Discrete Final Project:
Probabilistic Shortest Paths & Robotics Navigation Applications

Audrey Lee
Amy Phung
Shashank Swaminathan

December 2020

Abstract

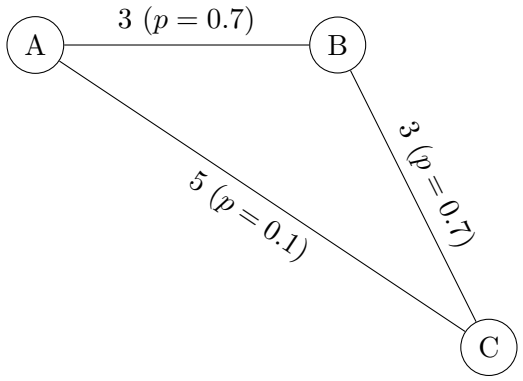
In conventional shortest path problems on deterministic weighted graphs, popular shortest path algorithms such as Dijkstra's or Bellman-Ford work well for finding the optimal path between two nodes. However, these approaches don't transfer well to stochastic graphs, where there is a probability associated with each edge that dictates how likely it is that the edge can be traversed. Our project focuses on the problem of stochastic graph traversal and discusses methods of finding shortest paths within a graph where certain edges may not always be present. In this report, we will discuss the Value Iteration and Policy Iteration methods for solving the shortest path problem in stochastic graphs and how these methods can be applied to landmark-based navigation in mobile robots.

1 Introduction

At the most basic level, a graph is a collection of nodes and edges. In a connected graph, there will exist a path between any two nodes within the graph, and the shortest path between two nodes is simply the path which goes through the least number of nodes. In a weighted graph, each edge also has a cost of traversal, which can simply be modeled as the edges' length. In these graphs, finding the shortest path is a little bit more complicated, since the cost of the edges traversed needs to be considered. One of the most common methods to solving the shortest path problem in weighted graphs includes Dijkstra's algorithm, which is a greedy algorithm that expands its search by iteratively exploring along the shortest path. Another popular shortest path algorithm for deterministic graphs is the Bellman-Ford algorithm, which iteratively expands its search on all of the nodes instead of just along the current shortest path. Because of this, the Bellman-Ford algorithm has a slightly longer runtime than Dijkstra's algorithm, but is able to handle graphs with negative weights while Dijkstra's cannot - in the case there is a "negative cycle" (i.e. a circuit in the graph where the total sum of weights is negative), Dijkstra's will simply choose to traverse the cycle endlessly since this would theoretically result in an infinitely negative cost while the Bellman-Ford algorithm will detect this and terminate. [2]

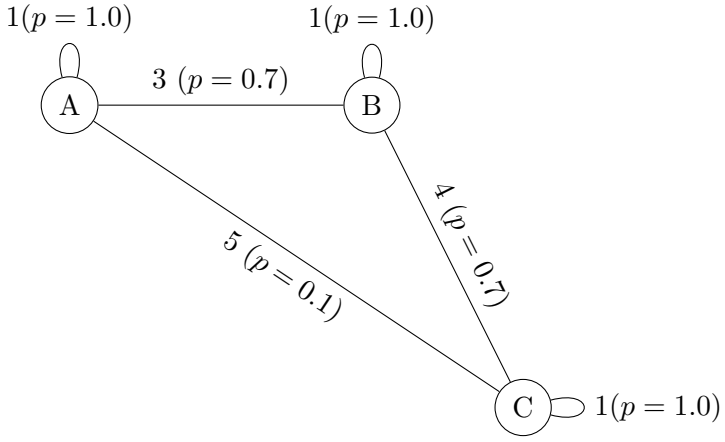
However, both Dijkstra and Bellman-Ford don't work for finding shortest paths in stochastic graphs where edges have a probability that represents the likelihood that it can be traversed at any given time. We will use the following hypothetical example to illustrate why.

Consider a simple graph with three nodes, labeled A, B, C with weights and probabilities as follows:



For the purposes of this example, let's say you'd like to find the shortest path between nodes A and C. In this graph, note that while the length of the edge between A and C is shorter than the sum of edges between A to B and B to C, that path is not always traversable, so we can't simply use the optimal policy found from Dijkstra or the Bellman-ford algorithms.

However, it's worth noting that in this simple case, there's nothing preventing us from just waiting until the edge to the optimal node (as computed by our deterministic shortest path algorithm) appears. To prevent this trivial solution, we need to give the self-edge a non-zero cost.



In this case, since taking the self-edge has a cost, it is not advantageous to wait for the edge between A to C to appear in most cases due to the low chance that the edge appears, even though it could theoretically result in a shorter path. In most cases, it would be a better policy to take the path from A to B and from B to C, which is contrary to the solution that Dijkstra's and the Bellman-Ford algorithms provide. This example illustrates the need for a different shortest path algorithm in order to find an optimal route through stochastic graphs.

In this report, we will discuss how stochastic graphs relate to landmark-based robotics navigation and show how finding a solution to the shortest paths problem in stochastic graphs will also lead to a solution in landmark-based robotics navigation. We will discuss the Value Iteration and Policy Iteration solutions to the stochastic graph traversal problem in more detail in sections 4 and 5.

2 Applications & Robots

Before continuing with the theory behind finding the shortest path in a stochastic graph, let's consider the applications in which it would be useful to provide some motivation for our work. For

starters, finding the best route between two points on a map in the presence of traffic can be well modeled as a stochastic graph - even though taking the highway will oftentimes increase the overall distance traveled, the high chance of getting stuck behind a street light while taking side streets makes the highway the preferred path in many cases. Stochastic graphs also have their uses in planning an optimal bus route - although taking a route with two different buses with a connection may theoretically result in a faster transit time, if each of the buses sometimes arrives ± 5 minutes late, it's often a better idea to take a different route with only one bus even if it takes longer than the two buses combined since it eliminates the chance of getting stuck at a connection. [3]

Let's consider how stochastic graphs can be used for landmark-based navigation on robots. As a hypothetical example, let's consider a situation in which there is a robot that needs to navigate through a house that's labeled with clearly marked visual fiducial tags in each room. In this example, let's also assume that the robot can only navigate to fiducials that it can detect from it's current position. A visual of this example can be viewed in Figure 1

Figure 1: Sample illustration of a robot's environment with clearly marked landmarks and a designated goal node

If we treat each of these fiducials as landmarks, we can create a representation of the robot's world as a stochastic graph. We can model each landmark as a node in our graph, and then the cost of traversal can be represented as the time it takes to cross that edge. [1]

In any given environment, there are a host of reasons why the robot may not be able to see (and thus cannot navigate to) other fiducials - in some cases, this is caused by static obstacles like walls or large pieces of furniture that require the robot to path-plan around (which can be modeled as edges with 0 probability). In other cases, there may be doors that temporarily restrict access to certain landmarks, which can be represented as an edge with some probability. Between some fiducials, there may be no physical barriers, but if there is a lot of foot traffic between one landmark and the other it might take a few seconds until the next fiducial can be detected - in this case, the

[1] note that this cost need not be time - other examples include distance between nodes, or power consumption needed to make the move

edge would have a very high probability. [3] In our example, the robot's possible routes and their associated probabilities are illustrated in Figure 2

Figure 2: Each possible route for the robot is labeled with an edge, whose thickness represents the probability that edge is traversable at any given time

In order to prevent the robot from choosing to stay at one landmark and wait for the unlikely edges to appear, we will also assign the self-nodes a cost and a probability. The cost of staying at the self-node is simply just the timestep in our problem (i.e. the amount of time it takes us to re-evaluate our options), and the probability of the self-edge is simply 1 since it is always an option to stay put. [3]

Now that we've reframed the landmark-based robot navigation problem as a stochastic graph traversal problem, we can see that if we find a method to find shortest paths in a stochastic graph, we've also found a way to find an optimal route for landmark-based robotics navigation. Let's begin to dive into the theory to find our optimal path.

3 Markov Decision Process (MDP)

There are a number of techniques that have been developed to find optimal solutions in a Markov Decision Process (MDP), so let's attempt to reframe our problem using a MDP formulation.

Let's start off with a bit of background - in a MDP, an agent chooses action a_t at time t with a state s_t and receives a reward r_t (also called a cost in the case that the reward is negative). The state changes probabilistically based on the current state and action chosen by the agent. In order to frame a problem as a MDP, the *Markov assumption* needs to hold - i.e. the next state must only depend on the current state, and not on the prior history of states. The *state transition function* $T(s'|s,a)$ represents the probability of ending up at s' if the agent takes action a at state s. The *reward function* $R(s,a)$ represents the expected reward (or cost, if the reward is negative) of taking action a at state s. A *policy*, denoted as π, determines what action to select given a specified state. The expected utility, also known as a *value function*, is denoted as U^{π} and represents the expected total rewards or costs accrued from following policy π. The *optimal policy* π^* is the policy that maximizes the expected utility

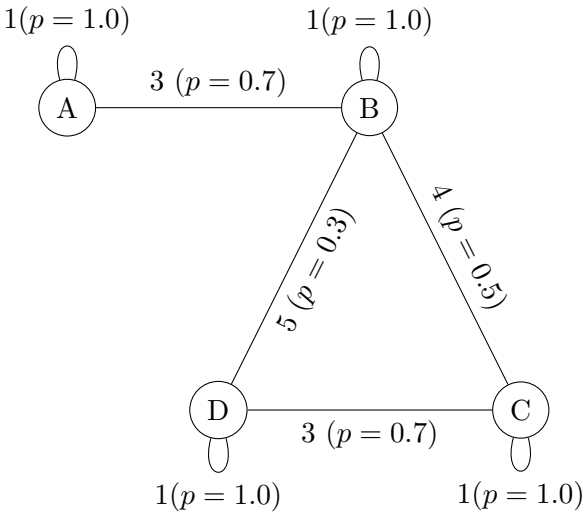
$$\pi^*(s) = \arg\max_{\pi} U^{\pi}(s)$$

for all states s [4].

We can reframe our robotics example to leverage MDP solution techniques as follows: our robot is the agent, the cost is time, and the state is the landmark that the robot is currently at. Finding an equivalent match for policy is a bit trickier - since we've established that we can't just denote one singular "best move" for each possible state, our action can't simply be a fixed node. Instead, our action a must encode a *strategy* - in other words, our action will be a list of nodes the robot should attempt to navigate to, assuming it's at a particular state. For example, for a robot currently at landmark C the strategy A, B, C means that a robot should attempt to go to node A if possible, B if A is not possible, and should stay at node C if both A and B are not possible. With this framing, we can see that our optimal policy π will contain a strategy for every possible state in the map. We will go over methods for computing the optimal policy in sections 4 and 5.

3.1 Adjacency and Probability Transition Matrix

We can represent our graph in a matrix format by keeping track of which nodes are adjacent to each other. Given an example graph, we can represent the nodes and their connections with ones denoting a connection between nodes and zeros if there is not a connection.



Given the example graph above, we have node A being connected to itself and node B. Node A's adjacency list can be represented as [1,1,0,0] where the columns represent the different nodes: [A,B,C,D]. Since we can represent the "current" node as the row and the adjacent node for the columns, we can create a matrix that represents how the nodes are connected to each other. For this example, the matrix is

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Since the example graph does not have directional edges, the matrix is symmetrical since a node going to another node is the same in reverse.

Now that we can represent our graph as a matrix, we can now generate the probability transition matrix needed to compute the optimal policy. Using the same logic as above, we can represent the probabilities like so

$$\begin{bmatrix} 1 & 0.7 & 0 & 0 \\ 0.7 & 1 & 0.5 & 0.3 \\ 0 & 0.5 & 1 & 0.7 \\ 0 & 0.3 & 0.7 & 1 \end{bmatrix}$$

This matrix tells us the current node (row) and the node the robot is going to (column) and the probability that it will be able to traverse that edge. By knowing the probabilities and formatting them in a matrix format, we can now find the optimal policy.

4 Value Iteration

Value iteration is one algorithmic way of finding the optimal policy in an MDP formulation. It is commonly used for its intuitive and simple implementation.

4.1 Process

The discussion in this subsection is derived from [4].

There are slight differences whether or not we consider an infinite horizon for utility. In the infinite horizon case, we can use the Bellman equation to show that the utility gained from the optimal policy satisfies the below condition. Hence, the goal is to find the policy that satisfies the below relation. However, algorithmically, it is difficult to directly solve for such a policy. Instead, we will use a recursive method to improve the gained utility from a base case until convergence (when the utility before improvement is sufficiently close to the utility post improvement, implying that utility is sufficiently close to the true optimal value).

$$U^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U^*(s') \right)$$

To do so, we start with a formulation for recursively determining the optimal utility U_n. Using a base case of $U_0(s) = 0, \quad \forall s$, we can recursively update U_n from U_{n-1} as shown below, until $||U_n - U_{n-1}||_\infty < \delta$ for some threshold δ. Here, $R(s,a)$ represents the reward function, $T(s'|s,a)$ represents the transition probability matrix, s' represents all the possible states that can be reached from current state s, and γ represents the discount factor. Note that n represents the finite time horizon, and so the larger n is before convergence, the larger time horizon is required to achieve the true optimal utility. Similarly, γ is a value between 0 and 1 that controls how much of the future utility is considered at the current state; the smaller it is, the less the algorithm will weight the effect of future states. This will speed up convergence, as there is a smaller effective time horizon.

$$U_n(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U_{n-1}(s') \right)$$

Once we have found the optimal utility, or a close approximation of it, we can extract the optimal policy from it. Recalling the Bellman equation, at optimal utility, we can perform the recursive update once more and remain at the optimal utility. Hence, we will perform the update one more

time, remaining at optimal utility, but we will then also save the mapping of actions to states that maximized the rewards during the update - the arg max operation. This is expressed as below.

$$\pi^*(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U^*(s') \right)$$

It can be shown that as long as U_0 is initialized to some bounded value (i.e. $|U_0| < \infty, \forall s$) the above process will still converge on an optimal utility and policy [4]. Thus, to speed up convergence, rather than initializing the base case as $U_n = 0$, we can initialize it to some estimate of the optimal policy, so that less iterations are required till convergence.

Finally, rather than representing the update of utility as an iterative process through all the states, we can also represent it concisely using matrix notation. Given the set of all possible states $\mathbb{S}$, let $\mathbf{R}^\pi$ be the $|\mathbb{S}| \times 1$ vector of rewards received at all $|\mathbb{S}|$ states following policy π. Similarly, let $\mathbf{T}^\pi$ be the $|\mathbb{S}| \times |\mathbb{S}|$ probability transition matrix of moving from state s to state s', for all possible states s and s', when following policy π. Finally, we'll define the $|\mathbb{S}| \times 1$ vector $\mathbf{U}_n^\pi$ to be the vector of utility received at all $|\mathbb{S}|$ states for time horizon n and policy π. We can then write the recursive relation for the update of utility $\mathbf{U}_{k+1} \leftarrow \mathbf{U}_k$ to be as shown below:

$$\mathbf{U}_{k+1}^\pi = \max_\pi \mathbf{R}^\pi + \gamma \mathbf{T}^\pi \mathbf{U}_k^\pi$$

We can then take this expression and with some manipulation turn it into a forward-looping expression for the update of utility, as shown below:

$$\mathbf{U}^\pi = (1 - \gamma \mathbf{T}^\pi)^{-1} \mathbf{R}^\pi$$

4.2 Proof

The formulation of the proof was derived from [1] and [5].

Our goal is to prove that the above stated Value Iteration method works, i.e. that it will eventually converge on the truly optimum utility and policy. We'll focus on showing the proof for the infinite time horizon case. To do so, let's first define a function of convenience $V(U)$ to represent the process of updating $U_{k+1} \leftarrow U_k$ as follows:

$$V(U) := \max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U(s') \right) \forall s$$

Here, $U_{k+1} = V(U_k)$ (for ease of notation, the (s) has been omitted).

Now, let us briefly consider what it means for the Value Iteration method to converge regardless of the base case's initial value. It implies that after sufficient updates, regardless of what U_0 is, U_n will approach U^*. This means that if we were to arbitrarily select two distinct values for U_0, which we'll name $\hat{U}$ and $\tilde{U}$, after a sufficient amount of updates the two utility values will converge. Mathematically, we have:

$$\lim_{n\to\infty} \hat{U}_n - \lim_{m\to\infty} \tilde{U}_m = 0$$

Our next step will build off that intuition, to prove the following: given two arbitrarily selected distinct values for U_0, denoted $\hat{U}$ and $\tilde{U}$,

$$||V(\hat{U}) - V(\tilde{U})||_\infty \leq \gamma ||\hat{U} - \tilde{U}||_\infty$$

To continue, let us first state w.l.o.g. that $V(\hat{U}) \geq V(\tilde{U})$. This is acceptable, because if in truth $V(\hat{U}) < V(\tilde{U})$, we can simply rename $\hat{U}$ as $\tilde{U}$, and vice versa. Next, let us name the optimal action to take at any state s to be a_s^*. Next, if we were to expand the contents of the max norm in the expression above using the definition of $V(U)$, we have:

$$\left(R(s, a_s^*) + \gamma \sum_{s'} T(s'|s, a_s^*) \hat{U}(s') - R(s, a_s^*) - \gamma \sum_{s'} T(s'|s, a_s^*) \tilde{U} \right) (s') \forall s$$

Because the rewards model is independent of current utility, we can cancel that out, leaving us:

$$\left(\gamma \sum_{s'} T(s'|s, a_s^*) \hat{U}(s') - \gamma \sum_{s'} T(s'|s, a_s^*) \tilde{U} \right) (s') \forall s$$

Likewise, because the transition model is only dependent on the state and action, for a given state and action, it will not change. We can thus combine the summation symbols as follows:

$$\gamma \sum_{s'} T(s'|s, a_s^*) \left(\hat{U}(s') - \tilde{U}(s') \right) \forall s$$

From the definition of the max norm, $||\hat{U} - \tilde{U}||_\infty \geq \hat{U}(s') - \tilde{U}(s') \forall s'$. Substituting this in for $\hat{U}(s') - \tilde{U}(s')$, we then get the following inequality:

$$\gamma \sum_{s'} T(s'|s, a_s^*) \left(\hat{U}(s') - \tilde{U}(s') \right) \leq \gamma \sum_{s'} T(s'|s, a_s^*) ||\hat{U} - \tilde{U}||_\infty$$

We now drop the $\forall s$ without loss of generality, as the above statement must hold true for any state s. Finally, recall that the transition probability represents the probability of the event of transitioning to state s' occurring. As we are summing across all states s', we are summing across the entire sample space of the transition probability, which reduces simply to 1 (because $||\hat{U} - \tilde{U}||_\infty$ is constant), from the fundamental rules of probability. Hence, we can reduce the expression to as follows:

$$\gamma \sum_{s'} T(s'|s, a_s^*) ||\hat{U} - \tilde{U}||_\infty = \gamma ||\hat{U} - \tilde{U}||_\infty$$

Hence, we have shown that for two arbitrarily selected values of U, at any state s, and thus for states, the following is true:

$$||V(\hat{U}) - V(\tilde{U})||_\infty \leq \gamma ||\hat{U} - \tilde{U}||_\infty$$

If we were to use the U_{k+1} notation instead of $V(U_k)$, we can express this as follows:

$$||\hat{U}_{k+1} - \tilde{U}_{k+1}||_\infty \leq \gamma ||\hat{U}_k - \tilde{U}_k||_\infty \tag{1}$$

Likewise, we can do a quick inductive proof to show that the proposition $P(n)$, seen in Equation 2, holds true for all positive integer values of n.

$$P(n) := ||\hat{U}_n - \tilde{U}_n||_\infty \leq \gamma^n ||\hat{U}_0 - \tilde{U}_0||_\infty \tag{2}$$

We start with a base case of $n = 1$. Here, we note from Equation 1 that we can write the expression seen below (where $k = 0$). This matches the expression for $P(1)$, proving the existence of the base case.

$$||\hat{U}_1 - \tilde{U}_1||_\infty \leq \gamma ||\hat{U}_0 - \tilde{U}_0||_\infty$$

Now, for the inductive step: let us assume the truth of the inductive hypothesis $P(k)$, that $||\hat{U}_k - \tilde{U}_k||_\infty \leq \gamma^k ||\hat{U}_0 - \tilde{U}_0||_\infty$ is true. Now, let us consider if this holds true for the case of $P(k+1)$. Let us start by rewriting Equation 1 below.

$$||\hat{U}_{k+1} - \tilde{U}_{k+1}||_\infty \leq \gamma ||\hat{U}_k - \tilde{U}_k||_\infty$$

We now assert the inductive hypothesis as seen below:

$$||\hat{U}_{k+1} - \tilde{U}_{k+1}||_\infty \leq \gamma ||\hat{U}_k - \tilde{U}_k||_\infty \leq \gamma \left(\gamma^k ||\hat{U}_0 - \tilde{U}_0||_\infty \right)$$

We then simplify to get the following relation:

$$||\hat{U}_{k+1} - \tilde{U}_{k+1}||_\infty \leq \gamma^{k+1} ||\hat{U}_0 - \tilde{U}_0||_\infty$$

Through this, we've now shown that if $P(k)$ is true, $P(k+1)$ must also be true. As the proposition holds true for the base case $n = 1$, through the principles of mathematical induction, we can conclude that the proposition holds for all positive integer values of n.

With the proposition proven, let us consider the case of $\lim_{n\to\infty} P(n)$. Here, we have as follows:

$$\lim_{n\to\infty} \left(||\hat{U}_n - \tilde{U}_n||_\infty \leq \gamma^n ||\hat{U}_0 - \tilde{U}_0||_\infty \right)$$

As long as we have a discount factor of $0 \leq \gamma < 1$, then the term γ^n will go to 0 as n approaches infinity. Thus, given such a discount factor, we can make the statement that

$$\lim_{n\to\infty} ||\hat{U}_n - \tilde{U}_n||_\infty = 0$$

Note that due to our assumption made when proving Equation 1, $\hat{U}_n \geq \tilde{U}_n$, and so $\hat{U}_n - \tilde{U}_n \not< 0$.

And with that, we've shown regardless of base case initialization, the value iteration method will always converge!

5 Policy Iteration

The discussion in this subsection is derived from [4].

Compared to Value Iteration (VI), Policy Iteration (PI) manipulates the policy itself rather than finding it through the optimal value function. In comparison to VI, PI has a larger time complexity, but will converge quicker which is what makes PI competitive and relevant.

Similar to VI, we will use a recursive method to improve the policy from a random base case until convergence. This implies that it will converge when the policy before it has been improved is deemed close to the policy post its improvement (which means that it's close to the true optimal policy).

5.1 Process

In PI, it will start with some random base case policy and find a value function corresponding to it. It will then try to improve the policy based off of the previous value function, and will recursively do so until it finds the optimal policy.

In the context of robotics, PI would involve calculating the optimal policy from the probabilistic transition between nodes/locations. Given an array of possible states and an array of possible actions (moving up, down, left, right, or staying in one place), it will check if the policy has stabilized from a current policy and a previous one. If it has stabilized, it has found optimal policy to go from a location to the goal location.

5.1.1 Policy Evaluation

Since PI tries to find the optimal policy, it will need to check if the policy has stabilized. This means that it needs to check if the change in the policy from one policy to another has changed by some δ (some pre-determined threshold value). If δ is greater than the threshold value, a new value function needs to be calculated to then calculate a new and improved policy. To check if the policy has stabilized, it will use the equation

$$\pi_n(s) = \arg\max_{\alpha}(R(s,\alpha) + \gamma \sum_{s' \in S} P(s'|s,\alpha) U^{\pi_{n-1}}(s')) \tag{3}$$

where n is the number of iterations (so the larger n is, the longer it took to converge), $\pi(s)$ is the current policy, s is the current state, s_1 will be the states leading up to the current state S, α is the possible actions at the current state, P is the probabilistic transition matrix, R is the reward matrix, γ is the discount factor, and U is the current value function evaluated at all states up to the current state.

Equation 3 calculates the policy given by the previously calculated value function. By comparing $\pi_i(s) - \pi_{i-1}(s) < \delta$, where i is the current step, then if the difference between the two steps are smaller than some pre-determined δ value, then the policy has not significantly changed, and therefore, is considered optimal. This means that the new policy is approximately the same as the previous policy, and then we know that the policy has stabilized, and we have found our optimal policy: $\pi^*(s) = \pi(s)$. [4]

5.1.2 Policy Improvement

If it has not stabilized, then we have to calculate a new policy by first finding a new value function from the current policy. The utility associated with the current policy can be calculated with this equation

$$U_k^{\pi_n}(s) = R(s, \pi_n(s)) + \gamma \sum_{s' \in S} P(s'|s, \pi_n(s)) U_{k-1}^{\pi_n}(s') \tag{4}$$

where k represents the number of iterations to find the utility associated with the current policy. We are now calculating the new value function for the current state to get a new possible "optimal" policy. This means that after getting the new value function, we will use it in equation 3 to calculate a new possible "optimal" policy.

It will then check if it has stabilized by comparing the new policy to the previous one. If not, we repeat the process. But, if it is considered optimal, then we have the policy for the robot to make its decisions.

5.2 Proof

The formulation of the proof was derived from [4]. This proof will show that Policy Iteration will reach an optimal policy. In other words, that Policy Iteration will converge and reach a result such that when there are a finite amount of states and actions, no two consecutive policies will have the same policy unless it's the optimal policy.

Let's look at a current policy π_t and the policy after the current π_{t+1}. Let U_n and $U_n + 1$ represent values for two consecutive iterations such that $U_n \leq U_{n+1} \leq U^*$.

If we focus on the Policy Improvement step, we can check the values of the consecutive iterations. Since we're trying to improve the policy, the next policy must be greater than or equal to the current policy which can be represented as

$$R(s, \pi_{n+1}(s)) + \gamma P(s'|s, \pi_{n+1}(s)) U_{k-1}^{\pi_{n+1}}(s') \geq R(s, \pi_n(s)) + \gamma P(s'|s, \pi_n(s)) U_{k-1}^{\pi_n}(s') \tag{5}$$

[3]

This means that there will be no duplicate policies, and only when this condition is met will we be able to actually compute and save the new optimal policy since otherwise, the policy would not be improving.

Now, assume for contradiction that the algorithm is stuck at a local optimum and not the global optimum. However, this cannot happen since there exists at least one state-action that is of higher value than the local optimum. Since the policy is either improving (increasing) or staying the same and cannot decrease, it cannot get stuck at a local optimum. Since the global optimum is of a higher value than the local one, the inequality from equation 5 would be true since there exists a value above the current one. This means that it will continue to iterate and improve until it reaches the global optimum and it cannot get stuck at a local optimum.

This means that Policy Iteration will converge and reach an optimal policy.

5.3 Time Complexity

Compared to Value Iteration, Policy Iteration has to solve the linear system of equations, which can take $O(N^3)$ time where N is the number of nodes in the graph. [3] Value Iteration has a linear time complexity, but Policy Iteration's time complexity is of cubic degree. Even though Value Iteration's time complexity is smaller, the goal of performing Policy Iteration is in hopes of converging to an optimal policy faster. This is because we are solving for the optimal policy directly instead of trying to compute the utility function exactly.

6 Code

We also decided to try and implement what we learned by programming it. Although it may not work, we used this opportunity to learn and deepen our understanding on the topics.

Here is the link to our Github repository.

Annotated Bibliography

- [1] We used this source for some of the background on Value Iteration and for the key theorems required to prove the method.
- [2] We used this source for some of the background information on conventional path planning algorithms in deterministic graphs as a point of comparison for the stochastic graph case.
- [3] The majority of our introduction and understanding of robotics with stochastic graphs. We used this source for background on Value and Policy Iteration, their equations, and a general understanding of proving their convergence.
- [4] We used this source for the background on Markov Decision Processes, including the formulations, equations, figures, and high-level algorithms.
- [5] We used this source to help derive the proof of convergence for the Value Iteration.

References

[1] Pieter Abbeel. *Markov Decision Processes and Exact Solution Methods*. URL: https://people.eecs.berkeley.edu/~pabbeel/cs287-fa12/slides/mdps-exact-methods.pdf.

[2] Omar Khaled Abdelaziz Abdelnabi. *Shortest Path Algorithms*. URL: https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/ (visited on 12/16/2020).

[3] Amy J. Briggs et al. "Expected Shortest Paths for Landmark-Based Robot Navigation". In: *The International Journal of Robotics Research* 23.7-8 (2004), pp. 717–728. DOI: 10.1177/0278364904045467. eprint: https://doi.org/10.1177/0278364904045467. URL: https://doi.org/10.1177/0278364904045467.

[4] Mykel J. Kochenderfer et al. *Decision Making Under Uncertainty: Theory and Application*. 1st. The MIT Press, 2015. ISBN: 0262029251.

[5] Pascal Poupart. *CS 886 Sequential Decision Making and Reinforcement Learning: Module 6 Value Iteration*. URL: https://cs.uwaterloo.ca/~ppoupart/teaching/cs886-spring13/slides/cs886-module6-value-iteration.pdf.