happens a few times when the time limit is almost zero, it may not be the best action to take.

In fact, just waiting in such case is like being stuck with local maxima. So, I've actually tried some kind of randomized annealing for when the best action is None (waiting) and the deadline is getting closer to zero (higher epsilon to allow randomization to happen). It is implemented in Smart3 Driver. However, this does not prove to be successful consistently as we are dealing a very specific case and it is kind of over-fitting to try handling such case.

The outcome of the 100 experiments for Smart3 Driver is as follows:

	Mean	**Std dev**	**Max**	**Min**
Success Rate	79.70%	12.91%	100.00%	30.00%
Coverage Rate	10.84%	0.97%	13.28%	8.85%

I believe this is where the route planner needs to be improved rather than making the driver logic more complicated than necessary.

To run the smart3 driver, the following command is used:

```
python smartcab/agent.py smart3 100 100 10
```

Conclusion

In the end, the best driver is Smart2 Driver. It learns to follow the traffic lights and the waypoints effectively by exploring as early as possible and then exploits the experience to reach the destination. Over multiple episodes, the driver gets smarter and smarter to be able to optimally handle the states as far as it has encounter them before. Success rates improves over multiple episodes making it more likely to reach the destination without getting any penalty.

When Smart Driver (not Smart2 Driver) encounters unknown state at later episodes, it has to explore that (state, action) pairs which may not always be optimal at that time where Q-learning can not help as it does not know all the rewards and the transition matrix up front. This behavior can be dangerous in public roads.

I treated the last 10 episodes of each experiment as the validation period and if the car encounters unlearned state in that period, it simply waits instead of explore which is a safety feature (in real life, if an auto driver encounters an unknown state, it should not randomly act but wait for the state to change so that it can handle). As such, Smart2 Driver is safer than Smart Driver.

Smart2 Driver may not always reach to the destination within the time limit. Especially when it ways for red signal while the waypoint says 'forward'. If this happens when deadline is getting closer to zero, it may not be an optimal action to take (i.e. it may be better to take another route).

However, this is where the route planner's navigation logic needs to be improved rather than the driver's learning ability.