# Project 3: Markov Decision Processes

## The Selected Problems

### The Smaller Problem

For the smaller Markov Decision Process problem, I chose to modify a simplistic Cereal Loyalty Example outlined by J.E Beasly, a professor at the Imperial College of London ([src](http://people.brunel.ac.uk/~mastjjb/jeb/or/markov.html)). Aside from a recent nostalgia kick on Life cereal, I found this to be an interesting problem for several reasons. Firstly, it is a very understandable problem, with four states (purchaser of Crispy, Crunchy, Mushy, or Scrunchy cereal). The problem being understandable means that I can look at the code and the outcome of the MDP, and make connections between the output, input, action, and transitions simply. The transition probabilities include an example of absorbing states – a person who purchases Crispy or Crunchy cereal will repurchase that cereal with a probability of 100% (once reached, the state will not be left). The transition probabilities for the Mushy and Scrunchy cereals complement each other well, with values for Crispy, Crunchy, Mushy and Scrunchy at 0.45, 0.4, 0.05 and 0.1 and 0.1, 0.2, 0.3 and 0.4 respectively (Mushy purchasers are more likely to try Crispy of Mushy in a new purchase process, whereas Scrunchy purchases are more likely to purchase Mushy or Scrunchy)

When I went to implement the transition probabilities, this problem made me think about how to translate those probabilities to be associated with particular action – I invented actions for cereal advertisements as if the decision maker is a cereal eater exposed to different advertisements – “If I am offered different discounts for Scrunchy what is the probability that I will buy {cereal} next time I’m out shopping?”

The rewards for this problem are relative to the “decision maker,” or the purchaser of the product. Various methods of advertisements may make Scrunchy cereal more rewarding.

The initial state in this problem is given by “current market shares” for each cereal ([0.2, 0.3, 0.3, 0.2], for Crispy, Crunchy, Mushy, and Scrunchy).

In the real world, an analyst might look at this problem and play around with how a marketing campaign could impact transition probabilities for example, and then make conclusions about how much could be gained from said marketing campaign. The fact that this data is understandable and contains several interesting transition examples (absorbing state & and complimentary transient states) makes this an interesting MDP.

### The Larger Problem

For the larger problem set, I decided to use the Taxi-V3 problem provided online by Dietterich ([src](https://gym.openai.com/envs/Taxi-v3/)). In this world, you are a taxi driver who must pick up and drop off passengers in a grid world. Unlike the smaller problem described above, there are *many* states in the taxi grid world (500 discrete states vs the 4 in the above example). Rather than picking the best cereal, the taxi world problem involves finding a passenger and dropping the passenger off at a *specified* location in order to receive an award – negative rewards come from not just picking the wrong cereal as in the above example, but from driving too far or illegally picking up or dropping off a passenger. Rewards can be immediate in the cereal example, but there are delayed rewards in this example. The contrast between this problem and the simplistic cereal problem complement each other nicely, making this an interesting problem to choose for the larger problem set.

## Value Iteration

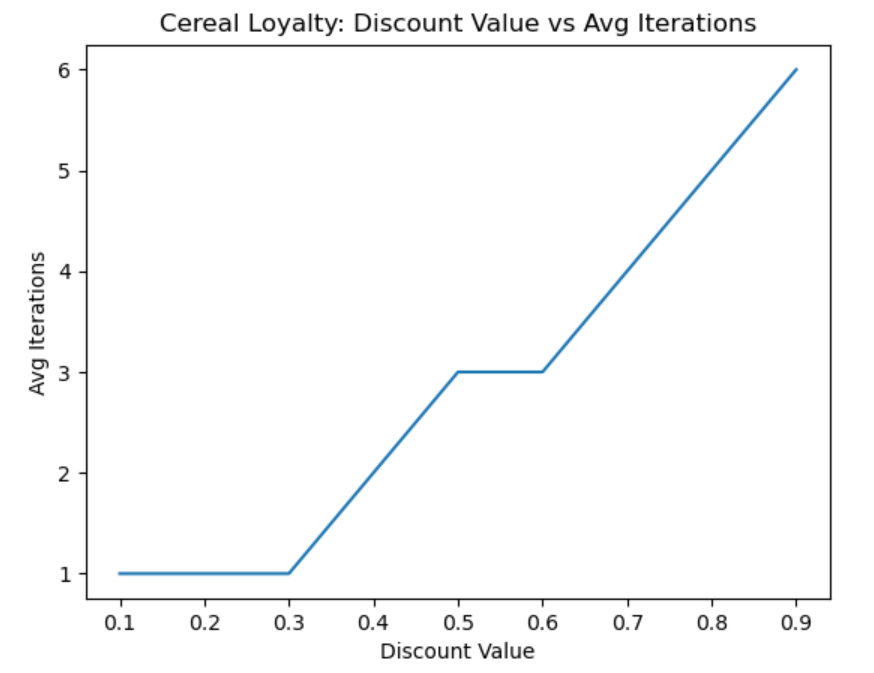
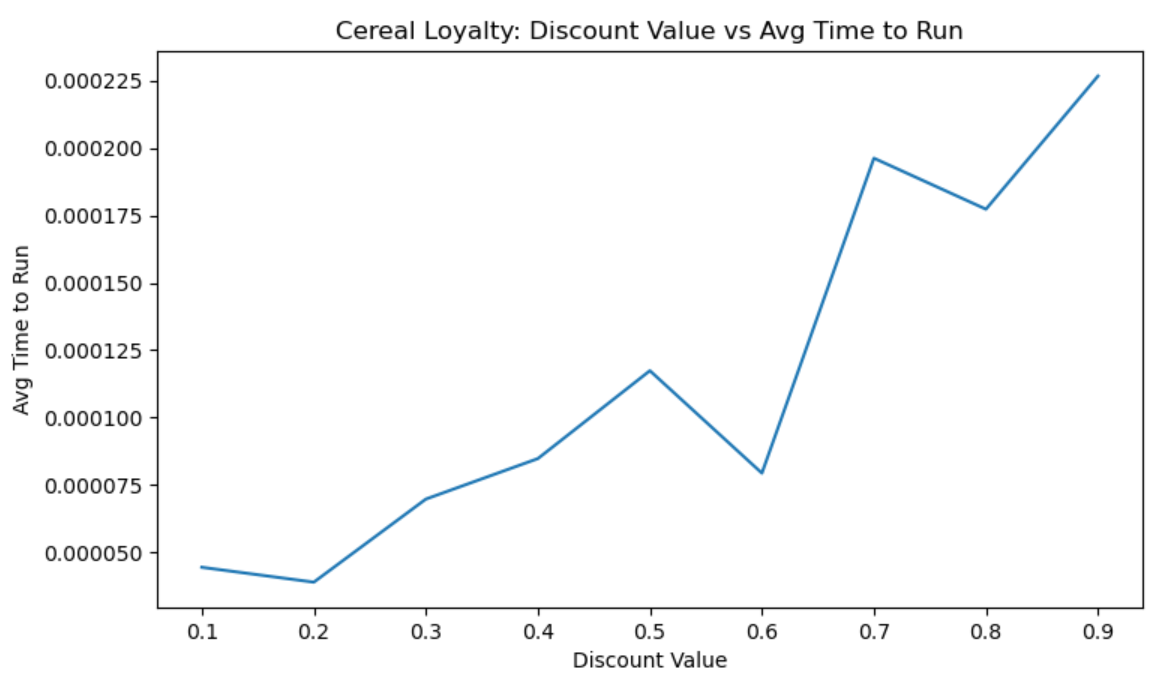
Value Iteration finds the best value for a problem, and from the best value can find the optimal policy. For the value iteration algorithm, I used the Python Markov Decision Process Toolbox ([src](https://pymdptoolbox.readthedocs.io/en/latest/)). The algorithm runs until an “epsilon-optimal policy is found or after a specified number (max\_iter) of iterations.” (my definition of convergence for value iteration for this assignment) (src). I set max\_iter to a very large number (src) in order to measure iterations until convergence OR until the default max iterations determined by the provided discount, epsilon, and state, where a smaller discount and a larger epsilon result in a smaller number of max iterations (src).

### The Cereal Problem

For the cereal problem, I set epsilon to a value of 0.5 - the maximum difference between the value functions of each iteration must fall below that number before the algorithm is considered “converged.” As a very simple problem, I felt comfortable raising the epsilon from the default 0.1 – I did not expect variance to play a huge role in the finding of the optimal policy, but experimented locally to verify. Larger epsilon values increased performance in terms of time, but not significantly.

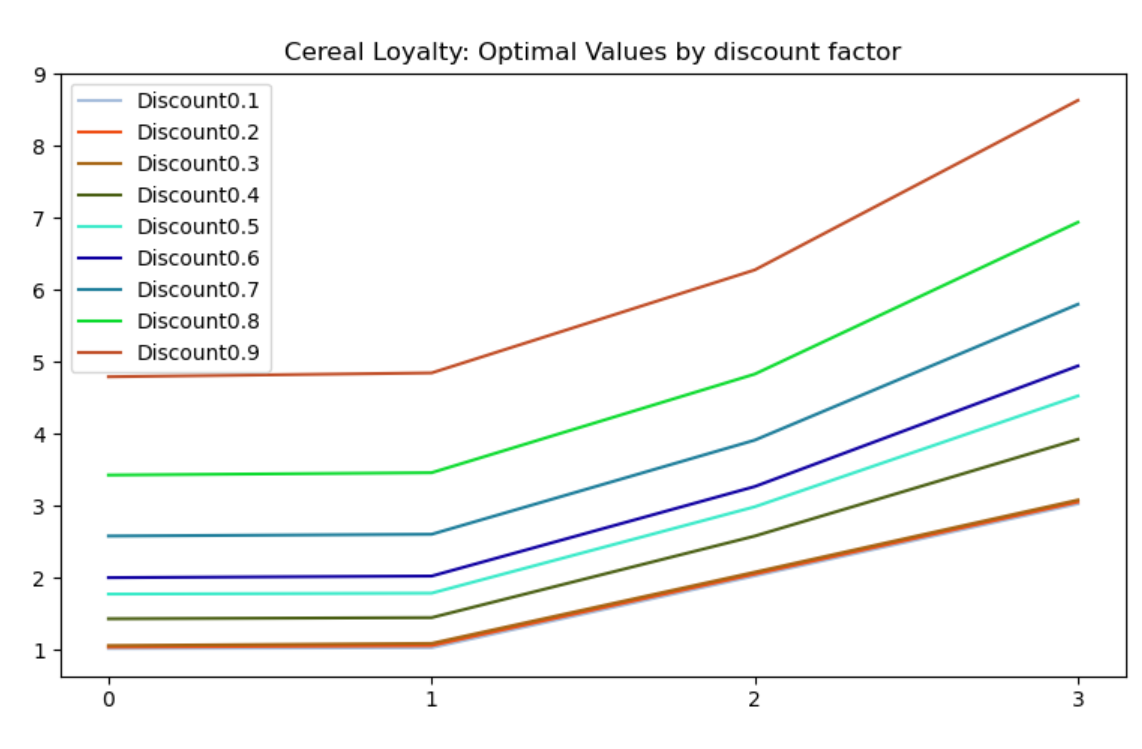
The initial state of this problem is as outlined in the problem section above, [0.2, 0.3, 0.3, 0.2].

I ran the value iteration algorithm 1,000 times to find an average (mean) run time & iteration count for various discount values (***figure 1***).



**Figure 1** – Average time to run and average Iterations until convergence for different discount values used when solving the Cereal problem.

Discount values of 0.4 and greater took more than 1 (up to 6) iterations until convergence. The average time to run also increased after a discount value of 0.2. The optimal value function changes based on the discount value but follows a similar trend (see ***figure 2***). This increase can be explained because as the discount factor increased, the number of iterations increased, and the total expected sum of rewards is expected to be greater as well.



**Figure 2** –Values by discount factor.

Epsilon-optimal policies were found for all of the discount factors pretty consistently, except for discount factors below 0.3 where on occasion where max iteration was reached - however even in those cases where max iterations was reach, the same optimal policy was found. There is not a 1:1 relationship between policies and values – a suboptimal value function can still be associated with an optimal policy.

The found optimal policy is (0, 0, 2, 2).

To validate that the policy is affected by a perceived increase in Scrunchy value (see rewards), I ran value iteration with a “rewards experiment” where Scrunchy’s reward was modified to be 0. The optimal policy in this experiment turned out to be (0, 0, 0, 0), showing that the increased reward of purchasing Scrunchy cereal (affected by the marketing campaigns actions) did in fact influence the optimal policy.

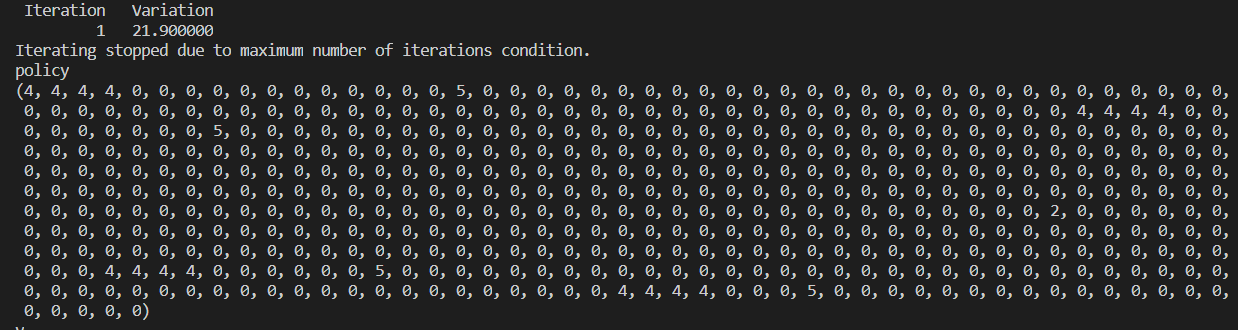
As a Scrunchy marketing analyst, manipulating transition probabilities and perceived rewards of the Value Iteration problem could give me values insight into what sort of milestones I need to aim for when creating new marketing campaigns. I could also decide what marketing campaigns to avoid or pursue, based on customer behavior predicted with MDP.

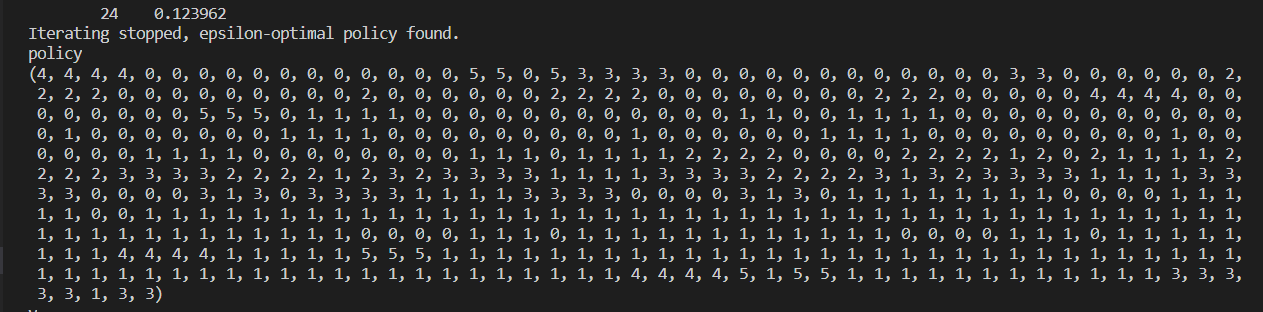
### Taxi Problem

The grid world’s state is encoded to a single number between 0-499 based on taxi location, customer location and destination. Transition probabilities for each actions are 1, meaning a move happens with absolute certainty when it’s selected.

The initial state of this problem (where the taxi cab is located, where the passenger is located, and where the passenger wants to go) is generated randomly at the beginning of each execution of this problem. The practice of averaging performance scores for this analysis is therefore extra important – the data shown below is averaged over the course of 1,000 runs (One problem generated for each of the 1,000 runs link).

I ran the taxi problem with various discount values and an epsilon set to 0.5. For smaller discount values, the algorithm would fail to optimize values other than the “obvious” wins of picking up a customer in a correct location (4) or dropping them off in a proper location (5) (***figure 3***)

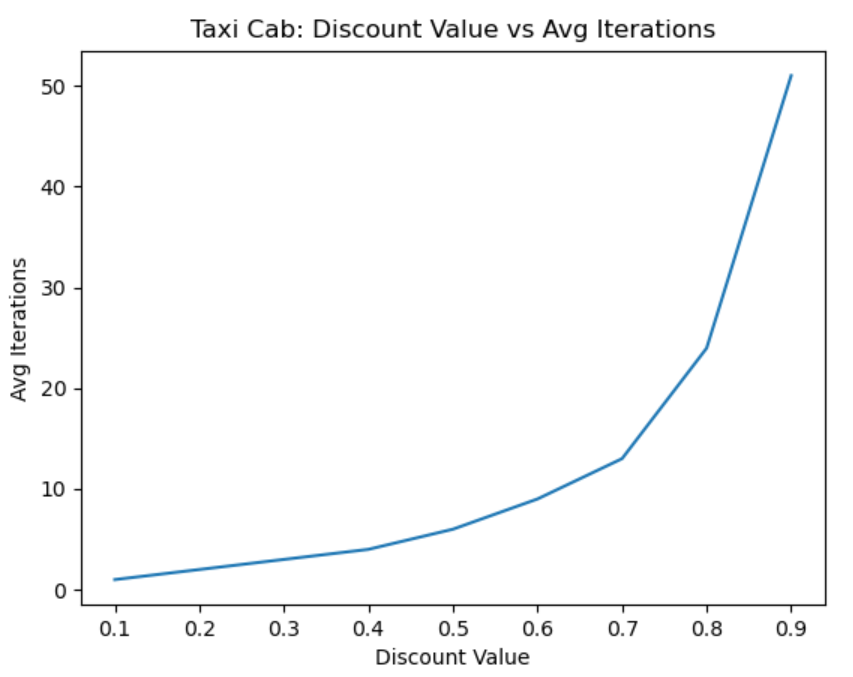
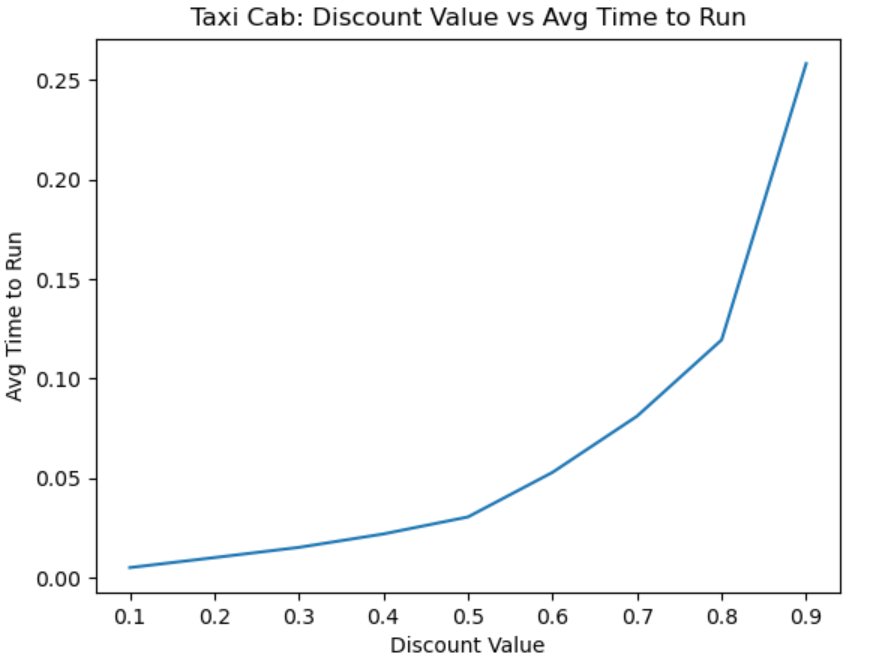




**Figure 3** – The above policies show what the found optimum action is for each given state. **Top -** With smaller discount factors (0.1 in the above example), the value iteration algorithm only identifies “obvious” moves, such as picking up passengers in correct locations (4) or dropping passengers off in a proper location (5). Everything else remains at the default action of 0. **Bottom –** With higher discount factors (0.8 in this example), epsilon-optimal policy is found and the “harder to identify but better” actions identified and replace the default actions of 0.

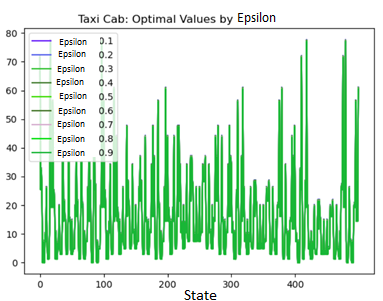
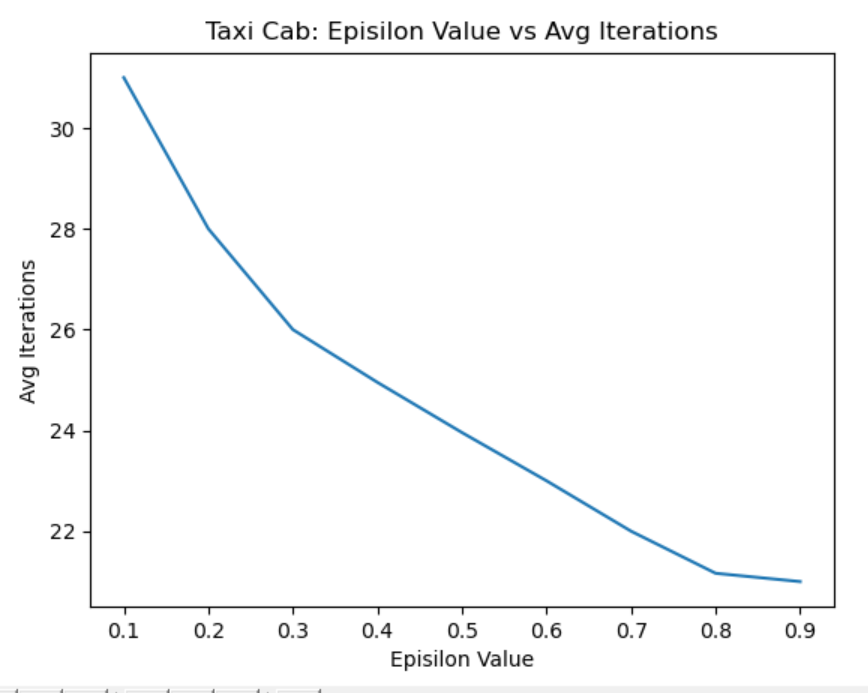
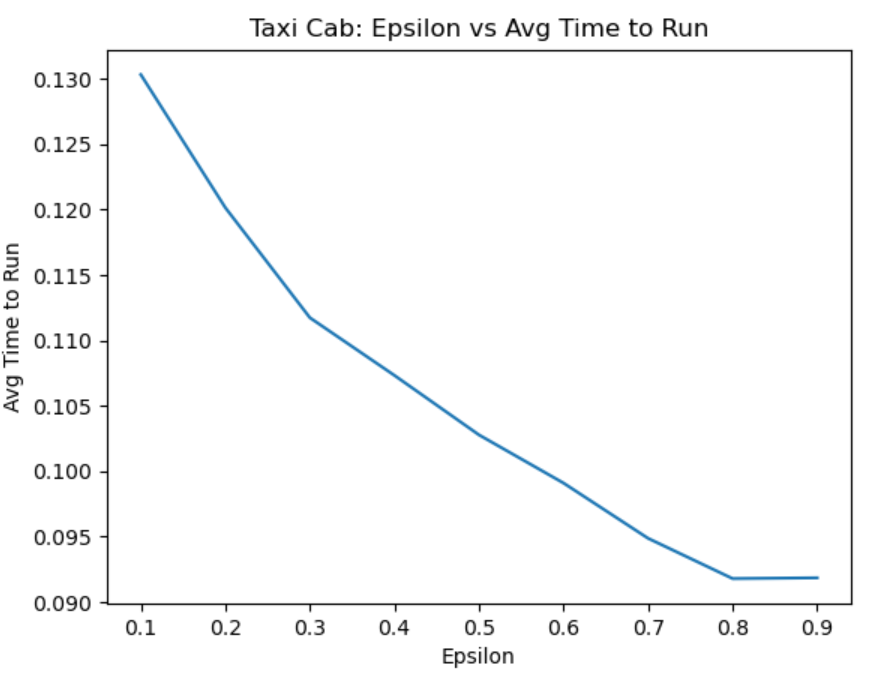
On consecutive runs, when a discount factor of 0.8 was reached, the terminating condition for the execution was not due to the max number of iterations, but rather the discovery of an optimal value function. You can see this visually with the inflection points in ***figure 4.*** It’s at this point that an epsilon optimal policy was found.

As anticipated due to the larger state size, the taxi cab problem took considerably longer with many more iterations to complete than the cereal loyalty problem (See ***figure 4***). This problem also showed a cleaner curve vs the less consistent growth seen in ***figure 1***. This can be explained because the time increments in ***figure 1*** are closer (range of 0.0002 for cereal vs 0.25 for Taxi)– therefore slight deviations from a “smoother” curve line are more visible.



**Figure 4** – The average time to run and the average number of iterations until termination for the Taxi Cab Value Iteration

Next, I experimented with different epsilon values, maintaining a discount factor of 0.8. As epsilon grew, the time to run decreased. The average number of iterations decreased at the same rate. The optimal values however remained consistent (See ***figure 5***) – the optimal value function was always found, and there for the epsilon optimal policy was found as well. When solving this problem, it goes to reason that a largest epsilon value should be paired with a small discount value to optimize the time to run.



**Figure 5 –** As epsilon grows, time to run (Left) and iterations (middle) decrease – the optimum values (right) remained consistent for this problem across epsilons.

## Policy Iteration

I used [pymdptoolbox’s implementation](https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html#mdptoolbox.mdp.PolicyIteration) of the Policy Iteration algorithm. I did not provide an initial policy, meaning the first policy will be fully defaulted with 0s. Like with Value Iteration, I set the max\_iter to a large number so that I could gain more insight into how long it could take to find the optimal policy. I experiment with different discount values below.

Convergence for the Policy Iteration algorithm happened when either a max iteration was hit, or the variation between the current policy and the previous policy is less than epsilon, with some consideration for the provided discount factor.

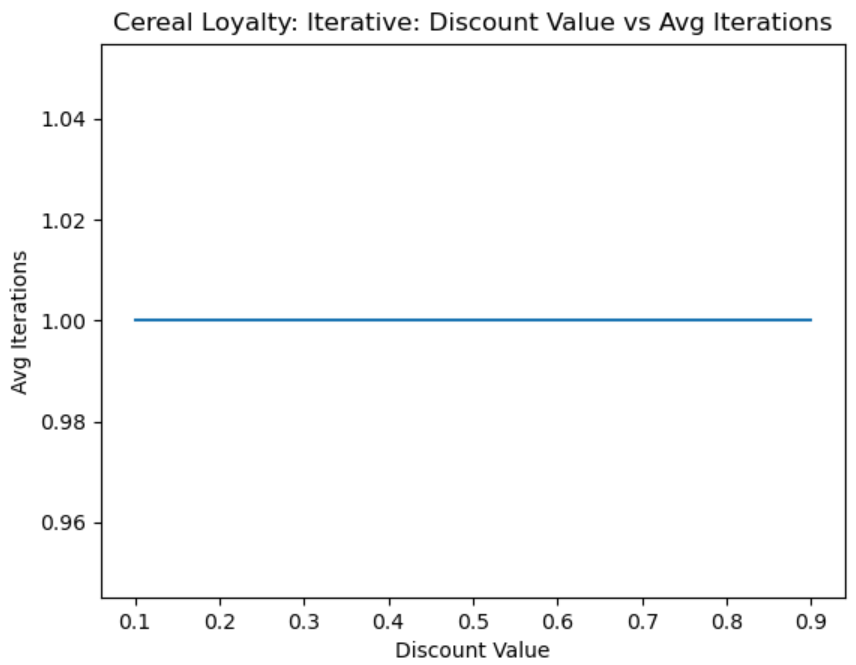
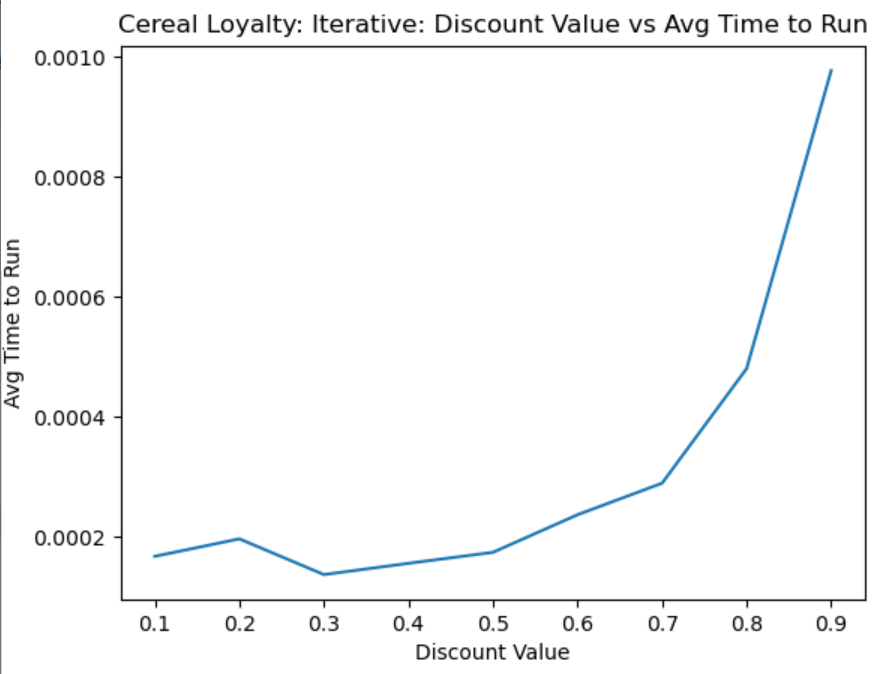
# ensure |Vn - Vpolicy| < epsilon

variation < ((1 - self.discount) / self.discount) \* epsilon

**Figure 6 –** The terminating condition for Policy Iteration, if max iterations have not been reached.

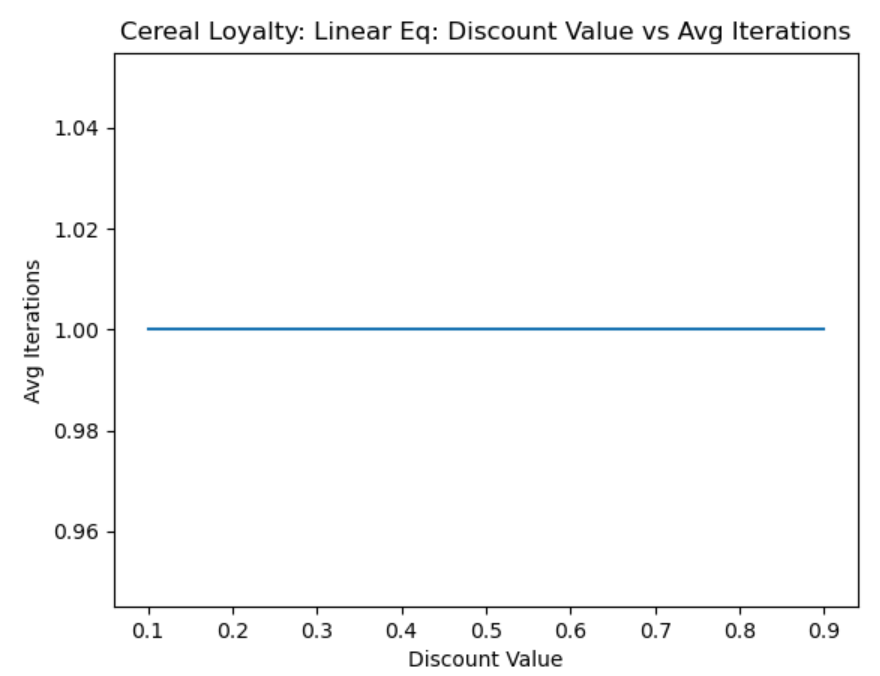
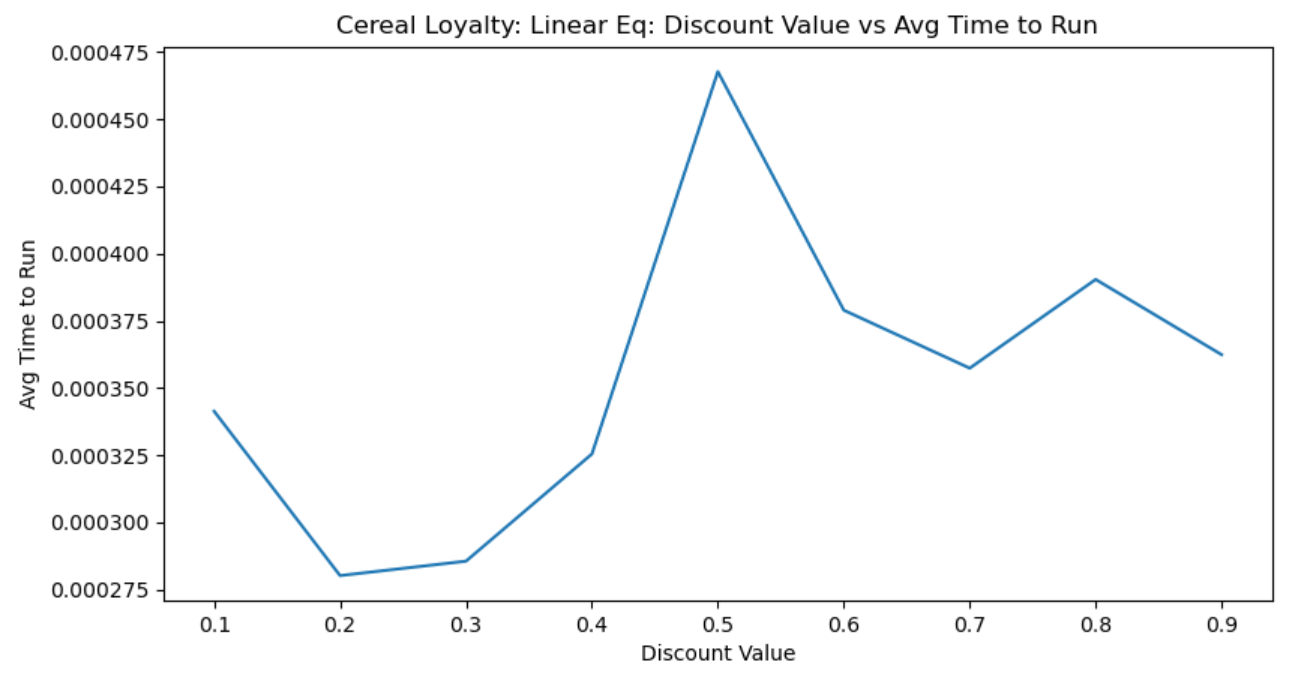
### The Cereal Problem

For the cereal problem I ran 1,000 iterations for discount values between 0.1 and 0.9. I first ran this code with an iterative eval\_type. Regardless of the discount value, the optimal policy (0, 0, 2, 2) was found after a single iteration. With larger discount values, the average time to run increased. This method, while fast, took longer to solve in comparison with value iteration [0.0002 – 0.0010 for policy iteration vs 0.00005-0.0000225 for value iteration] (***figure 7, figure 1***).



**Figure 7**

I next ran the same code but with a linear equation eval\_type. The same number of iterations were required for convergence (1) and the optimal policy was found (0, 0, 2, 2). The time to solve was faster with linear equations vs the interative approach. Beause this is a more simplistic problem, representing it and solving it mathematically is easy to do, explaining the increased performance in terms of time vs the iterative solver.



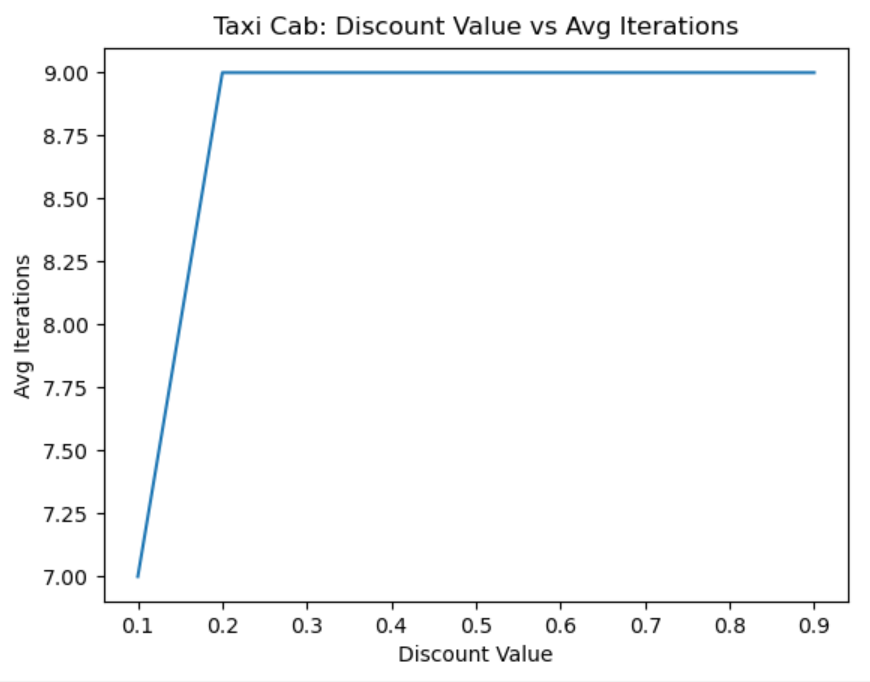
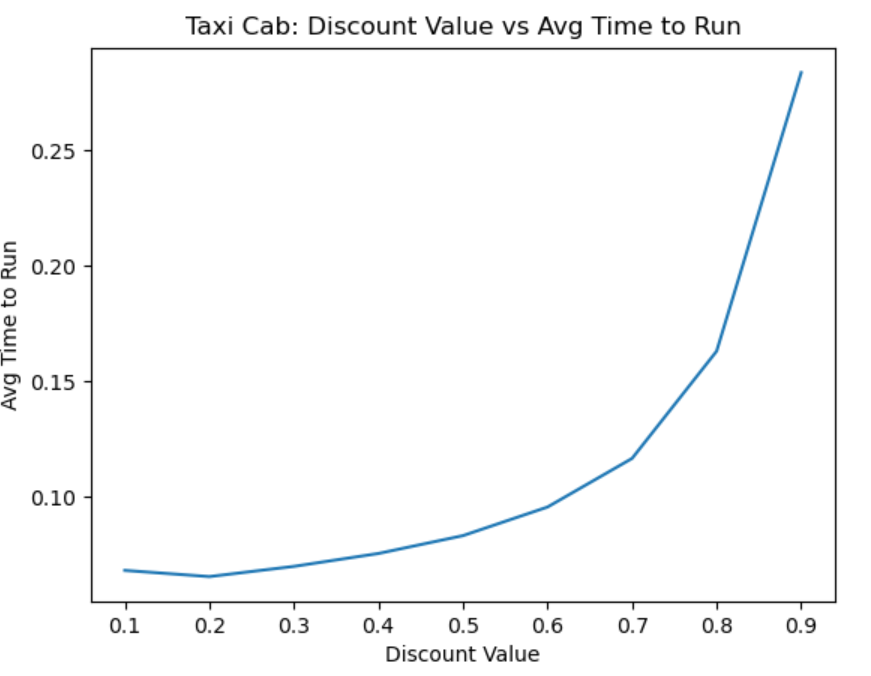
**Figure 8**

Even the lower discount value however runs took more time to run policy iteration than the higher discount value iterations of the value iteration. This goes to show that value iteration is the preferred way of solving this problem. I expand on why in the Part 1 Conclusion below.

### The Taxi Problem

I ran the same policy iteration experiments with the taxi problem as with the cereal loyalty problem. Various discount values were explored, and I used the iterative solver, and then the linear solver for the problem.

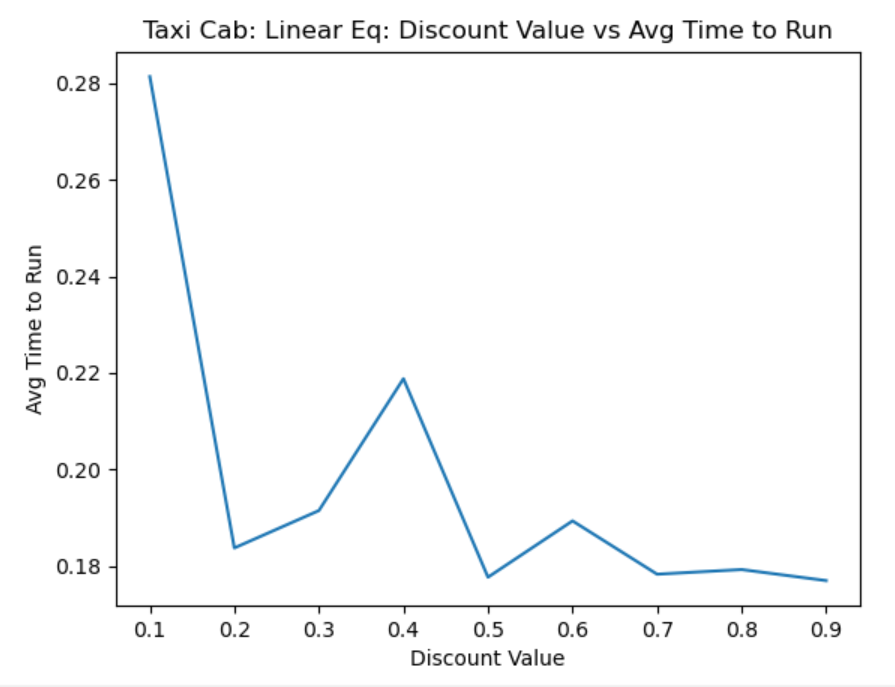
It took approximately the same amount of time to solve the taxi problem with policy iteration as with Value Iteration; however, it took much fewer iterations (0-9 for Policy Iteration vs 0-50 for Value Iteration) (***figure 9***). The optimal policy was also found with a lower discount value – 0.2 (below 0.2 the max iteration was reached before the epsilon optimal policy was found), vs 0.8 for the value iteration algorithm.



**Figure 9**

In ***figure 10,*** I re-ran the taxi cab problem with a linear equation solver. I took slightly longer for Policy Iteration to solve the problem with the linear equation solver. This follows with the increase in complexity of the problem, in contrast with the simpler Cereal Loyalty problem where the linear equation solver took less time than the iterative policy iteration.

The linear equation solver was able to find the optimal policy with a lower discount value, however. But is this trade off worth the increased time? In the case of complex problem I’d argue no.



**Figure 10**

## Part 1 Conclusion – Value Iteration vs Policy Iteration

In both value and policy iteration, it was shown that finding the appropriate discount value and epsilon values for a problem is important to optimizing the performance in terms of time and computational complexity. Smaller discount values and larger epsilons are able to solve problems faster and in fewer iterations, so when time or computational complexity is a limiting resource, the smallest discount value and largest epsilon value should be found for a given problem.

Both algorithms can be tweaked in other ways for optimal performance as well – In the case of policy iteration for example, a linear equation solver vs an iterative solver is preferable for less complex problems that can be solved simply with algebra.

Polity iteration takes more time per iteration to execute but can solve problems in fewer iterations. This makes policy iteration preferable for larger, more complex problems, whereas value iteration should be used for less complex problems – More iterations may be required, but the time for each iteration will be shorter.

## Reinforcement Learning Algorithm

I decided to use [Python Markov Decision Process Toolbox](https://pymdptoolbox.readthedocs.io/en/latest/index.html)’s implementation of QLearning for my reinforcement learning algorithm ([src](https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html" \l "mdptoolbox.mdp.QLearning)).

The QLearning algorithm implementation is greedy, and selects the actions that provided the best value. Below, you’ll see analysis of the QLearning algorithm with various discount values and number of iterations. The used implementation of QLearning does not allow the direct customization of the learning rate (also know as alpha) by default. Instead, the learning rate decays based on the provided discount and the delta between Q values (***figure 11***).

# Updating the value of Q

delta = r + self.discount \* self.Q[s\_new, :].max() - self.Q[s, a]

dQ = (1 / \_math.sqrt(n + 2)) \* delta

self.Q[s, a] = self.Q[s, a] + dQ

**Figure 11 –** Rather than offering an easy configuration value for learning rate, the used implementation of the QLearning algorithm updates Q with a decaying rate over time – each iteration, the Q value will have a slower learning rate

I used the provided learning rate while tweaking the discount and n\_iteration parameters, but then customized this code to explore exploration vs exploitation.

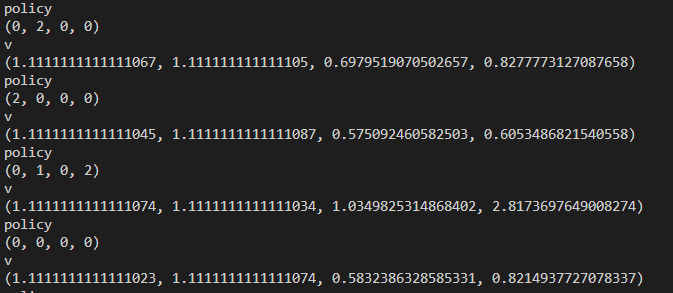
Q = self.Q[s, a] \* learning\_rate + self.Q[s\_new, :].max() \* (1-learning\_rate)

**Figure 12 –** Custom learning rate code, where when 1 is provided, no exploration happens, and when 0 is provided, no exploitation happens.

Values in each visualization below are averaged over the course of 1,000 iterations.

### Cereal Problem

QLearning struggled to consistency find the optimal policy for this problem.



**Figure 13 –** Different policies found for different runs of the QLearning algorithm, each with a consistent discount value, learning rate, and n\_iterations.

#### Custom Learning Rates

I re-ran the QLearning algorithm with a discount values of x, n\_iterations, and various learning rates enabled with custom code.

Now pick your favorite reinforcement learning algorithm and use it to solve the two MDPs. How does it perform, especially in comparison to the cases above where you knew the model, rewards, and so on? What exploration strategies did you choose? Did some work better than others?