Grid-based Cooperation Environment

Ahmed Jaafar, Gregory Smelkov

December 2022

1 Introduction

For this project, we are experimenting with a new environment that requires two agents to work together to reach the goal in the shortest number of steps possible. Then the environment will be tested to see what policy and value functions can be extracted through a variety of different experiments and several different algorithms.

2 Environment

The environment is based off of the 687-Gridworld. It uses a similar movement options and a slightly larger grid but adds a few new interactable elements.

Figure 1: Final version of environment map

2.1 Iteration

The process for creating a good and interesting environment took many steps and experimentation. Some of the big changes are listed below.

Some of the big changes visible throughout these steps stem from the initial bots struggling to actually find the collaborative solution. Adding the walls between steps a and b allowed better testing of the models to ensure that it was actually possible for the collaborative approach to occur naturally.

Removing the walls, and moving Bot1 down two blocks (b to c) ended up being the missing piece to getting consistent results in finding the collaborative path. It made the distance to go around the walls just long enough to motivate exploring. It also made it a lot more likely that both bots would be in the correct places at the right time to make the collaboration happen.

After seeing very positive results with that, experiments with decreasing the water reward proved very successful to the point that the results for Monte-Carlo show an incredibly interesting learning approach as it finds both the easy path and then the collaborative path. This is explained further in section 3. Due to that, the negative reward for the water tile was entirely removed in the final version (c to d) as it was deemed no longer necessary.

Our final iteration of this environment leaves a very clear optimal path that requires both agents to step right so that one of them hits the button while the other one passes through the door. The alternatives to that are to go around the walls which is simpler to accomplish because it doesn't require the precise timing or patience that the optimal path takes.

Figure 2: Iterations of environment (a, b, c, d)

2.2 Old elements

Some of the elements are carried over from 687-Gridworld

2.2.1 Goal

The goal is the end state for this environment. To end the episode one of the agents needs to reach the goal. It also provides a large positive reward to encourage the bots to make an attempt to reach it.

2.2.2 Wall

A wall is an impassable space. Agents are unable to enter them and they serve to provide some structure to the map and act as obstacles to avoid. In addition to the walls visible on the map, it can be assumed that the tiles outside of the environment are all maps, keeping the agents in.

2.2.3 Water

The water tile motivates bots to avoid it with a negative reward but otherwise allows the agents to walk through it. We ended up removing this because it made it too easy to find the optimal path.

2.3 New elements

2.3.1 Button

The first of the new objects is a button. In this environment, the button is wired to the door such that the door is only open if an agent is standing on top of the button. Once that agent leaves the button, the door does not stay open.

2.3.2 Door

The door acts like a wall until it receives a signal to be open. It does not allow agents to step onto it if it is closed and if an open signal is not maintained, it will shut. If an agent is in the door while it is shutting, that agent will be able to move off of it. This can be visualized as though the agent in the doorway is holding it open until they step out of it when it shuts.

2.3.3 Second Agent

The largest difference addition is a second agent. The agents are shown in the picture of the environment as Bot1 and Bot2. They have their own individual value functions and policies, but share rewards as only one of them needs to reach the goal. The agents are also not allowed to share the same space, and although they are not aware of each other, if

one of them tries to take a step into a state that has the other bot, it will act like there is a wall there and stay in it's current spot.

2.4 MDP

2.4.1 State

For the MDP that goes along with this environment, the state is simply the position of each of the agents. Since each agent has a separate value function and policy, their states don't include the positions of the other bots. The goal is for them to learn to cooperate without specific knowledge of each other's locations. The state also doesn't include whether or not the door is open as with how simple an environment this is, it is probably inconsequential for Bot2 to try to walk through the door until it is open.

2.4.2 Actions

The actions are the same as 687-Gridworld: attempt up, attempt dowm, attempt right, attempt left. Each of these would make the agent try to take a step in the corresponding direction and would fail if there was something blocking that movement such as a wall, door, or the other agent.

2.4.3 Environment Dynamics

To simplify these experiments, the environment dynamics have been changed so that the agent will always try to move in the direction that it's policy specified, without getting confused and going left and right like in the original 687-Gridworld. This changes it from a stochastic model to a deterministic one.

2.4.4 Rewards

The rewards are also being carried over from the original 687-Gridworld. The goal will provide a positive reward of +10 when one of the bots reaches it, while the water tile will provide a negative reward of -10 whenever a robot steps in it. This is one of the areas we want to experiment with to see how different rewards will affect the amount of cooperation that the robots will display. We ended up removing the water tile entirely from the map as the negative reward it provided made it too quick to find the optimal state, and not having the water adds some complexity that highlights the need for cooperation.

3 Monte-Carlo (Greg)

3.1 Implementation

This algorithm implemented the MDP in Python. Since the goal of this problem was to experiment with multiple independent agents, this MDP's state only includes the positions of each agent and the status of the door.

The way I implemented the interactable objects is through arrays of their coordinates, for example the walls are handled by initializing the array 'walls' with '[(1, 2), (3, 2), (2, 3), (4, 2)]' which is the coordinate pair for each wall should be. This makes it very easy to move walls around while experimenting with different environment variations.

The door's implementation is relatively simple, it is treated like a wall in the move function. By treating it like an open space otherwise it allows a more accurate estimate of the value of that state to be formed. The door's status is tied to the button and is implemented by setting the doors coordinates to (-1, -1) whenever an agents status is set to the button coordinates, and back to (5, 2) when the agent's status is set to not the button. This makes it very easy to check when the door is open.

The multiple agents are also able to collide with each other which is handled similarly only in the move function so that it wouldn't affect their policy calculations too much as they would likely have a reason to move there despite the obstruction.

The environment dynamics were changed to be deterministic instead of stochastic because it allows for more consistency and due to the complexity with having two agents, it improves on the completion rates. It also simplifies some of the big changes I needed to make to accommodate two agents and objects with varying states.

3.2 Analysis

3.2.1 Value function and policy

Bot1:

Value function:

3.7715	4.2386	4.8181	5.3776	5.9651
3.7301	4.0474	0.0000	5.8562	6.5995
4.7594	5.5022	7.2229	0.0000	7.4204
3.6620	3.9136	0.0000	7.5779	8.3317
3.5024	4.1495	0.0000	8.3812	9.5184
4.8185	5.8313	8.1963	9.5124	0.0000

Policy:

→	→	→	→	↓
↓	↓		→	↓
→	→	↓		↓
↑	↑		↓	↓
↓	↓		→	↓
→	→	→	→	G

Bot2:

Value function:

3.7769	4.2395	4.8192	5.3782	5.9654
3.7275	4.0455	0.0000	5.8572	6.5997
4.7596	5.5021	7.2230	0.0000	7.4202
3.6620	3.9139	0.0000	7.5720	8.3316
3.5034	4.1491	0.0000	8.3825	9.5190
4.8181	5.8308	8.1960	9.5123	0.0000

Policy:

→	→	→	→	↓
↓	↓		→	↓
→	→	↑		↓
↑	↑		↓	↓
↓	↓		→	↓
→	→	→	→	G

The final value functions for each of these was able to correctly identify the shortest path to the goal, which requires one agent to press the button while the other goes through the door. This is especially interesting when visualized against the policy retrieved after only running for 1,000 episodes instead of 10,000 like the ones above.

Policy (Bot2, 1000 episodes):

$$\begin{array}{|c|c|c|c|c|} \hline \rightarrow & \rightarrow & \rightarrow & \downarrow & \downarrow \\ \hline \rightarrow & \uparrow & & \rightarrow & \downarrow \\ \hline \rightarrow & \uparrow & \rightarrow & & \downarrow \\ \hline \rightarrow & \uparrow & & \downarrow & \downarrow \\ \hline \uparrow & \uparrow & & \downarrow & \downarrow \\ \hline \uparrow & \uparrow & \rightarrow & \rightarrow & G \\ \hline \end{array}$$

At this point the value for the door is still 0.00 which indicates that it hasn't been discovered yet. The bots will choose to take the long way around , even if it is the bot that starts in the bottom left corner. With more training, the bots end up finding the shorter path and shift their policy to almost always use it. This can be seen in the graphs from the epsilon experiments below.

The policies for 10,000 steps also show the value functions produce safeguards against an exploratory step to either direction, bringing the bot back to that optimal path of pushing the button and going through the door.

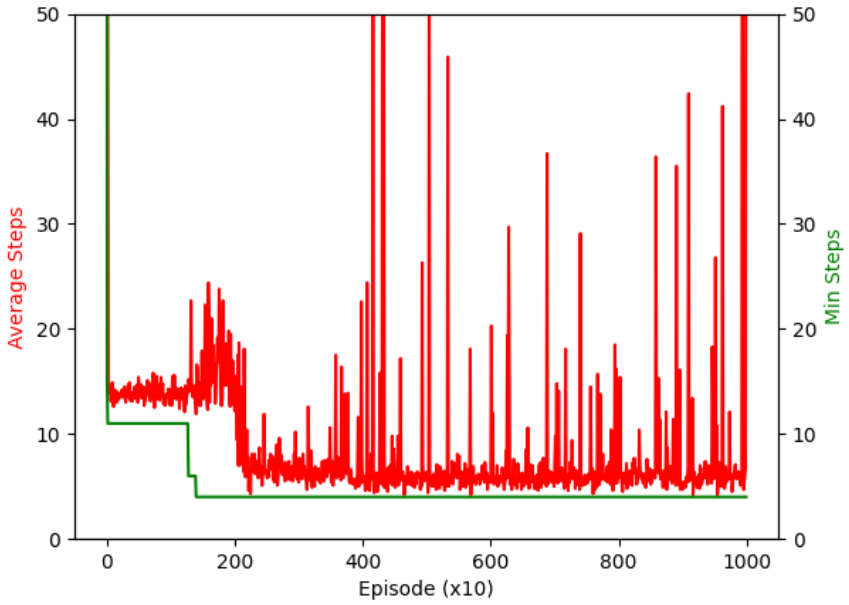


Figure 3: $\epsilon = 0.2$

3.2.2 Epsilon experiments

Fairly quickly it became evident that varying the value of epsilon played a large part in how fast it was able to discover the better policy.

In Figure 3, which is run with an epsilon of 0.2, shows clearly that the model is able to find the long way around the walls before it finds the better path. It only takes a 5 or 6 episodes to find the long path, 11 steps at the minimum. At around 1300 episodes, it makes a breakthrough and finds the shorter path. In this case it is very visible how it still doesn't find the best path but one close to it. For the next 700 episodes it struggles to learn the best way to consistently achieve the optimal run every time until around episode episode 2000 where the average runtime drops significantly. The big spikes are likely a results of a relatively high epsilon so the agents tend to veer off the simple path fairly often to run in circles until they get back on track.

Figure 4 Shows what happens with increasing epsilon. Due to the environment dynamics being deterministic, if $\epsilon = 0$ the episodes don't actually end up finishing. $\epsilon = 0.05$ ends up not finding the better path within 10000 episodes. As the epsilon goes up, there are a couple visible patterns. First, the point where the runtime drops to the optimal gets sooner and sooner. That is because it needs to find the button and door at the same time first, and higher values of exploration are more likely to accidentally be in the right place at the right time. Second, the amount of noise that occurs after the drop. There is usually a few hundred episodes after the drop of relatively good performance followed

by a number of spikes as the cooperation fails. Based on these graphs, the amount of noise seems to happen earlier and earlier as the epsilon increases. The consistency of the noise increases significantly as well.

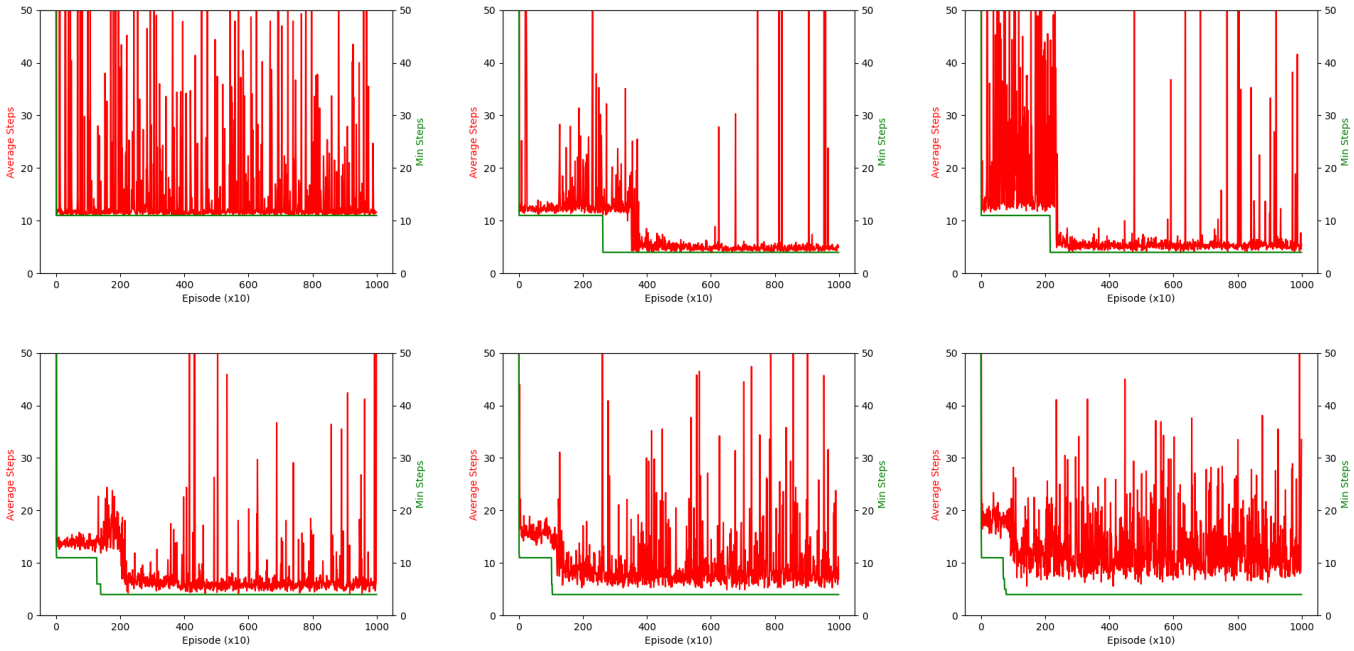


Figure 4: $\epsilon \in [0.05, 0.1, 0.15, 0.2, 0.3, 0.4]$

4 Q-Learning (Greg)

4.1 Implementation

This was implemented very similarly to Monte Carlo except for a few small changes. Q is being tracked with an object using state and the action as the key. Aside from switching the algorithm, everything else in the model functions the same. I experimented with adding a small negative reward for each step taken to encourage it to find the shortest path faster but after changing the hyper parameters around it started finding it fairly consistently, almost all the time. Occasionally it still gets stuck on the longer path around and perhaps with enough episodes it would naturally resolve itself. Decreasing gamma to 0.8 from 0.9 made it much more likely to find that shortest path.

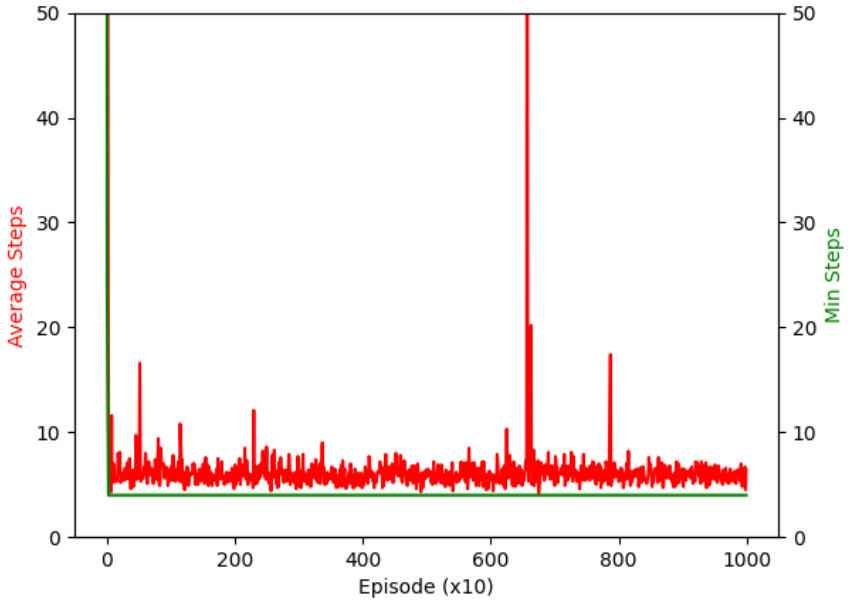


Figure 5: $\epsilon = 0.2, \alpha = 0.1, \gamma = 0.8$

4.2 Analysis

Figure 5 shows the optimal path being found very quickly with similar levels for epsilon as MC had. I had to raise alpha to 0.1 from 0.03 (what I had for the original 687-Gridworld) for it to get better results with more consistency. The values and policies for that graph are below.

Bot1:

Value function:

7.2984	7.8950	1.3847	2.2855	1.4480
9.6525	11.9620	0.0000	3.3973	4.7341
11.5075	13.8725	17.5916	0.0000	6.2196
9.4876	11.7726	0.0000	1.9016	7.9456
6.7533	8.8023	0.0000	3.0717	9.9845
1.0653	0.7103	0.0000	7.3786	0.0000

Policy:

↓	↓	→	↓	↓
→	↓		→	↓
→	→	↓		↓
↑	↑		→	↓
→	↑		↓	↓
↑	↑	←	→	G

Bot2:

Value function:

0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.3445	0.0000	0.0000	0.0000
2.0541	2.2222	0.2041	0.0000	0.6907
2.9111	3.6675	0.0000	3.8580	5.0740
3.8760	4.3699	0.0000	8.0000	6.3999
4.5442	5.5076	8.0000	10.0000	0.0000

Policy:

←	←	↓	←	←
↓	↓		←	→
→	↓	↑		↓
↓	↓		→	↓
↓	↓		↓	←
→	→	→	→	G

Some of the experimenting with ϵ and α resulted in this interesting policy and values below where both agents swapped places. This was more efficient than going the long way around but still significantly slower than just going directly right. If you follow the paths from the start points of each agent (2, 0) and (5, 0) you can see that Bot2 goes left, then up towards the button, while Bot1 goes down then right towards the goal. This would require Bot1 to wait for one turn at the door but it determined that that was still better than going around.

Bot1:

Value function:

1.7266	2.5927	0.3832	−0.1306	−0.1096
3.6079	3.1701	0.0000	−0.1043	0.1790
4.2139	3.5999	1.9086	0.0000	3.1318
4.8406	4.0080	0.0000	7.6252	8.6518
5.5158	5.8903	0.0000	8.8550	9.9500
6.2012	6.9307	8.8550	9.9500	0.0000

Policy:

→	↓	←	←	↓
↓	←		→	↓
↓	←	←		↓
↓	←		↓	↓
↓	↓		→	↓
→	→	→	→	G

Bot2:

Value function:

9.1526	16.8438	2.8054	−0.0199	−0.0199
16.9370	18.6979	0.0000	−0.0199	−0.0199
18.5377	19.5569	20.8397	0.0000	−0.0190
16.7057	17.6651	0.0000	−0.0100	−0.0100
15.3410	16.1475	0.0000	0.0000	0.9950
13.3130	14.7488	1.3508	0.9950	0.0000

Policy:

↓	↓	←	↑	→
↓	↓		↓	→
→	→	↑		↓
↑	↑		↓	↑
→	↑		→	↓
→	↑	←	→	G

5 SARSA (Ahmed)

5.1 Implementation of Components

- The walls were implemented so that if an agent hits a wall, it stays in the same spot.
- The door was implemented so that it stays closed and acts like a wall unless one of the agents opens it. This was done using two flags, one per agent, so if agent one opens it then that flag would be True, if agent one closes it by stepping off the button then it becomes False. If at least one of them is True then it's open. If an agent is on the state of the open door and another agent closes the door, the door closes after the agent steps off the door state. The agents don't know the door exists.
- In each episode, in the code agent 1 steps first and then agent 2, so practically they step together.
- The agents don't know the other exists. To them they just magically sometimes get a positive reward if the other reaches the goal. If the agents are in front of each other, then the one stepping treats the other one as a wall and stays in the same state.

5.2

i I picked $\alpha = 0.2$. After trial and error, I found this to converge within 20000 episodes, and 20000 was a good balance between converging and not taking too long to run.

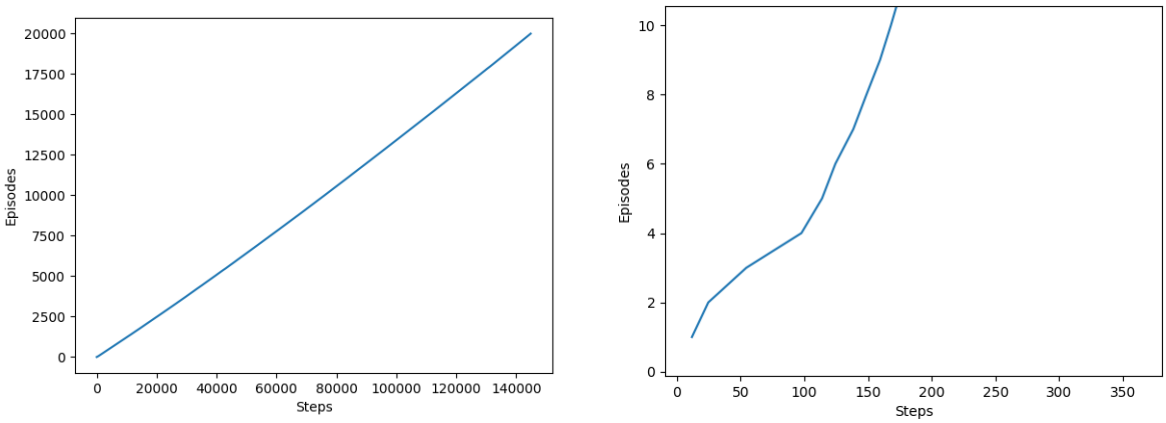
ii I initialized Q to zeros. I picked this because I thought it would be a good balance to be neutral rather than start optimistic or pessimistic, especially since most states have reward of 0.

iii I used ϵ-greedy. One of the reasons I chose this is because I already had the code from a previous homework. Another reason is because I wanted to be able to control a hyperparameter that falls between 0 and 1 rather than 0 and ∞.

iv I had to balance exploration with staying on the left side for both bots. The bots needed to go to the right side, especially bot1, so that way they know there is a goal and a reward there, but, especially for bot1, to learn that it's better to stay on the left side since it can open the door for bot2 and get a reward faster. They also needed to know to go to the goal if they ended up on the right side, because what was happening sometimes was that they both knew what to do on the left side (bot1 presses the button, bot2 goes through the door), but the policy was very wrong on the right side so they wouldn't go to the goal if they went around the top. This wasn't easy to do since our bots start at a fixed position and not randomly each episode. ϵ helped with exploration and γ helped with how myopic the agents are (which helped decide how much they should wait for the reward on the other side, thus how often they got to the other side). Using one ϵ and one γ often led to one bot having a correct right side and the other not. Thus, I experimented with two γs and two ϵs, one per bot. $\gamma 1 = 0.8$ and $\gamma 2 = 0.7$. 1 was higher than 2 because it needed to be a bit less myopic in order to go see the reward at the goal over the top and realize it's not the fastest way to get there, whereas 2 needed to stay more near the door and not go over the top as much. $\epsilon 1 = 0.4$ and $\epsilon 2 = 0.3$. 1 was higher than 2 to explore more for the same reasons as γ. Professor da Silva also recommended to have a lot of exploration in the beginning for the reason of having a good policy on the right side. 1 decayed by 2% every 500 episodes because staying at 0.4 the entire time was too much exploration and it wouldn't be able to get to the button as well on the left side. 2 stayed constant the entire time in order to explore into the right side, since sometimes it has to go to the right side over the top which is a long ways away, thus a higher exploration would help.

5.3

This is learning curve where, on the x axis, the total number of actions taken by the agents (since they step together, their steps are not added) are shown, from the beginning of the training process (i.e., the total number of steps taken across all episodes up to that moment in time). On the y axis, the number of episodes completed up to that point are shown. Since learning is successful, this graph has an increasing slope (Made clear in the zoomed version), indicating that as the agents take more and more actions (and thus executes more learning updates), fewer actions/timesteps are required to complete each episode.



(a) Regular

(b) Zoomed

Figure 6: Steps vs Episodes (Averaged over 20 runs)

5.4

The greedy policy with respect to the q-values learned by SARSA (Top is Bot1, bottom is Bot2):

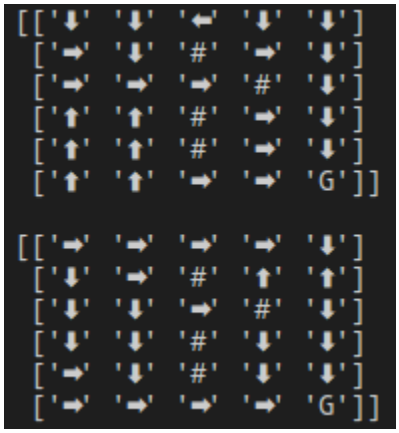


Figure 7

As you can see, Bot1 on the left side goes towards the button and if it's on the right, then it just goes to the goal since it's closer to that there. Bot2 goes towards the door at the bottom, but if it ends up on the right side by going around the top rather than through the door, then it goes down to the goal since it's closer to that there. When it's on the button, it hits the wall that way it stays on the button. Overall the SARSA algorithm worked well.

6 Q-Learning (Ahmed)

6.1 Implementation of Components

The same implementations as SARSA.

6.2

i I picked $\alpha = 0.075$. After trial and error, I found this to converge within 20000 episodes, and 20000 was a good balance between converging and not taking too long to run. Also, in HW5 my α went down from SARSA to Q-Learning, so I figured it might at least help a little bit here, at least as an inference to start experimenting with.

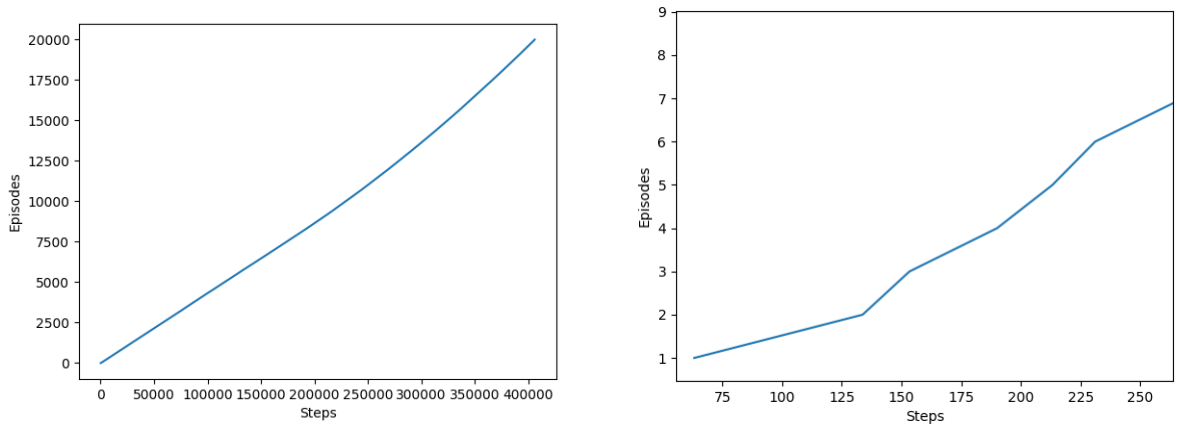
ii I initialized Q to zeros. I picked this because I thought it would be a good balance to be neutral rather than start optimistic or pessimistic, especially since most states have reward of 0.

iii I used ϵ-greedy. One of the reasons I chose this is because I already had the code from a previous homework. Another reason is because I wanted to be able to control a hyperparameter that falls between 0 and 1 rather than 0 and ∞.

iv I had to balance exploration with staying on the left side for both bots. The bots needed to go to the right side, especially bot1, so that way they know there is a goal and a reward there, but, especially for bot1, to learn that it's better to stay on the left side since it can open the door for bot2 and get a reward faster. They also needed to know to go to the goal if they ended up on the right side, because what was happening sometimes was that they both knew what to do on the left side (bot1 presses the button, bot2 goes through the door), but the policy was very wrong on the right side so they wouldn't go to the goal if they went around the top. This wasn't easy to do since our bots start at a fixed position and not randomly each episode. ϵ helped with exploration and γ helped with how myopic the agents are (which helped decide how much they should wait for the reward on the other side, thus how often they got to the other side). Using one ϵ and one γ often led to one bot having a correct right side and the other not. Thus, I experimented with two γs and two ϵs, one per bot. $\gamma 1 = 0.8$ and $\gamma 2 = 0.7$. 1 was higher than 2 because it needed to be a bit less myopic in order to go see the reward at the goal over the top and realize it's not the fastest way to get there, whereas 2 needed to stay more near the door and not go over the top as much. $\epsilon 1 = 0.5$ and $\epsilon 2 = 0.5$. Professor da Silva also recommended to have a lot of exploration in the beginning for the reason of having a good policy on the right side. After 8000 episodes, 1 was decayed by 2% every 500 episodes and 2 was decayed by 1% every 500 episodes, because staying at 0.5 the entire time was too much exploration and they wouldn't be able to get to the button and door as well on the left side. The reason 2 was decayed less than 1, is so that it stays higher and explore more on the right side, since sometimes it has to go to the right side over the top which is a long ways away, thus a higher exploration would help.

6.3

This is learning curve where, on the x axis, the total number of actions taken by the agents (since they step together, their steps are not added) are shown, from the beginning of the training process (i.e., the total number of steps taken across all episodes up to that moment in time). On the y axis, the number of episodes completed up to that point are shown. Since learning is successful, this graph has an increasing slope (Made clear in the zoomed version), indicating that as the agents take more and more actions (and thus executes more learning updates), fewer actions/timesteps are required to complete each episode.



(a) Regular

(b) Zoomed

Figure 8: Steps vs Episodes (Averaged over 20 runs)

6.4

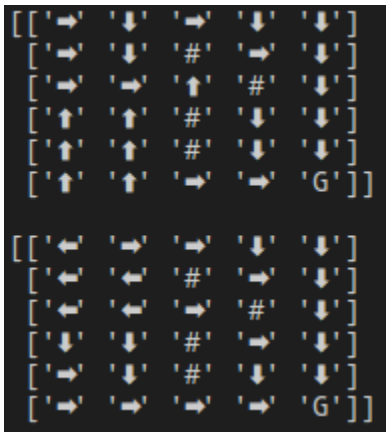


Figure 9

As you can see, Bot1 on the left side goes towards the button and if it's on the right, then it just goes to the goal since it's closer to that there. Bot2 goes towards the door at the bottom, but if it ends up on the right side by going around the top rather than through the door, then it goes down to the goal since it's closer to that there. When it's on the button, it hits the wall that way it stays on the button. Overall the Q-Learning algorithm worked well.

7 Conclusion

7.1 Summary

Overall the assignment went well, in terms of results it was great to be able to see how the algorithms were able to learn the better paths if they existed. In particular with Monte Carlo where the uptick in effort was visible for it to find the optimal paths. SARSA and Q-Learning did pretty well, and with enough exploration were able to find a policy very close to the optimal one, one that gets to the goal in the shortest amount of time. It was interesting to get two agents to collaborate with each other successfully without knowing the existence of each other using multiple algorithms. There was some additional specific insight we had regarding several parts of the environment.

7.2 Difficulties with shared rewards

One of the largest challenges we ran into was figuring out how to deal with shared rewards. The problem stemmed from the occasional possibility that a state Bot1 is at will be rewarded +10 because Bot2 coincidentally landed on the goal at that point. A problematic series of events that might cause this to happen would be Bot1 hits the button, Bot2 walks through the door, Bot1 leaves the button, Bot2 hits the goal and generates the reward, Bot1 gets rewarded for being part of the team, but the reward will be for rambling around elsewhere instead of getting the door open and holding it there. This was one of the things that made it difficult for the bots to learn the correct path as getting to the goal in the shortest manner requires several sequential steps to take place, and due to how much luck is involved (exploration value needs to be high to find the shortest path but also causes the random spikes in number of steps taken to complete later) the likelihood of it happening is very low. Part of them learning to collaborate requires that initial luck and for them to stumble into the correct path, realize it's significantly faster than the alternatives and then store that in the values of those states.

7.3 Difficulties with exploration

Getting the agents, especially Bot1 to go to the right side over the top and see the reward at the goal while at the same time having the policy choose to stay on the left side for the button and door was difficult. For SARSA and Q-Learning(Ahmed), a lot of tuning had to be done for γ and ϵ. Both agents needed to know what to do on both sides of the grid in case they ended up on the right side. Basically, they both needed to know where to go to get the reward in the shortest amount of steps, and getting that to work for the right side was challenging and required a lot of exploration since both agents are initialized on the left side.

8 Division of Labor

8.0.1 Ahmed

- Section 5 SARSA
- Section 6 Q-Learning
- Section 7.1 Summary
- Section 7.3 Difficulties with exploration

8.0.2 Greg

- The write up for sections 1, 2
- Section 3 Monte Carlo
- Section 4 Q-Learning
- Section 7.1 Summary
- Section 7.2 Difficulties with shared rewards