ECE:517
Project #1:Report:
Authors: Ronald Randolph, John Geissberger Jr.

Introduction

For this problem, the main objective is getting ice cream as fast as humanly possible. However, life is not always easy and this problem is no different. The path leading to obtaining ice cream as fast as possible is full of impediments. Here is a simple breakdown of the problem at hand: An individual seeks to obtain ice cream as fast as they possibly can and in order to obtain ice cream one must not only go to the ice cream store, but they must also find a parking spot next to the ice cream store. The choice of the parking spot is critical, because the objective is based on time. Therefore, if one parks too far away from the ice cream store one may still get ice cream, but not as fast as they could have if they parked closer. One simple solution might be to park closer. However, this ice cream store is a busy location such that each parking spot has a probability that represents the likelihood that it is taken. Furthermore, since everyone else also wants to get ice cream as fast as they can the spots closest to the ice cream have a higher likelihood of being taken compared to the spots that are farther away.

In order for one to achieve their goal of obtaining ice cream as fast as possible, one must construct a strategy that takes into account factors such as, the probability of a parking spot being taken, the time it takes to walk to the store, and the finite number of parking spots available to park in. Additionally, the strategy constructed must be optimal in the sense that it will determine when and where an individual parks such that the time associated with getting ice cream is the minimum amount of time possible compared to all other strategies.

Dynamic Programming

In order to solve the problem described above, one could choose to utilize a dynamic programming strategy in order to produce a solution. One viable approach to producing a solution to the problem stated above is to run through all of the different possible ways one could park and obtain ice cream, compute the time associated with those different ways, and finally chose the path in which the time is minimized. This type

of strategy could be characterized as a “brute force” strategy. However, one of the issues with this type of strategy is that it can become computationally inefficient depending on how one implements code to produce a solution. One way to overcome this issue of computational inefficiency is to utilize dynamic programming.

Dynamic programming can be described as a “careful brute force” strategy because, in most cases, it can reduce the time complexity associated with a “brute force” strategy. One of the ways that dynamic programming reduces the time complexity is by utilizing memoization, which memorizes values which will be needed for later computation. Memorization can allow for a reduction in time complexity when the same computation will need to be computed numerous times. Instead of calculating a value numerous times, why not calculate it once, store it, and access it later? Memoization can produce profound effects on the reduction of time complexity, which is why it is an integral aspect of reinforcement learning.

Although dynamic programming is a crucial element of reinforcement learning, it still has limitations within the field. For instance, one of the important assumptions associated with dynamic programming is that a, “perfect model of the environment” associated with the problem must be provided. This assumption is rarely met and in many cases, a “perfect model of the environment” cannot be produced. As a result, other methods must be implemented in order to produce a solution. Luckily, due to numerous assumptions and simplifications in this problem, a “perfect model of the environment” can be provided, which allows for the utilization of dynamic programming. Specifically, one can determine that some values, the transition probabilities, are constantly required for computation and are constantly required to be computed themselves. Therefore, one can implement a memoization technique that allows for the storage of these values, which will result in a reduction in computation time.

This project calls for the use of value iteration. Value iteration is a dynamic programming process that combines policy improvement and evaluation in a similar manner as policy iteration. However, the biggest difference between these two DP approaches is that value-iteration truncates the policy evaluation steps such that it only needs to sweep through the value function once. This often results in a faster convergence.

Markov Decision Process

Many reinforcement learning problems can be framed as Markov Decision Processes, or MDP's, which have many benefits. One of them being that MDP's can be solved through dynamic programming. Furthermore, the MDP framework can provide an elegant abstraction for the reinforcement learning problem. As a result, this problem was framed as an MDP. First, one must be familiar with what makes up an MDP. An MDP can be defined as a 5-tuple, as seen in the figure below.

$$\mathcal{M} = \langle S, A, R, P \rangle$$

The elements of the 5-tuple are, States, Actions, Rewards, and a Transition Probability Model. When one considers the problem at hand, these elements could be defined as following: States are all M possible parking spots and whether or not they are free or taken. Actions are either drive, park, or wait. Rewards are associated with the time taken at each step and whether or not one obtained ice cream. Lastly, the transition probability model was defined according to the instructions given in the project. These definitions can be further observed below.

States - S

1 NT	2 NT	3 NT	….	M-1 NT	M NT
1 T	2 T	3 T	….	M-1 T	M T

T = Taken, NT = Not Taken

Actions - A

A(0) - Drive	A(1) - Wait	A(2) - Park

Rewards

r_No	Reward for not getting ice cream
r_Yes	Reward for successfully getting ice cream
t_Drive	Time taken to drive to the next spot
t_Wait	Time taken to wait at a spot
t_Walk	Time taken to walk to the next spot

The transition model is determined through the provided argument, P_{same} which is supplied at runtime. This variable represents the likelihood of a spot staying open or being taken. In addition, additional variables were provided - T_{drive}, T_{walk}, T_{wait}, which represent the time associated with driving, walking, and waiting, respectively. All of these variables, and others, were treated as command line arguments into the program and were then utilized to produce a transition probability. The rewards were structured such that an immediate reward was provided for each action taken and final rewards were provided based off of the end result of obtaining ice cream. For instance, if the action 'Wait' was taken the reward associated with that action was $-T_{wait}$ or if the action 'Drive' was taken then the immediate reward was $-T_{drive}$. One case that required more attention was the case when the agent parked. If the agent parked and the space it parked in was not taken then a reward of 150 + ($-T_{walk}$*dist) was returned, where dist was equivalent to how many spots away the agent parked from the store. On the other hand, if the agent parked in a spot that was taken then it would receive a reward of -100, which was an attempt to control the agent's behavior such that it learned to never park in a spot that was taken. Lastly, if ice cream was not obtained then -100 was added to the sum of all of the immediate rewards prior.

Code Design and Data Structures

The included file, *iceCream.py*, contains the implementation designed to solve this problem through the structure of a MDP and the utilization of dynamic programming. For this project two DP separate approaches were taken in this project – policy iteration and value iteration. By default, this program will optimize using the value iteration process. This can be changed by changing *def_process* to 0 or 1 in the *main()* function before execution.

The value iteration portion of the code is broken down into three main functions. The first function, *value_iteration()*, evaluates the policy until convergence. For each iteration through the current value function, this function utilizes the second function, *prob()*, which returns the action-state value for a given action and state. The value for each state is then set to the maximum value returned from all actions in A(s). This process repeats until the value function reaches convergence. After convergence is reached, the value function is then passed to the final function, *output_policy()*, which greedily calculates a policy given a value function.

The policy iteration portion of the code is also broken down into three main functions. First, the function, *policy_iteration()*, updates the value function through the use of *probs()* in a similar manner to the function, *value_iteration()*. However, after initial convergence, it passes the value function to *policy_improvement()*. This function iterates through the current policy and greedily updates it based on the given value function. If there were any updates to the policy, the policy is declared unstable and is passed back to the previous function, *policy_iteration()*. This cycle continues until the policy is considered stable and optimal.

Both of these dynamic programming approaches utilize a common function, *prob()*. This function takes the following: the current action, current state, transition matrix, value function, and the discounting factor. Using this data and the equation below, the function calculates and returns the state-action value given a state and action.

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big]. \end{aligned}$$

Eq. (4.6) in “Reinforcement Learning: An Introduction,”

Additionally, the program utilizes a brief helper function, *initialize()*, which initializes the data structures for the state space, value function, policy, and transitional matrix. There were multiple structures utilized to contain the various collections of data. Many of these collections were represented through the use of arrays and matrices created and manipulated using the numpy external library. Both the state space and policy were stored in 1D arrays of length 2*M. The first M elements represent the taken parking spots and the second M elements represent the free or available parking spots. The value function is stored in a similar manner with the exception of an addition element to represent the value of the terminal state. As such the 1D array for the value function is of length 2*M+1. Finally, the transition matrix, which contains the probabilities of each space being available, is represented as a 1D array of size M. While many of these data structures could have easily been stored and represented in the form of multi-dimensional arrays, to simplify iteration - and improve code readability, the data for this project was stored in 1D arrays.

Correctness Results

For the example setup, the parameters and rewards were set as the following: The lot consist of 12 Parking Spaces. The probabilities for each spot initially being available is linearly spaced between P_{min}= 0.1 and P_{max}= 0.9. The probability that a spot remains taken/available after waiting is P_{same}= 0.9. The time taken for each action is T_{drive} = 1, T_{walk} = 3, T_{wait} = 10. Finally, the rewards for not getting ice cream and getting ice cream are R_{no} = -100 and R_{yes} = +150, respectively.

Optimal Value Function

	1	2	3	4	5	6	7	8	9	10	11	12	TS
T	53.081	51.465	49.396	46.798	43.579	39.608	34.745	28.833	21.749	13.500	4.413	-4.587	0
NT	117	120	123	126	129	132	135	138	141	144	147	150	0

Optimal Policy

	1	2	3	4	5	6	7	8	9	10	11	12
T	Drive	Drive	Drive	Drive	Drive	Drive	Drive	Drive	Drive	Drive	Drive	Wait
NT	Park	Park	Park	Park	Park	Park	Park	Park	Park	Park	Park	Park

T = Taken, NT = Not Taken

The value function is a function of states that gives a general estimate of how good or valuable it is to be in any given state. From observation of the optimal value function above, the states representing the taken parking spaces have a much smaller value than that of the free spaces. This is to be expected as each available space allows the agent to park and get ice cream. Whereas, each taken space does not. Furthermore, it can be seen that the spots further away from the ice cream shop have a higher value than that of the ones closer to the shop. The reason for this is that as the agent is at a further spot, it is more likely to find an open spot before reaching the shop. With each passing spot, this probability lessens, thus resulting in a lower potential for reward. Lastly, the final taken spot, 12, has a negative value. This is because each action at this state will more than likely result in a negative reward. If the agent parks or drives, it will immediately be given a reward of -100 for failing to get ice cream. As such, the only option the agent has is to wait in hopes that the spot will become available. Unfortunately, the penalty for waiting is -10 each time, and there is only a probability of 0.1 that the spot will become available.

As mentioned before, the free parking spots all have values of over 100. This is because even with the penalty for walking from the furthest spot, the agent will still be rewarded handsomely for getting ice cream. Another pattern to note is that the values of the free parking spaces increases as they get closer to the ice cream shop. This can be explained by the penalty for walking. After parking, the agent is given a reward of -3 for each spot it has to walk past to get to the shop. As such, the value of open space 12, is 150, which is the total reward for getting ice cream.

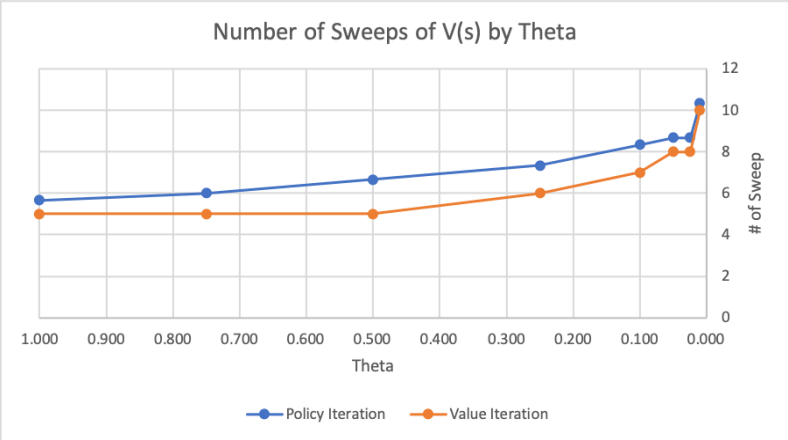
The optimal policy illustrates the most effective actions to take at any given state. From the diagram above it appears that the agent will consistently drive past all taken spots except the last one. The reason for this is because after driving past the last spot, the agent forfeits the opportunity to get ice cream and thus gets a reward of -100. As such, the agent decides that it is more advantageous to instead wait at the last parking space until it is available. Additionally, it decides to wait at the last spot instead of any previous ones because the probability that any one of them will become available after waiting is uniform throughout the spots. As such, the agent waits at the last spot to negate the penalty incurred from walking from further spots.

In regard to the free parking spaces, it is clear that whenever an available parking space is encountered the agent will immediately park in the spot. After successfully parking, the agent is given a reward of 150 minus the penalty for the distance walked. The value of this scenario greatly outweighs the probability of driving to another open spot. Of course, if the penalty for walking was increased, the agent would probably forgo parking in some of the more distant spaces.

Results Analysis

During the completion of this project three interesting questions came up. First, one may ask what happens when the reward value is transformed such that it now takes on a different form? Second, how will the agent's behavior change when the negative immediate time rewards associated with T_{wait} are taken out? Third, how will the agent's behavior change once the negative punishment associated with parking in a spot that is taken is reduced? Before these interesting questions were implemented some outcomes were assumed. For instance, one of the assumptions associated with the second question was that the learning agent would learn a strategy such that it would drive to the first spot in the lot and wait until the spot was open. In regards to the third question, it was assumed that the learning agent would learn a strategy such that it would drive to the first spot and just park, regardless of whether or not there was a car there or not. One can view the observations below in order to determine if these assumptions were accurate.

Because our code includes the implementations of both policy iteration and value iteration, our first exploration will be comparing the efficiency and effects of the two. In both processes, during the policy evaluation, the value function is repeatedly updated until the amount of change, Δ, is smaller than a predesignated value, Θ. The example setup does not designate a value for Θ, so the value within the implementation was defaulted to 0.1. The graph below illustrates the effect incurred as Θ is lowered towards 0.



As expected, as Θ is lowered, more iterations through the value function are required. From this one can assume that Θ sets the limit or definition for convergence in these processes. Additionally, it should be noted that lowering Θ did not result in any differences in the final optimum policy. However, it clearly has an effect on efficiency and should be adjusted accordingly. Finally, and perhaps most notably, the policy iteration approach always sweeps through the value function more. While this difference within the parameters of this specific problem seem insignificant, this can have a large effect on convergence time and runtime as the state and action spaces grow.

From the optimal policy supplied earlier, one can see that under normal circumstances an agent would act as expected. The agent drives past taken, far away spots, and parks at any open spot. However, it might be more interesting to explore how the agent would act when the form of the reward/penalty is transformed, such as $T_{wait} = -(1/T_{wait})$. Surprisingly, applying this transformation to the immediate penalties, T_{wait}, T_{walk}, T_{drive}, and R_{no} , resulted in no changes to the final optimal policy. That being said, the values for all states within the value function slightly increased.

By default, the agent is penalized and denied ice cream when it attempts to park in a taken spot. However, what if the agent is extremely wealthy and values ice cream more than their own vehicle? It might be interesting to explore how will the agent's behavior changes as the negative punishment associated with parking in a spot that is taken is reduced and it is still allowed to get ice cream. For this observation, the penalty for walking is kept. In this observation, we found that due to an extreme love for ice cream, it will opt to park in some of the closer taken spots. A sample optimal policy for this situation is provided below. While this may be an extreme - and highly unlikely - situation, it illustrates how developers need to place a high degree of consideration into how their reward functions are designed and thoroughly consider what signal is being communicated to the agent through the rewards.

Optimal Policy for An Irresponsible Wealthy Ice Cream Lover

	1	2	3	4	5	6	7	8	9	10	11	12
T	Drive	Drive	Drive	Drive	Drive	Park	Park	Park	Park	Park	Park	Park
NT	Park	Park	Park	Park	Park	Park	Park	Park	Park	Park	Park	Park

T = Taken, NT = Not Taken

Conclusion

Some of the main things that was achieved with this project was learning how to write code for a reinforcement learning problem, working through the dynamics and outcomes involved with the reward signals, and conceptualizing the Markov Decision Process in terms of a reinforcement learning problem.

Throughout the completion of this project numerous lessons, such as how effective the MDP abstraction is and how effective rewards can be in regards to altering behavior, were learned. It has been mentioned, throughout the class and textbook, how the MDP framework can serve as an abstraction for the reinforcement learning problem itself, this fact makes intuitive sense and is relatively easy to understand. However, this information was understood at a much deeper level once the algorithms and code have been constructed in order to implement a solution. One can now define a relationship between the MDP framework and the data structures, algorithms, and functions utilized to produce a solution. Another lesson learned was associated with the relationship between the reward signal and the optimal policy. Once a solution was produced one could view the optimal policy associated with various reward signals, which allowed for

analysis to be conducted. Specifically, one could ask why is a specific policy being constructed, such as observing the agent waiting at specific spots and driving at others, and how have the reward signals produced that outcome? For instance, the optimal policy, when the settings follow the instructions given, is to drive at the earlier spots, but wait once it has reached spots closer to the store. However, this same behavior is not observed if the negative reward associated with T_{walk} is reduced. Therefore, one can walk away from this project knowing that if one wants to truly understand a specific behavior exhibited by an agent then one should start with developing their understanding of the rewards and the signals they are communicating to the agent.

Appendix

Our group determined it would be best to split the work associated with the report and the code such that no individual was particularly responsible for any particular section. Therefore, no individual was necessarily "in charge" of any particular part. The report was created via a shared google doc and code was shared back and forth as well. Throughout the completion of this project both group members were actively engaged in producing solutions in regards to the code and writing different sections of the report.

References:

R. Sutton and A. Barto, "Reinforcement Learning: An Introduction," 2018, 2nd Ed.

http://www.ccs.neu.edu/home/rplatt/cs7180_fall2018/slides/mdps.pdf