Multi-Agent Rendezvous Using Reinforcement Learning

Sarah Mitchell [*], Gadiel Sznaier Camps [†], and Juan Martinez Castellanos [‡]
Stanford University, Stanford, CA, 94305

Our objective is to design a problem with *n* robotic agents that are attempting to autonomously rendezvous to a desired final formation. To incentivize the agents to make decisions that will ultimately guide them to the desired final formation, we imposed an L2 norm cost to penalize the agents more if they were farther away from the desired location. To make the problem non-deterministic, two forms of random noise were introduced to the transition matrix: randomly retarding noise, and a simplified wind model. A Hooke Jeeves local search approach is shown to have great results and fast runtime under significant additional noise, while a Q-Learning approach is shown to take significantly longer to account for exploration but benefits greatly from nearest neighbor interpolation.

All of our code can be found here: https://github.com/jam14j/AA228_final_project

I. Introduction

Problem

The project we developed solves a Multi-agent Reinforcement Learning problem. We have *n* robot agents which are attempting to rendezvous to a desired final shape formation. The agents move in two dimensions, and each one has a fixed start point and end point. In this sense, we are essentially running *n* independent agents side-by-side. However, the robots are intrisically tied together by sharing one global cost function. This is created by having a penalty value imposed upon each agent that will incentivize the agents to make decisions that will ultimately guide them to the desired final formation.

The main decisions that our agents will have to make is determining in which direction they want to travel and figuring out where their final destination is supposed to be. The uncertainty in our project primarily comes from the fact that the agents do not know the cost function by which their states are being evaluated. Moreover, each agent does not have an individual cost function pertaining to its state. Rather, every agent has access to the same global cost function, which is a collective cost based on all their states combined. The robots will have to either approximate this cost function or otherwise decipher how their movements contribute to the overall cost, and thus how they can take the optimal actions to reduce the cost and bring them closer to their final destination.

Literature Review

Multi-agent robot planning is a problem that builds on several modern decision making methods, and is crucial for systems that need multiple agents to work together as a team. Such examples include satellites in orbit coordinating optimal signal coverage, or robotic swarms cooperating for widespread disaster relief [1]. However, these problems can be challenging in many ways. Multi-robot systems can be designed in a global fashion or in a distributed fashion. Global systems are those in which all the agents share a common body of knowledge about the systems, including the location of all the other agents. Distributed systems, on the other hand, are when each agent does not know the positions of the other agents in the system exactly, and therefore must operate independently of one another. This type of system is much more challenging, because each agent has much less information informing its decisions, and must be able to handle much more uncertainty.

Our problem, described above, has taken inspiration from a similar project done by Rubenstein et al [2]. Their project programs a thousand-robot swarm to self-assemble into a particular formation, similar to ours. However, we have made many simplifications in order to make this idea more reasonable for our final project. Rubenstein describes the

[*]MS, Electrical Engineering, scmitch@stanford.edu
[†]MS, Aeronautics & Astronautics, gsznaier@stanford.edu, AIAA Student Member
[‡]MS, Aeronautics & Astronautics, jam14j@stanford.edu

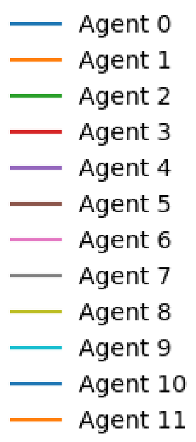

Fig. 1 Most of our figures represent the path of 12 agents. Having a long legend in each figure would be obtrusive; instead, we show it here once.

project with a distributed system of knowledge, and also furthermore explains that the "robots have no direct information about their global position, only the distances to nearby neighbors." Although we are using a distributed system for our knowledge, we limit much of the complexity of a multi-robot system by removing all interactions between robots. We are also assigning each agent a hidden predetermined final state, whereas Rubenstein's swarm may have any particular robot end up anywhere in the shape, as long as the overall swarm assembles into the formation.

One last point of complication that occurs in multi-agent systems is a concept referred to as credit assignment, as discussed by Nguyen, Kumar, and Lau [3]. Credit assignment is the problem that arises when many robots are moving at the same time, but only have access to a cost function that describes the global state of the system. For example, if half of our robots move closer to the goal and half of them move farther away, we might not know which robots improved the score and which ones made the score worse. Our solution to this issue will be addressed later in our methods section.

On a final note, the algorithms that we implemented in this paper were taken from the AA228 "Decision Making Under Uncertainty" class book, *Algorithms for Decision Making* [4].

II. Methods

Cost Function

Initially we had considered using a separate cost function for each of the agents, but ultimately we decided against this because having separate cost functions is essentially equivalent to running a completely independent simulation for each agent. We wanted our agents to work together, not in parallel. Thus, our cost function evaluates the global cost of the current agent configuration. The cost for all of the algorithms implemented represents the norm of a vector comprised of the distances between each agent and their desired position:

$$C = \|[d_1 \; d_2 \; d_3 \; \ldots \; d_n]\| \tag{1}$$

$$d_i = \|x_i - x_i^*\| \tag{2}$$

We define C as the cost, d_i as the distance between the i^{th} agent and its desired position, x_i as the i^{th} agent's current position, and x_i^* as the i^{th} agent's desired position.

Additional Noise

In order to test the robustness of each of our methods, we added two sources of additional noise error to our agent's movement. First, we added a retarding error, where each movement that an agent makes has a 10% chance of not moving. Second, we added wind to the simulation that makes it so that each action has a 25% chance of moving to the right instead of the desired direction. For each method we implement, we can compare robustness based on how it responds to each type of noise.

Hooke-Jeeves

Our initial method to move each of the n agents is a local search approach using the Hooke Jeeves algorithm. As mentioned before, the desired destination for each agent is not known by the agents themselves, but it *is* known by the cost function. Since we are working in two dimensions, each agent chooses greedily between four actions: moving a step size $\alpha = 0.1$ in the directions north, south, east, or west. The algorithm greedily chooses the direction that decreases the global cost by the largest amount for each agent at each step. Initially we tested this method with 12 agents starting in a small circle and spiraling outwards into their respective goals. Figure 2 shows the outcome of this simulation under 3 different conditions. Each of these simulations runs in under a second, with an average of 0.7 seconds runtime.

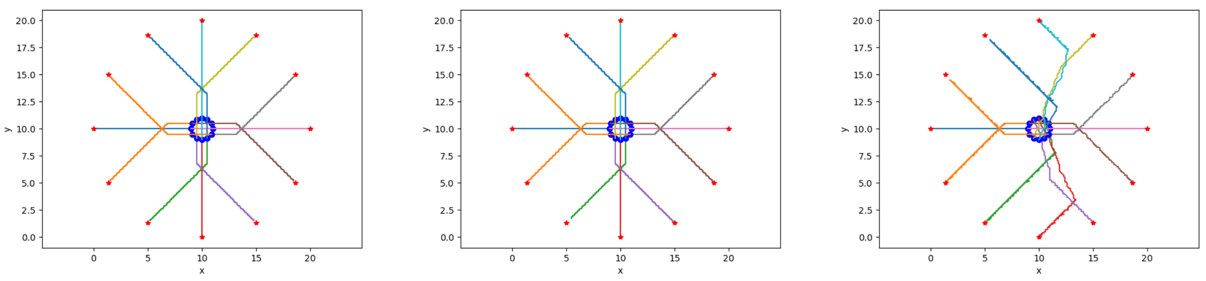

Fig. 2 12 agents moving from a small inner circle to a bigger outer circle. The goal positions are rotated by 6 positions, compared to the starting positions. From left to right, these are the simulations with no additional noise, with retarding noise, and with both retarding and wind noise

This method succeeds at reaching the desired formation even under large amounts of error introduced by wind. The retarding error makes it so that some of the agents do not quite make it to the goal, such as the green agent on the bottom left. The wind error significantly alters the path taken by the agents, but it does not prevent them from reaching the goal. Hooke Jeeves works particularly well in our situation because it directly evaluates the utility function, which in our case corresponds to the negative of the cost function. This is convenient because the reward in our problem only depends on the current state configuration instead of actions. Another reason that Hooke Jeeves works well for us is that we have only one globally optimal final position for each robot. There are no local optimums to get stuck in, which is one potential downside of Hook Jeeves that we need not worry about.

Since the algorithm worked surprisingly well, we decided to test its limits by simulating more complex systems. First, we doubled the number of agents and changed the goal shape from a circle to an ellipse. Second, we doubled the number of agents again and changed the goal shape to a Koch snowflake. From the results of these simulations, shown in figure 3, we concluded that the Hooke-Jeeves local search method works well even for very complex formations.

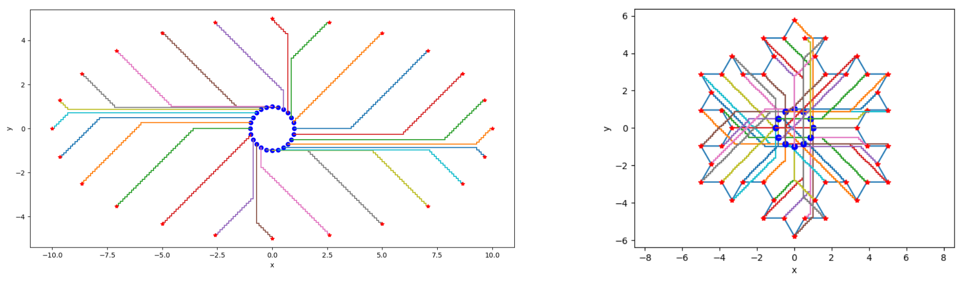

Fig. 3 Left: 24 agents moving from a circle to an ellipse rotated by 3 positions. Right: agents moving from a small inner circle to a bigger Koch snowflake, where the goal of each agent is determined by the index of the snowflake's vertices.

Q-Learning

The next method we tried was Q-Learning. Q-Learning is a model-free reinforcement learning approach which performs an iterative estimation of the action-value function $Q(s, a)$. In our case, state s is a linearly-indexed integer value describing our absolute location on a discretized grid world, and action a is the index of our possible actions, in the order north, south, east, west. To learn the values in the Q matrix, we must first go through an exploration phase. For each new sample in exploration, we use the incremental update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\Big).$$

We used a fixed learning rate α and discount factor γ. These are also good opportunities to explore variations in performance, although we did not experiment with these values due to the length of time it takes us to produce a good trial of our problem.

Since each of the agents will require a different policy to move to their separate destinations, we need to have a different Q matrix for each agent. Technically, all agents could obey the same Q matrix, but this would require entirely redefining our states with coordinates relative to their final destinations. This becomes much more complicated when the final state is unknown and only the cost function is available to guide the agents. Thus, we decided to stick to our plan of having a separate Q matrix and a separate policy for each of the agents.

Our approach to Q-Learning is to simplify the overall process by doing a single phase of random exploration to learn the Q matrices, and then creating a fixed policy for each agent to follow from its initial position to its goal position. We can make this simplification, namely the decision to stop updating the Q matrix during travel, by taking advantage of our knowledge of this particular problem. We know that each agent's optimal solution is to move in a straight line from the initial position to goal position, and that we will not gain any benefit from continuing to update the Q matrix in our wake as we travel. Thus, we can avoid the computational complexity needed to update Q after the exploration stage. One potential pitfall, however, is that this assumption leaves us open to false movements and rewards if our Q function is not as accurate as we thought it was. Because we are not expecting to exactly reach all of our target positions with this error in mind, we will have our agents follow their learned policy for a fixed number of iterations rather than wait until the agents converge to their final destinations. To deal with the problem of credit assignments of our total

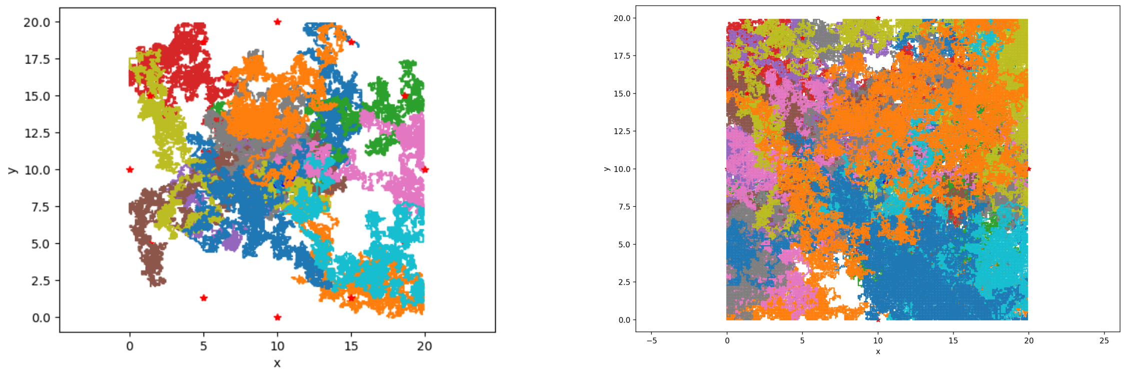

Fig. 4 Simulation of the random exploration. From left to right, random exploration with 10,000 steps and 50,000 steps

cost, we are able to extract an equivalent individual reward value for each agent's actions, even though the cost function is cumulative. We achieve this by only moving one agent at a time and looking at the difference in total cost. First we send the current state configuration into the cost function as a baseline. Next we look at just one agent and do a random action from that state (during the exploration phase) to get a new cost. We then subtract this new cost from the baseline cost to get the incremental cost associated with that individual agent's action, which is the sample we use for the Q-Learning update step.

In Fig 4, we see the results of exploring for 10,000 iterations and 50,000 iterations. While most of our world is covered by the combined agents' random movements using 50,000 iterations, we also notice that any given agent's color does not cover the entire map. This means that each agent has many unexplored states, and as a result the policy may not be fully representative of the true world behavior.

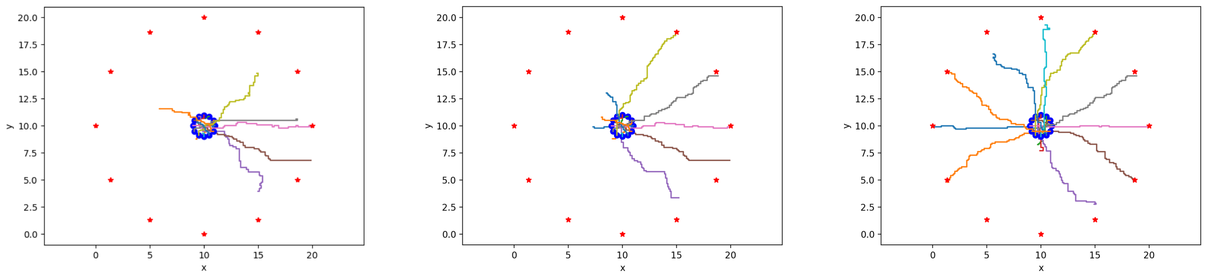

Fig. 5 These plots were created using the optimal policy obtained immediately after exploring the map. No interpolation using nearest neighbor was done to augment unexplored states. From left to right, the plots are: 100,000 iterations, 200,000 iterations, and 500,000 iterations of random exploration.

In Fig 5, we show the results of computing a policy after the random exploration phase, with increasingly more iterations of exploration. For the smaller iterations, shown on the left, we have not explored enough of the map. Many agents get trapped in small cycles of actions near the center of the map within the small starting circle, because their random actions have not yet made it to the outer reaches of the map. For larger numbers of iterations, we see that more agents have explored a larger portion of the map, and as a result their policies are more optimal. Theoretically, we could keep lengthening the duration of our exploration phase until all agents create policies that bring them near their true destinations. However, the downside of this method is runtime. 100,000 iterations (shown on the left of Fig 5) takes about half an hour to run, while 500,000 iteration (shown on the right) takes about 2-3 hours. We would need an even longer exploration time than this for all agents to have reasonable trajectories.

In the end, the downfall of our exploration method is that we are not guaranteed to explore every state, and it takes a long time to make an accurate policy by exploring randomly. An alternative approach is to spend less computational time exploring, and instead approximate the empty entries in our Q matrices by using interpolation. Because we are using norms for our cost function, we know that every visited state-action pair in Q will have a nonzero positive cost, or equivalently a nonzero negative reward. After exploration, we know that any pair with $Q(s, a) = 0$ has not been visited, and we should approximate this entry with the existing data that we have collected. The simplest method for this approximation is the nearest neighbor approach, with $k = 1$.

We first tried to implement the nearest neighbor approximation using for-loops, but this proved intractable for our grid resolution of 0.1 (which creates 40,000 states), because we had to compare every unvisited state with every other state in order to find the nearest neighbor. To calculate these approximations using nearest neighbor more efficiently, we use some clever tricks to combine the agents' Q matrices together in a such a way that allows us to use Python's matrix operations instead of for-loops to speed up complexity.

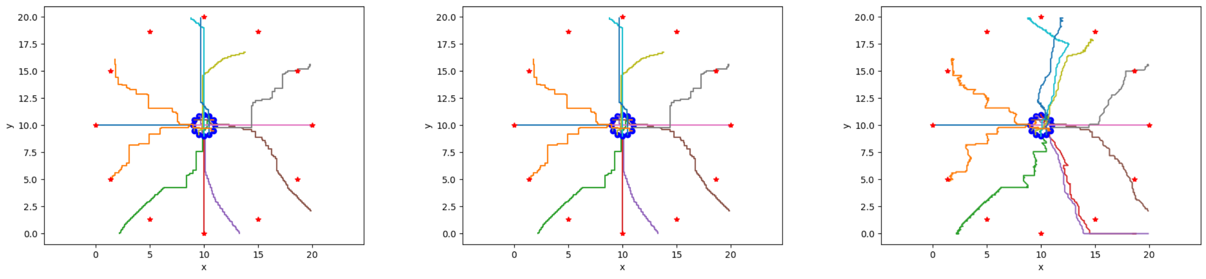

Fig. 6 Agents obeying optimal policies generated from interpolated state action matrix,generated after exploring randomly for 50,000 iterations. From left to right: using no noise, only retarding noise, then both retarding noise and simplified wind noise.

We begin by stacking each agent's Q matrix into a single three-dimensional tensor of dimension ($|S|$ x $|A|$ x n). For each action, we can then find a 2D vector listing all states that have taken this action; each row represents a different robot. To make sure that the operations are associated with the correct agent, we then sort the vector based on agent ID using `argsort()`. This matrix is used to create a new matrix containing all explored states and a corresponding matrix containing those states' real world coordinates. For any given action, the number of states in which that action was performed is different depending on the agent, due to random exploration. That is, Agent 1 might have performed action 3 in 10 different states, but Agent 2 might have performed action 3 in 12 states. This difference requires us to pad the state matrix with zeros, and the real world coordinates matrix with invalid coordinates, in order to obtain two uniform matrices. The padded coordinate matrix is then subtracted from the position of the current state and passed through the L2 norm to obtain the Euclidean distance from every sampled state to the current state. A row-wise `argmin` is then used to find the index location where minimum distance was found for a given agent. This is then used to obtain the corresponding state from the padded state matrix. Finally, the vector of closest states for each agent for the current action is used to obtain the reward value, which is in turn used to set the reward for the currently examined state. Once the Q matrix is fully updated, the 3D tensor is distributed back to each agent for use.

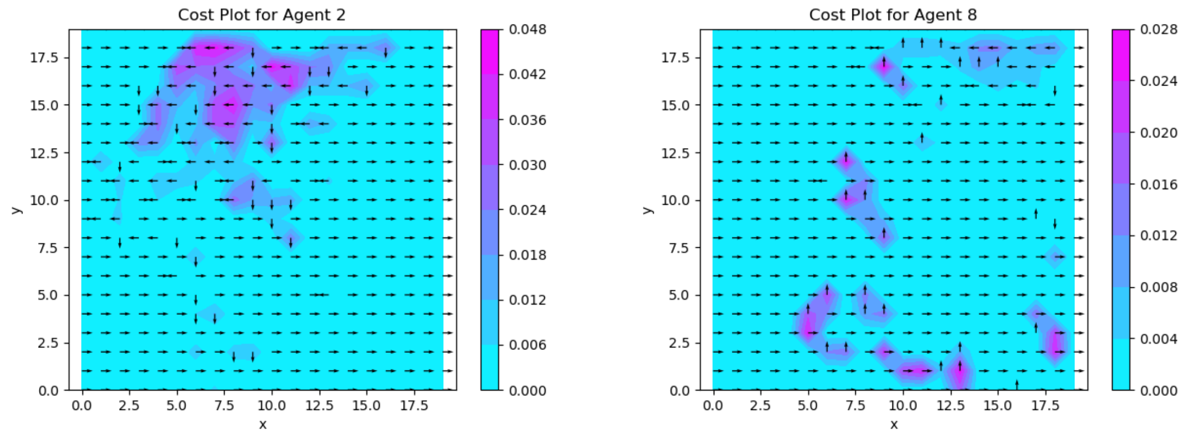

Fig. 7 Q matrices generated by nearest-neighbor interpolation. The arrows represent the action chosen at each state.

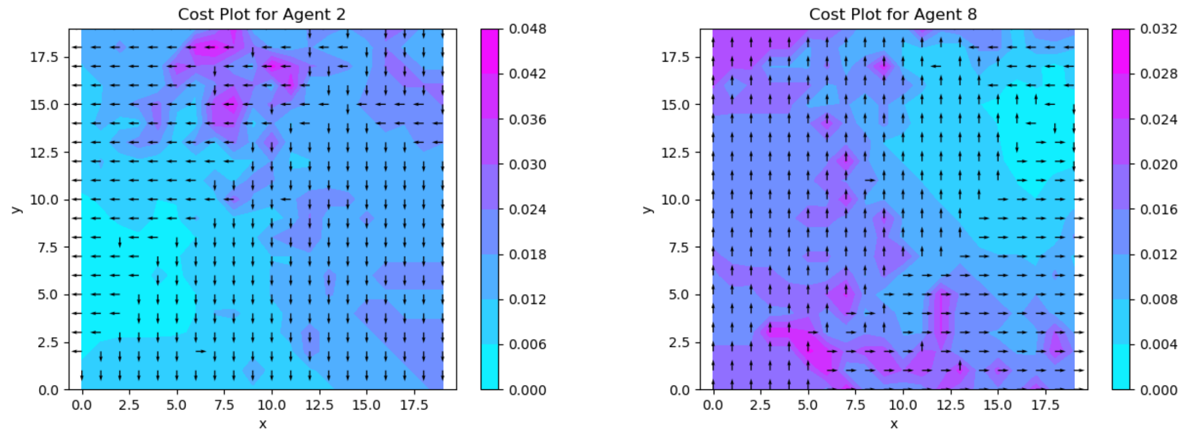

Fig. 8 Q matrices generated by nearest-neighbor interpolation. The arrows represent the action chosen at each state.

Using this method, we then extracted the optimal policy for each agent after only 50,000 iterations of exploration. The resulting policies, with and without noise, are shown in Fig 6. When comparing Fig 6 with Fig 4 (no interpolation) we can see that this approach is much better than simply performing random exploration for either 100,000 or 200,000 iterations. For the interpolated policy without noise (Fig 6 on the left), we do see some nonoptimal policy actions when agents overshoot their target, such as the green agent. This must be a side effect of interpolation, because such issues are not seen on the most-explored version of Fig 4. However, this inaccuracy is more than made up for in runtime. The methods with interpolation ran significantly faster, since exploring for 50,000 takes only 15 minutes to run, and the additional interpolation using nearest neighbors only adds another 10 minutes. This illustrates the importance of using interpolation schemes like nearest neighbor to augment incomplete knowledge of an environment during reinforcement learning.

To further illustrate the difference that interpolation makes, we chose 2 exemplary agents (2 & 8) and plotted the contour of their state-action matrices before and after interpolation, where the areas of highest and lowest reward are clearly shown on the color-coded regions. We have also overlaid the optimal action from each state. The results are shown in Fig 8. We can first note that for two different agents, the contour plots are very different. This makes sense as each robot has a different objective that it is moving towards; this difference in goals is encoded in the cost function. Furthermore, we see that most of the contour before interpolation is colored cyan. This corresponds to 0 cost, which means that we have not yet visited these states. As a result, the policy selects the default action when no information is present, which is to move east. However, after interpolation we see that the nonzero cost values have spread across most of the map, resulting in a much more informative policy. Now we see that the region of near-zero cost is only present in the regions surrounding each agent's target destination, and the majority of the policy arrows are pointing in that direction.

III. Conclusion

This project aimed to guide *n* agents from an initial formation denoted by a small circle, to a final formation denoted by a larger rotated circle. The navigation of each agent was to be executed without explicit knowledge of the location of the goal. Instead, each agent had to develop and follow a policy to find its respective goal. These policies were developed based on a global cost function that assigned more cost to formations that were farther away from the desired formation.

Our first solution attempt was a Hooke Jeeves local search algorithm that greedily chooses the next action that each agent must take to reduce the global cost by the largest amount. This method worked well for our base problem, and also worked well for more complex shapes. It was successful even when tested with additional sources of noise error, and resulted in agents arriving at their final destination with high precision.

Our second solution attempt was a Q-Learning algorithm, where each agent independently estimates an action-value matrix and generates a corresponding policy based on that matrix. The action-value matrices are estimated by initially performing a random exploration phase, and using those random samples as inputs to our iteratively update our estimation function. The policy obtained from this procedure was never successful for all our agents. At most, it was successful for 10 our of the 12 agents after 500,000 steps of random exploration, at the cost of multiple hours of runtime. Because of this, we implemented a nearest neighbor algorithm to improve each agent's action-value matrix. The results obtained with this interpolation technique were much better. All of the agents moved towards their goal and were fairly resilient under additional sources of noise. However, many of the agents still did not reach their goal.

IV. Group Member Contributions

We had a team of three working on this project, consisting of Sarah, Juan, and Gadi, who all worked together in equal parts. Juan wrote the initial Hooke-Jeeves algorithm and noise additions, which everyone then helped debug. Sarah wrote the random exploration function, finding the optimal policy, and initial nearest neighbors approximation, while Gadi wrote the Q-Learning update function and the functions converting back and forth from linearly indexed states and world coordinates. He also wrote the computationally sped-up version of the nearest neighbors algorithm (debugged with Juan), allowing us to run the interpolation algorithm in a reasonable amount of time. Sarah met with our TA Jean twice to trouble shoot various issues that the group ran into, and Juan ran most of our simulations. We all worked equally on the write up of the paper.

References

[1] Stormont, D. P., “Autonomous rescue robot swarms for first responders,” *CIHSPS 2005. Proceedings of the 2005 IEEE International Conference on Computational Intelligence for Homeland Security and Personal Safety, 2005.*, IEEE, 2005, pp. 151–157.

[2] Rubenstein, M., Cornejo, A., and Nagpal, R., “Programmable self-assembly in a thousand-robot swarm,” *Science*, Vol. 345, No. 6198, 2014, pp. 795–799.

[3] Nguyen, D. T., Kumar, A., and Lau, H. C., “Credit assignment for collective multiagent RL with global rewards,” *Advances in Neural Information Processing Systems*, 2018, pp. 8102–8113.

[4] Kochenderfer, M., Wheeler, T., and Wray, K., “Algorithms for Decision Making,” 2020.