Learning to Find the Optimal Path

Kevin Robb (4033), Logan Elliott (5033)
Long Reinforcement Learning Project
(Dated: November 9, 2020)

I. INTRODUCTION

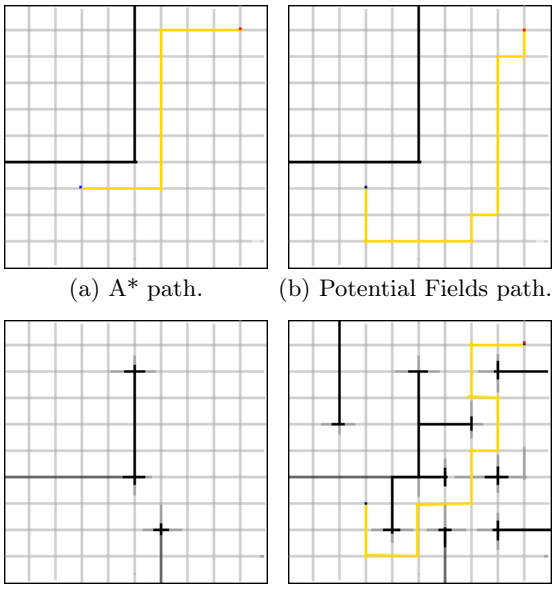
Path planning is a vital aspect of mobile robotics. The ability to explore an environment autonomously is an important area that has been and continues to be researched very thoroughly. A robot that can explore and learn how to find a path through a map will be more prepared to journey to unknown environments, such as for search-and-rescue or space exploration applications. We won't always have optimal localization or a map of our environment so it's important to experiment with other approaches to robot localization. The domain of this projects centers around comparing reinforcement learning ideas against traditional path planning methodologies. We seek in this project to develop agents which learn to find a path that approaches optimality with minimal training time.

II. IMPLEMENTATION

We created a robot simulation with Robot Operating System's (ROS) Stage simulator and the turtlebot3. This simulator allows us to perform repeatable robot experiments in 2D using custom-designed environments. Additionally we wrote a custom control architecture leveraging the robot's odometry and LiDAR data. This allowed us to standardize all of our agents' interactions within the environment; they each receive the same discretized sensor values and can control the robot via the same commands. All the agents can send one of four control messages: North, East, South, West. Each command will result in movement to the adjacent tile in the specified direction, unless the direction is obstructed, in which case the robot will do nothing. Turning is handled by the control node, so agents need only worry about the direction they want to move. These simplifications allow for great reduction in the size of our state space; reinforcement learning is much more approachable in this 10x10 tiled environment proxy than the underlying continuous one. This can in principle be expanded to the desired degree of precision (by discretizing the environment into smaller tiles), requiring more computation time and power.

We implemented three heuristic agents to compare against our learning agents.

1. Random Agent: chooses a random action each timestep
2. A* Agent: first finds the optimal path and then pursues it
3. Potential Fields Agent: generates a cost map which reactively guides it to the goal



(a) A* path. (b) Potential Fields path. (c) Training map. (d) Validation map.

FIG. 1. The maps used in our simulation.

Both A* and potential fields are using a map of the environment for their path planning. A* is representing the optimal path through the environment, and will set the optimum goal for our learning agents. Potential fields serve as a non-optimal metric to compare against, as it has drawbacks such as the robot getting trapped locally and failing to reach the goal.

We first used a simple map to demonstrate functionality of our heuristic agents. As demonstrated in FIG. 1(a)-(b), agents travel from $(-2,-2)$ to $(4,4)$, with the center of the map as the origin. The optimal path found by A* and the non-optimal path found by Potential Fields are both shown; the random agent is different every time and generally involves a lot of backtracking and looping, so it is not included.

We implemented our reinforcement learning agents using the map shown in FIG. 1(c), and after tuning all hyper-parameters and behavior, we ran all agents on the more complicated map shown in FIG. 1(d). Us-

ing a different map for training and for validation ensures we did not simply overfit our agents to the specific map we started with. FIG. 1(d) also shows the optimal path found by A*, which we use to compare all learning agents; this path has length 18, and we will consider an agent to have found the optimal path if it has length of 19 or less. The simulated map described by FIG. 1(d) is included in FIG. 2.

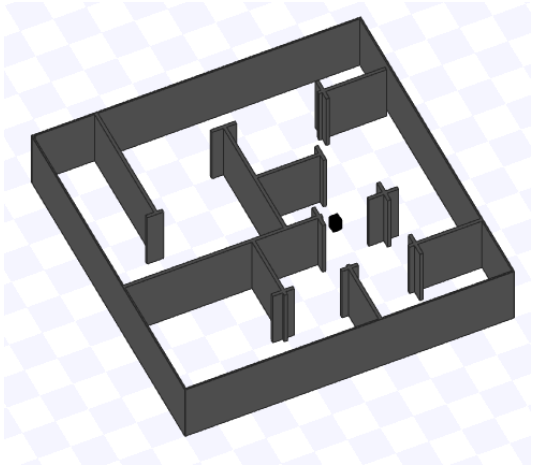


FIG. 2. The actual simulated map used for all agents as described in Results.

III. HYPOTHESES AND COMPARISONS

We have six main hypotheses and three comparisons between learning methods for this experiment, detailed as follows:

- H_0: Q-Learning will outperform the random agent after training.
- H_1: SARSA will outperform the random agent after training.
- H_2: Q-Learning will find a path that is within 10 percent of the length of the optimal path found by A*.
- H_3: SARSA will find a path that is within 10 percent of the length of the optimal path found by A*.
- H_4: Q-Learning will outperform Potential Fields by taking more risks and avoiding local traps.
- H_5: SARSA will outperform Potential Fields by taking more risks and avoiding local traps.
- C_0: Q-Learning vs. SARSA.
- C_1: Double-Q-Learning vs. Q-Learning.
- C_2: Expected SARSA vs. SARSA.

These hypotheses allow us to accomplish our stated goal of comparing our RL ideas to more established metrics. Additionally, the comparisons allow us to contrast different implementations of similar RL ideas.

Our experiments around this hypothesis are simple. We first use an easy training map with multiple correct paths, shown in FIG. 1(c), as we work to implement our algorithms and demonstrate functionality. Then we use the more complex environment shown in FIG. 1(d) and run each algorithm several times to generate some data that we discuss in Section V. Each algorithm follows the details laid out in Section II, and is considered to have finished training when it consistently achieves a path length within 2 steps of optimality (determined by A*). Our main metric of comparison is the number of steps an agent takes to reach the goal, and a secondary metric is the number of episodes it takes to achieve near-optimality. The benefit of path length as a metric is that it is shared by the RL and heuristic algorithms, so provides a standard for comparison. The secondary metric only makes sense for RL agents, so it is used only for comparison between learning algorithms.

IV. LEARNING METHODS

Our main two reinforcement learning methods are SARSA and Q-Learning. We use a standard implementation of each, as well as some variations for comparison as specified below. We chose these two algorithms because of their prevalence in the literature and their ability to be used fairly well for our task; we wanted to be able to represent the rewards of each given path segment, and both Q-Learning and SARSA allow us to model path segments as state-action pairs, and update the rewards for each pair separately.

The initial implementation of SARSA used the following update rule:

$$Q(S,A) \leftarrow Q(S,A)+\alpha\cdot\left[R+\gamma\cdot Q(S',A')-Q(S,A)\right] \quad (1)$$

where

- S is the current state,
- S' is the next state,
- A is the chosen action from S to S',
- R is the reward obtained for taking action A, and
- A' is a possible action from S' to some other state.

This follows the standard SARSA implementation outlined in class. When a state is discovered by the agent, its value is initially set to 0. In the final implementation, we used $\alpha = 0.05$ and $\gamma = 0.99$. We balance the exploration vs. exploitation dynamic by using an ϵ-greedy approach, with $\epsilon = 0.2$. At the end of training, we reduce this to $\epsilon = 0.05$ so the agent will mostly follow the path it has determined is the highest value; we do not go all the way down to $\epsilon = 0$ to prevent the agent from potentially becoming trapped locally.

The Q-Learning agent has a similar update rule:

$$Q(S,A) \leftarrow Q(S,A) + \alpha \cdot \left[R + \gamma \cdot M - Q(S,A)\right] \quad (2)$$

where

$$M := \max_{A' \in S'} \{Q(S', A')\} \ . \quad (3)$$

Here we use $\alpha = 0.05$, $\gamma = 0.9$, and $\epsilon = 0.2$, similarly decreasing ϵ at the end to allow for maximum performance. The max in the update equation is computed only among actions that have been taken previously; the agent takes a random action if it's in a state from which it has not acted before.

The reward function for both agents is as follows:

$$R = \begin{cases} 100, \text{ reached goal} \\ -1, \text{ made a valid move} \\ -5, \text{ tried to move into a wall} \end{cases} \quad (4)$$

We originally had an additional punishment for moving into a previously visited tile on the same path (since this is a clear redundancy), but decided that it gives our agents more information than they really need or perhaps should have access to. Removing this did increase the average number of episodes needed to learn the optimal path in some cases, but did not impact the ability of the agent to eventually converge on the optimal path.

Our comparison learning algorithms are based on public implementations of Double-Q and Expected SARSA. More information on expected SARSA can be found here and here. More information on Double-Q-Learning can be found here.

Expected SARSA follows a very similar structure to base SARSA. The main difference is the update rule, which is:

$$Q(S,A) \leftarrow Q(S,A) + \alpha \cdot \left[R + \gamma \cdot \sum_a \pi(a|s_{t+1}) \cdot Q(s_{t+1}, a) - Q(S,A)\right] \quad (5)$$

where for $G = \#$ Greedy Actions,

$$\pi(a, s_{t+1}) = \frac{\epsilon}{\#\ Actions} + \begin{cases} 0, \ a = Non-Greedy \\ \dfrac{1-\epsilon}{G}, \ a = Greedy \end{cases} \quad (6)$$

Double-Q uses the following equations in place of the standard update rule.

$$\begin{aligned} a^* &= argmax_a Q^A(s', a) \\ b^* &= argmax_b Q^B(s', a) \\ Q^A(S,A) &\leftarrow Q^A(S,A) + \alpha \cdot \left[R \right. \\ &\quad \left. + \gamma \cdot Q^B(S', A^*) - Q^A(S,A)\right] \\ Q^B(S,A) &\leftarrow Q^B(S,A) + \alpha \cdot \left[R \right. \\ &\quad \left. + \gamma \cdot Q^A(S', A^*) - Q^B(S,A)\right] \end{aligned} \quad (7)$$

where we update A vs. B randomly.

We found that training using the full simulator was fairly slow, taking at least twenty seconds for each of hundreds of episodes. Initial episodes could take hundreds of steps longer than necessary to reach the goal, further extending the runtime. We had the time necessary to train our agents, but this gave us a good area for optimization. We developed the Accelerated SARSA agent to satisfy the novelty component of the project. This algorithm uses a map of the environment that the robot builds as it explores; the first few iterations are essentially random, which gives a good expansive basis for the map. We use a Python dictionary to store data for each state, with coordinates of each tile serving as keys; this allows us to add newly visited states without needing a gridded structure that assumes more knowledge of the map than the agent should have. Data stored includes previously taken actions with rewards and destination states. After enough iterations have gone through the full simulation to build up an internal map, the agent undergoes many more training runs entirely in simulation which do not depend on the actual motion of the robot. These simulated runs help us converge to optimality in much less real-time.

V. RESULTS

We see that Q-Learning, SARSA, and all the variations are able to more-or-less converge onto the optimal path length of 18 steps within 50 episodes. Each of the plots in FIG. 3-6 shows several training sessions from scratch of the given agent to provide an idea of the consistency of the learning algorithms. The general trend we see is that Q-Learning and its variations (FIG. 3, 6(a)-(b)) quickly and consistently converge directly to the optimal value in about 25 episodes. SARSA and its variations (FIG. 4, 6(c)-(d)) are a bit less consistent, sometimes fluctuating in step numbers and even increasing at times, but mostly converging within 45 episodes.

The random agent (FIG. 5) never converges, wildly fluctuating and staying near the max cutoff of 1000

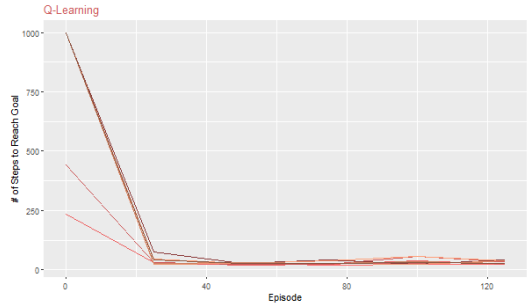


FIG. 3. The learning curve for the Q-Learning agent.

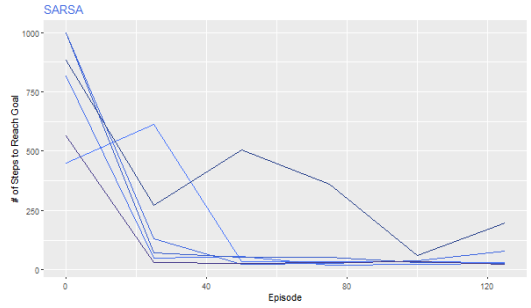


FIG. 4. The learning curve for the SARSA agent.

steps. Potential Fields aggressively avoids walls, so it gets trapped in the initial room on this more complicated map and is unable to complete the course. We can safely say that our first two hypotheses, H_0 and H_1, are shown to be true, since our learning agents clearly beat random, converging while random does not and achieving consistency. H_2 and H_3 are also proven since all of our algorithms found the goal and escaped the local minima trap of potential fields. All of our learning methods and their variations clearly outperform both of these methods by finding the goal and learning to improve their number of steps until converging at or near the optimal path.

Looking at H_4 and H_5, we need to compare the number of steps that each agent consistently converges to. We know the optimal path length for the validation map in FIG. 1(d) on which all of these graphs are based is 18 steps. All six learning agents are able to converge to the range of 18-20 steps to reach the goal, and waver slightly even after convergence due to our non-zero ϵ value. When we run these agents with $\epsilon = 0$ after training, they precisely follow the optimal path and achieve 18 steps. Thus these hypotheses are verified as well.

In performing the comparisons specified in C_0, C_1, and C_2, we have two evaluative metrics: efficiency and consistency. Efficiency describes how rapidly an agent converges to the optimal path, measured in number of episodes. Consistency describes what portion of several training sessions have similar behavior versus those that

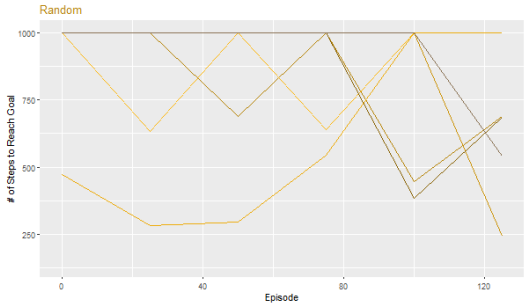


FIG. 5. The "learning curve" for the random agent.

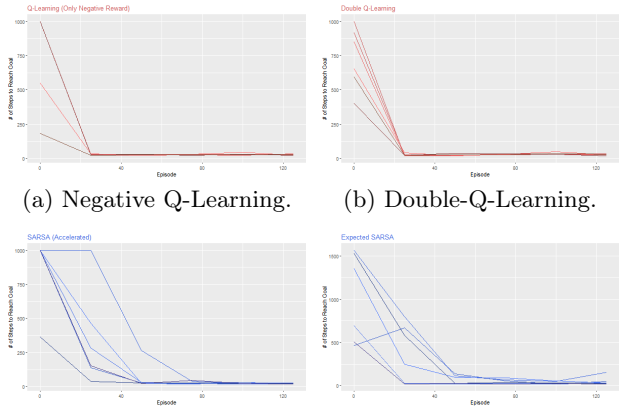
fail or perform abnormally.

The variation in steps on the very first episode is attributed entirely to randomness, as the agents all essentially wander to build a map and find the goal in the beginning. This is consistent across almost all the agents. We see a general trend that Q-Learning methods tend to take a more aggressive approach and converge more quickly while SARSA is more conservative. This aligns with what we have learned in class.

For C_2, we see from the differences between FIG. 4 and FIG. 6(c)-(d) that the variations in SARSA improve the consistency of convergence to the optimal path, but they do this at the cost of efficiency, taking more episodes to settle on the correct path. Expected SARSA's tendency to weight the actions based on probability turned out not to be very effective, since the robot could select any action at any point with minimal consequences; the extra computation to eliminate the random selection of A was only minimally helpful. Accelerated SARSA proved to be an interesting experiment, but not altogether successful. Tuning this agent proved frustrating; too low of an ϵ would cause it get trapped locally, but too high of an ϵ would negatively impact training. Given that it didn't have assess to the actual map, only its self-created one, it would train poorly if it repeatedly chose actions which weren't in its map. It would either weight a bad action too highly or give too much negative reward from too much frequency on a good action. Accelerated SARSA has some merits, but it needs more work before a declaration of its relative value can be made.

For C_1, Q-Learning (Fig. 3) is not improved significantly by the Double-Q-Learning variation (FIG. 6(b)), nor by the removal of positive rewards (FIG. 6(a)). This is not surprising given that Double-Q is meant more for solving some edge case of Q-learning for which it estimates poorly, and our reward function is sufficient even without an explicit reward for reaching the goal.

For this environment, in which there is no real danger besides a negative penalty for choosing an obstructed action (wasting a move), Q-Learning is able to thrive by quickly aligning to the optimal path, while SARSA's



(a) Negative Q-Learning. (b) Double-Q-Learning.

(c) Accelerated SARSA. (d) Expected SARSA.

FIG. 6. Variations of our main RL methods.

safer approach is unnecessary. If we had allowed our agents to make obstructed moves and die upon collision with a wall, we expect that Q-Learning would not have so clearly outperformed SARSA, and Double-Q might have had more of an effect. We can conclude for C_0 that our implementation of Q-Learning is both more efficient and more consistent than any SARSA variation attempted. This is not surprising given that a more aggressive approach can achieve the goal much faster in a relatively safe environment such as ours.

Overall, we would argue that the RL algorithms are useful in path planning. They have proven to be comparable to the traditional methods. The downside to the RL algorithms is the large amount of time to train, but a benefit is that a prior map of the environment is not required. If a large amount of time is available but an existing map of the environment is not, RL might provide a viable solution for navigating the space.

VI. CONTRIBUTIONS

Logan did the initial environment setup and map design as he had prior experience. Logan and Kevin collaboratively wrote the detailed controls architecture for the robot. Kevin completed initial testing in the environment and designed the first agent, random. For the basic heuristics Kevin wrote Random and Logan wrote A*. Logan wrote potential fields to add more hypotheses for the additional graduate requirements. For the RL agents Logan wrote basic SARSA and expected SARSA for comparison. Additionally, Logan wrote the accelerated SARSA for the graduate novelty. Kevin wrote Q learning, and collaborated with Logan for double Q for comparison. Kevin wrote a majority of the data collection code and all of the plotting code in R.

VII. RELATED WORK

In 2017, Meyes et al. simplified a 6-axis industrial robot into 2D to play a wire loop game [2] . The wire loop game involves guiding an apparatus along a wire without touching it. This robot uses computer vision to created a gridded approximation that is fed through Q-Learning. This approach to RL is very similar to ours. We also discretize our environment. Where we use odometer and LiDAR data, they use computer vision. Additionally, they choose to simplify their 3D environment into 2D as we also did to decrease computational power. The most interesting difference from our RL algorithms is how they attempt to accelerate their training; it uses a two-phase approach where in the first phase it tracks the last stable state, and on training failures forces the robot to start from the failure point. This attempts to speed up reaching stable points, since it is inefficient to repeatedly train the first half without seeing the latter half as often. In the second phase, it looks at places that have seen less training, and trains them more to compensate [2]. Theoretically, this is a very interesting approach to training and we would have liked to numerically compare it to our accelerated SARSA, but they don't provide any data or source.

An article by Xu et al, also from 2017, uses $Sarsa(\lambda)$ combined with deep learning to path plan [1]. This is a much more computationally intensive algorithm than anything we implemented. They follow the standard implementation of $Sarsa(\lambda)$ with qualification traces.

$$\begin{aligned} \delta_t &= [r_{t+1} + r \cdot Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)] \\ Q_{t+1}(s_t, a_t) &\leftarrow Q_t(s_t, a_t) + a \cdot \delta_t e_t(s_t, a_t) \end{aligned} \tag{8}$$

$$e_t(s,a) = \begin{cases} \gamma\lambda\delta_{t-1}(s,a) + 1 & s = s_t, a = a_t \\ \gamma\lambda\delta_{t-1}(s,a) & \text{else} \end{cases} \text{ [1]} \tag{9}$$

Their neural network is Denoising Sparse Auto Encoders (DSAE) based on a traditional three-tier network, the Rumelhart AutoEncoder [1]. This reduces the dimension and noise of the data and is used to find features in the environment. This is passed through a Self Organizing Map network to find the R value of $Sarsa(\lambda)$. The general flow is the environment is observed and passed through the DSAE to interpret the environment. This is fed through the SOM Network to get the reward value. From there standard $Sarsa(\lambda)$ can be performed. Their testing is done on a much larger grid that is much denser then ours. Their training data indicates that they learn almost as fast as we did, on a much more complicated environment. They also prove it in a complex continuous environment with several experiments. Overall this algorithm looks to be

very robust, albeit complicated, but slightly outside the scope of the class. We wish they had done more analysis of each individual part so we had a better understanding of how useful the SOM network is at calculating the R value versus a static reward in standard $Sarsa(\lambda)$.

There are also references to older modifications of SARSA used for path planning. In 2009, Ramachandran and Gupta published their paper "Smoothed Sarsa: Reinforcement Learning for Robot Delivery Tasks" [3]. Their environment is a partially observable Markov decision process (POMDP). This has been built using a Dynamic Bayesian Network and a custom region-based particle filter. This POMDP is what SARSA is choosing actions from. It's worth noting that their state space is so large, being continuous, that they use function approximation. As noted by the authors, the backed-up SARSA estimate isn't very good if the uncertainty is high [3]. This is where their smoothed-SARSA implementation comes in. They delay the backup until the uncertainty is reduced; once the uncertainty is reduced, the estimate is much better. The pseudo-code is shown in the paper if a deeper analysis is desired. We don't believe that this would have been helpful in our implementation, since environmental noise was quite low in our case. Conceptually, this idea is very interesting for a continuous real-time environment. Their experimentation is done in a very similar environment to ours, but significant numerical data is not provided. This paper has been cited only a few times, and it would appear that it is computationally too difficult to be viable.

The final reviewed SARSA implementation was Iterative SARSA, published in early 2020 by Vellore Institute of Technology undergrads [4]. Iterative SARSA is an interesting take on traditional SARSA. Once the robot reaches the end goal of the algorithm, the goal and start are interchanged and it is rerun. Unfortunately it's more computationally expensive than SARSA, and takes longer than Q-learning[4]. They also choose to use a non-annealing exploration for SARSA, and thus it doesn't converge to optimal. This makes both SARSA and Iterative SARSA look worse than they should be. Overall this algorithm does not seem to provide too much value unless the environment is hard to reset and backtracking is a large time saver.

An alternate method of representing state-action pairs is discussed by Sallans and Hinton in their 2004 paper [5]; they use a method of function approximation with an undirected graphical model called a product of experts, and take the negative free energy of a given state-action pair under the model as its Q-value. They then use Markov chain Monte Carlo (MCMC) sampling of the distribution of free energies to take actions and train. This does not guarantee success in training except for the limit under which a large number of actions are taken, but allows training without an agent knowing anything about the function or the model itself. We use a simpler representation in our experiment, but we follow the same principle of building up a map while training and keeping track of series of state-action pairs to create more complicated actions (i.e., traversing from the start to the goal). If we were to expand the size of our map dramatically or increase the granularity (approaching a continuous rather than discrete representation), then we would be more likely to make use of MCMC sampling, since it is particularly effective for large action spaces [5].

A very interesting algorithm which combines our two main approaches, SARSA and Q-Learning, is presented by Wang et al. in their 2013 paper on Backward Q-Learning [6]. This approach discusses the benefits and pitfalls of each, such as SARSA having faster convergence but easily becoming stuck in a local minimum and Q-Learning having better final performance at the cost of longer training time. In looking particularly at the action-selection policy, this paper proposes Backward Q-Learning as an integration of both these methods in an attempt to improve learning speed and final performance. This algorithm involves directly tuning the Q-values of each state-action pair, and allowing these values to only indirectly affect the agent's action-selection policy. It would be interesting to see this algorithm implemented for a similar situation as ours to see if it leads to a noticeable improvement, or to what degree we would need to increase our complexity to see a difference. The authors of this paper do discuss some example applications which go into the feasibility of the algorithm and determine that it does outperform standard Q-Learning and SARSA in their case [6].

VIII. SUMMARY AND FUTURE WORK

In this project, we implemented and tested standard implementations of some common reinforcement learning algorithms, SARSA and Q-Learning, and compared them to some variations as well as some heuristic agents. We determined that each of our hypotheses was correct, and all learning agents were able to learn a path within 2 steps of the optimal path, outperforming the random and potential fields agents and coming close to the ideal A* agent. All training and validation was done using Python with ROS Melodic and its Stage simulator in Ubuntu 18.04. Data was written to *.csv* files and then grouped and plotted using a custom R function. Our GitHub repository can be found here.

The first piece of future work would be expanding our tests to push the limits of Double-Q and Expected SARSA. It would be interesting to push our algorithms into situations for which they weren't adapted well and see then if the more complex variations fare better or continue to learn comparably. Secondly, we would per-

form a more thorough investigation into the asymptotic nature of Accelerated SARSA to get a better idea of its behavior. Our environment is very slow to train, so we wish to investigate other ways to speed it up like improving Accelerated SARSA and possibly some of the ideas from the related work section.

These algorithms provide the flexibility of running without prior knowledge of the environment given the time to train. This is a valuable skill in unknown environments. RL proved to be a reliable solution, with statistics to match the traditional methods. The major downside is the amount of time it takes to train. Overall, this project provided a deeper look into RL path planning and how it compares to traditional path planning.

[1] D. Xu, Y. Fang, Z. Zhang and Y. Meng, *Path Planning Method Combining Depth Learning and Sarsa Algorithm*, 2017 10th International Symposium on Computational Intelligence and Design (ISCID), Hangzhou, 2017, pp. 77-82, doi: 10.1109/ISCID.2017.145.

[2] R. Meyes, H. Tercan, S. Roggendorf, T. Thiele, C. Büscher, M. Obdenbusch, C. Brecher, S. Jeschke, and T. Meisen, *Motion Planning for Industrial Robots using Reinforcement Learning*, Procedia CIRP, 11-Jul-2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S221282711730241X. [Accessed: 03-Nov-2020].

[3] D. Ramachandran and R. Gupta, *Smoothed Sarsa: Reinforcement learning for robot delivery tasks*, 2009 IEEE International Conference on Robotics and Automation, Kobe, 2009, pp. 2125-2132, doi: 10.1109/ROBOT.2009.5152707.

[4] P. Mohan, P. Narayan, L. Sharma, T. Jambhale, and S. Koul, *Iterative SARSA: The Modified SARSA Algorithm for Finding the Optimal Path*, International Journal of Recent Technology and Engineering (IJRTE), vol. 8, no. 6, pp. 4333–4338, Mar. 2020.

[5] Brian Sallans and Geoffrey E Hinton, *Reinforcement Learning with Factored States and Actions*, Journal of Machine Learning Research, 5(Aug):1063–1088, 2004.

[6] Yin-Hao Wang, Tzuu-Hseng S Li, and Chih-Jui Lin, *Backward Q-Learning: The Combination of SARSA Algorithm and Q-Learning*, Engineering Applications of Artificial Intelligence, 26(9):2184–2193, 2013.