Multi-Agent Pathfinding using Reinforcement Learning - COSC 4368 Team URSA

Index:

Imports:

1. numpy: Used for array operations and numerical computations, which is crucial in managing the grid-based world and performing matrix manipulations.
2. random: Utilized for generating random numbers, which is essential for implementing stochastic behaviors in the agents' actions, such as choosing random actions when necessary and for exploration strategies in Q-Learning and SARSA.

Global Variables:

In our code we used several global variables which define the characteristics and functionalities of the environment. These include `pickupLocations` and `dropoffLocations`, both of which are dictionaries. The `pickupLocations` dictionary maps specific grid coordinates to the number of blocks available for agents to pick up, initially set with values at coordinates (1, 5), (2, 4), and (5, 2) each having 5 blocks. The `dropoffLocations` dictionary keeps track of how many blocks have been deposited by agents at certain coordinates, starting from zero and incrementing as blocks are delivered at coordinates (1, 1), (3, 1), and (4, 5). Another global variable, `world_size`, is an integer set to 5, representing the dimensions of a 5x5 grid world. This variable ensures that agent interactions occur within the specified environment parameters. These global variables provide structure and set the operational framework within which agents perform tasks such as moving, picking up, and dropping off blocks.

Classes/Modules:

Agent Class

The `Agent` class encapsulates the functionality of the individual agents operating within a grid-based environment. Each agent is uniquely identified by a `name` and has a `position` represented by a tuple indicating its coordinates within the grid. Agents have the capability to `move` within the grid, following the boundaries and avoiding spaces occupied by other agents. Movements result in score penalty, reflected in the `score` attribute, which is adjusted based on actions such as picking up or dropping off blocks. Each individual agent also keeps track of its `reward` in order to later pick the next optimal action. Agents possess a `have_block` flag indicating whether they are carrying a block. The class includes methods for moving (`move`), picking up blocks (`pickup`), and dropping them off (`dropoff`). During the `move` operation, the agent checks if the move is valid, updates its position, and performs pickup or dropoff actions if applicable, modifying the `score` and `have_block` status and keeping track of the `reward` based on the action taken.

QLearningAgent Class

The `QLearningAgent` class implements the Q-learning algorithm, which is a model-free reinforcement learning technique used to inform the agents about optimal actions based on state assessments and rewards. Key parameters include the `alpha` (learning rate), which determines the impact of new information on the agent's knowledge; `gamma` (discount factor), which balances the importance of immediate versus future rewards; and `epsilon`, governing the likelihood of taking exploratory actions to facilitate a comprehensive understanding of the environment. The `q_table` stores the Q-values for each state-action pair, initialized to zero for unknown pairs. Methods within this class help retrieve and update these Q-values (`get_q_value`, `update_q_value`), decide actions based on the current policy (`get_action`, using an epsilon-greedy strategy), and perform purely random (`PRandom`) or optimal actions (`PGreedy`). `PExploit` offers a blend of exploration and exploitation, primarily favoring the best-known actions while allowing for random choices to ensure diversity in experienced states.

SARSA Class

Similar in structure to `QLearningAgent`, the `SARSA` class implements the SARSA learning algorithm, another reinforcement learning method that, unlike Q-learning, updates its policy based on the action actually taken next, rather than the best possible future action. This class maintains the same structure for attributes and methods as `QLearningAgent`, with the primary difference being in the `update_q_value` method. In SARSA, the Q-value update considers the actual next action taken, which integrates the policy into the learning process more tightly, making it an on-policy method. This approach can lead to different learning dynamics and agent behaviors, especially in terms of how quickly and effectively they converge to an optimal policy.

These classes are fundamental to the operation of the simulation, directing how agents learn from and interact with their environment. They are crafted to ensure that agents can not only perform their tasks within the grid but also adapt their strategies over time through learning mechanisms defined by the Q-learning and SARSA models. This setup allows the exploration of various strategies and their effectiveness in a controlled yet dynamic setting.

Interaction Between Modules:

These three modules or classes interact with each other since the agent class produces the environment that the agents are in and the q-learning and SARSA classes promote the learning process of the agents. The agents are navigating around the world while avoiding other agents. Their goal is to pick up and drop off blocks at the designated coordinates. Each move will affect the agents' score/reward. The position of the agent gives a state to the reinforcement learning algorithms where it is used to calculate q-values for the possible actions the agents can make. These calculated values are stored in the q-table. The agents use these values to make a decision on their action. Then this cycle repeats as the agent moves to a new position and a new state is sent to the learning algorithm.

State Space and Q-Table Design:

State Space:

We first initialize our agents with its name/color, their positions, rewards, as well as if they are currently holding a box or not. To put it in variables, it looks like Agent("name/color", (agent's coordinates), have_block, score). Have block is a boolean or binary variable which will either be 0 for when the agent does not have a block and is looking for a pickup cell or 1 for when the agent has a block and is looking for a dropoff cell. The score is an integer that will decrease or increase depending on the move it makes. If the agent is picking up or dropping off a block, the reward will be +13, however, if the agent does any other move, it will decrease by 1.

Q-Table:

Each row in our table represents a state while each column represents the possible actions the agent can take. We first show the state, which is the coordinates of the agent. Then we show the possible actions or directions that the agent can take or move in, as well as its corresponding q-value, which estimates the reward the agent is expected to receive if taking that action. The design of the q-table helps look up the q-values efficiently and enforces learning an optimal policy.

Implementation of All Constraints:

In the multi-agent simulation, the implementation of constraints is crucial to ensure that agents operate within the defined rules of the environment, handle interactions correctly, and adhere to the physical limitations of the grid. These constraints influence how agents decide their movements, interact with pickup and dropoff locations, and avoid conflicts with other agents.

<u>Movement Constraints</u>
One of the primary constraints in the system is the regulation of agent movement within the grid boundaries and avoiding collisions with other agents:

- Boundary Constraints: Agents are confined to the 5x5 grid, and any action that would move an agent outside these boundaries is considered invalid. This is implemented within the `move` method of the `Agent` class. Before an agent moves, the method checks if the new position (after applying the action vector) remains within the grid limits.

- Collision Avoidance: Agents must also avoid moving into grid cells occupied by other agents. During each move attempt, the `move` method checks the proposed new position against the positions of all other agents in the simulation. If another agent occupies the target position, the move is blocked, and the agent must choose a different action.

```
def move(self, action, agents):
    new_x, new_y = self.position[0] + action[0], self.position[1] + action[1]
    if 0 <= new_x < world_size and 0 <= new_y < world_size:
        for agent in agents:
            if agent.position == (new_x, new_y):
                return False
        self.position = (new_x, new_y)
        return True
    return False
```

Pickup and Dropoff Logic

Agents interact with the environment by picking up and dropping off blocks at specified locations. This behavior is governed by several rules:

- Pickup Logic: An agent can pick up a block if and only if it is not already carrying a block, it is positioned at a designated pickup location, and there are blocks available at that location. This logic is encapsulated in the `pickup` method of the `Agent` class, which checks these conditions before allowing a block to be picked up.

- Dropoff Logic: Conversely, an agent can drop off a block only if it is carrying one and it is at a designated dropoff location. The `dropoff` method manages this process, ensuring that the agent meets all conditions before a block is deposited.

```
def pickup(self):
    if self.can_pickup(self.position):
        self.have_block = True
        pickupLocations[self.position] -= 1

def dropoff(self):
    if self.can_dropoff(self.position):
        self.have_block = False
        dropoffLocations[self.position] += 1
```

Learning Constraints

For the learning agents (`QLearningAgent` and `SARSA`), constraints are implemented to ensure that learning follows the principles of reinforcement learning:

- Q-Value Initialization and Update: Initially, all Q-values are set to zero, representing no prior knowledge. They are updated based on the rewards received and the estimated future rewards, with distinct mechanisms for Q-learning and SARSA reflecting their respective theories.

- Q-Table Utilization: For both the `QLearningAgent` and `SARSA` agents, a single Q-table is used across all state-action pairs, rather than using separate Q-tables for different aspects or dimensions of the environment. This approach simplifies the model by maintaining a unified representation of the learned values.

- Exploration vs. Exploitation: The epsilon-greedy strategy is used to balance exploration (trying new actions) and exploitation (using known good actions), with the `epsilon` parameter controlling this balance. Random actions are selected based on `epsilon`, while the best-known actions are chosen otherwise.

These constraints are essential for maintaining the integrity of the simulation, ensuring that agent interactions are realistic and adhere to specified rules, and that learning progresses in a controlled and theoretically sound manner. They collectively ensure that the simulation provides meaningful insights into the behavior and learning of agents within a complex, rule-based environment.

Experiment Result Charts:

Experiment 1a	Red Agent Reward	Blue Agent Reward	Black Agent Reward	Did it complete?	Number of steps till completion if it completed
Run 1	-342	-408	-352	Yes	681
Run 2	-226	-122	-92	Yes	370

Experiment 1b	Red Agent Reward	Blue Agent Reward	Black Agent Reward	Did it complete?	Number of steps till completion if it completed
Run 1	-136	-96	-2	Yes	295
Run 2	-555	-504	-391	Yes	1707

Experiment 1c	Red Agent Reward	Blue Agent Reward	Black Agent Reward	Did it complete?	Number of steps till completion if it completed
Run 1	-712	-672	-609	Yes	1632

Run 2	-466	-247	-292	Yes	743

Experiment 2	Red Agent Reward	Blue Agent Reward	Black Agent Reward	Did it complete?	Number of steps till completion if it completed
Run 1	-317	–320	-266	Yes	693
Run 2	–392	-448	-440	Yes	973

Experiment 3a Alpha = .15	Red Agent Reward	Blue Agent Reward	Black Agent Reward	Did it complete?	Number of steps till completion if it completed
Run 1	-239	-116	-195	Yes	452
Run 2	-240	-164	-245	Yes	495

Experiment 3b Alpha = .45	Red Agent Reward	Blue Agent Reward	Black Agent Reward	Did it complete?	Number of steps till completion if it completed
Run 1	-415	-286	-357	Yes	787
Run 2	-161	-147	-208	Yes	422

Experiment 4	Red Agent Reward	Blue Agent Reward	Black Agent Reward	Did it complete?	Number of steps till completion if it completed
Run 1	-336	-371	-389	Yes	1053
Run 2	-131	-80	-76	Yes	315

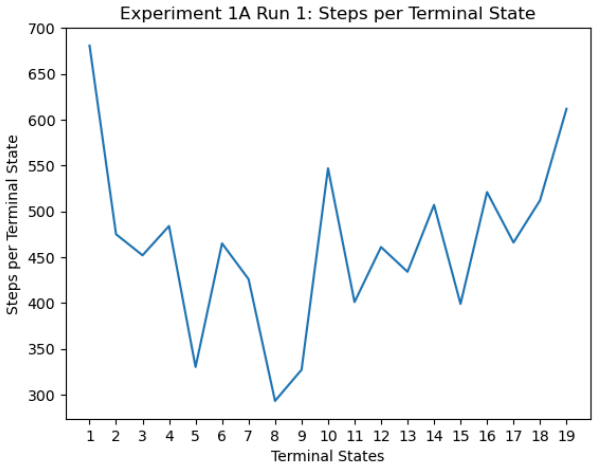
Analysis of Experiments:

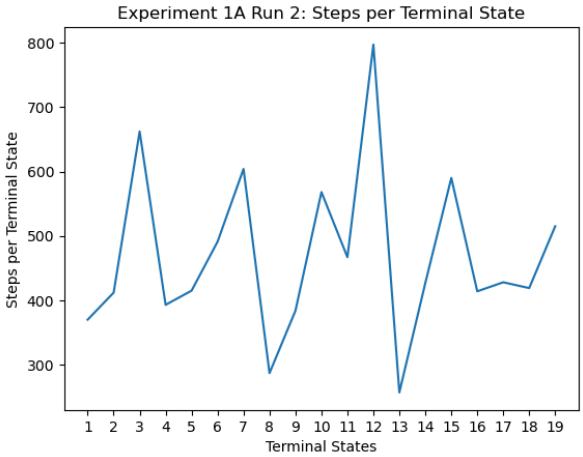
Experiment 1a:

In this experiment, we ran all the steps using the traditional q-learning algorithm and only random.

For Experiment 1A, our 2 runs both reached 19 terminal states after running PRANDOM 500 times, then running PRANDOM 8500 more times. Run 1 had the steps per terminal state drop dramatically within the first 3 terminal states, dropping from 681 to 452, before fluctuating within the 325-525s range for most future terminal states, with one noticeable drop to 293 steps at terminal state 8 and a gradual increase to 612 steps near the end at terminal state 19. Meanwhile, Run 2 started off with a

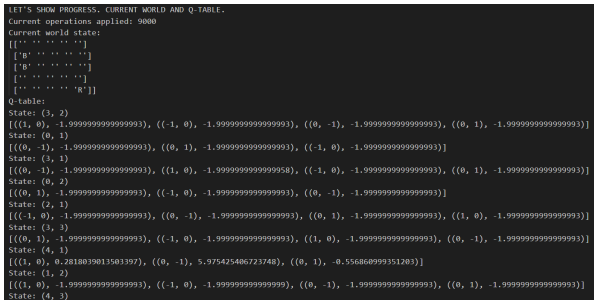
lower steps per terminal state count at 370, fluctuating within the 285-665s range, with one noticeable increase to 797 steps followed by a significant drop to 257 steps and a return to a more normal 427 steps at terminal states 12-14. While all reward values per agent were negative, their magnitudes were approximately scaled to the number of steps per terminal state; terminal states with significantly larger steps per state had noticeably larger magnitude on their reward values, such as how a state with 681 steps in Run 1 had rewards of [-342, -408, -352] compared to how another state with 293 steps in Run 1 had rewards of [-35, -56, -90]. The more random spread and fluctuations of steps per terminal state and rewards is likely a result of using PRANDOM, as the random actions would eventually lead to terminal states with varying amounts of time between each state.

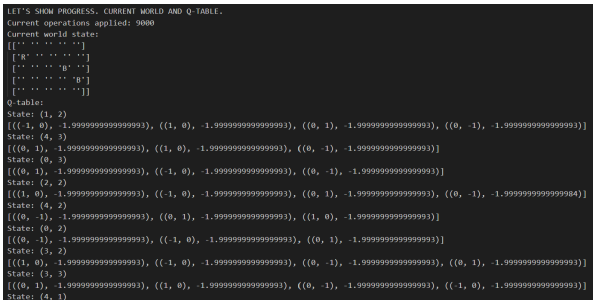




RUN 1:
Steps per Terminal State: [681, 475, 452, 484, 330, 465, 426, 293, 327, 547, 401, 461, 434, 507, 399, 521, 466, 512, 612]
Rewards per Terminal State: [[-342, -408, -352], [-271, -118, -249], [-223, -128, -219], [-230, -231, -172], [-84, -73, -174], [-227, -170, -175], [-166, -178, -162], [-35, -56, -90], [-124, -124, -40], [-220, -274, -285], [-113, -146, -201], [-241, -180, -166], [-222, -170, -182], [-180, -261, -271], [-238, -138, -94], [-320, -238, -199], [-243, -154, -207], [-220, -212, -220], [-304, -262, -365]]

RUN 2:
Steps per Terminal State: [370, 412, 662, 393, 415, 491, 604, 287, 384, 568, 467, 797, 257, 427, 590, 414, 428, 419, 515]
Rewards per Terminal State: [[-226, -122, -92], [-150, -140, -183], [-284, -369, -342], [-186, -120, -148], [-154, -172, -168], [-165, -291, -208], [-236, -360, -368], [-37, -83, -101], [-164, -154, -144], [-282, -235, -257], [-228, -148, -243], [-410, -499, 445], [-68, -56, -18], [-124, -132, -216], [-322, -271, -282], [-136, -126, -197], [-96, -208, -200], [-102, -229, -184], [-132, -235, -308]]



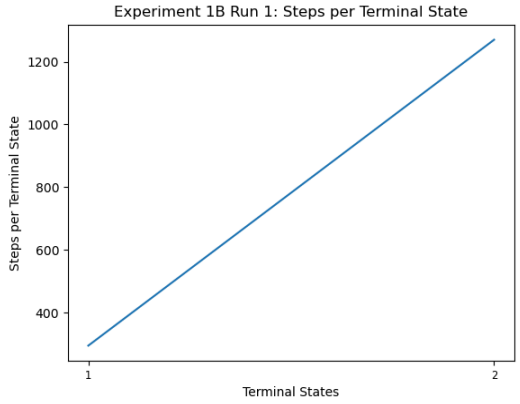


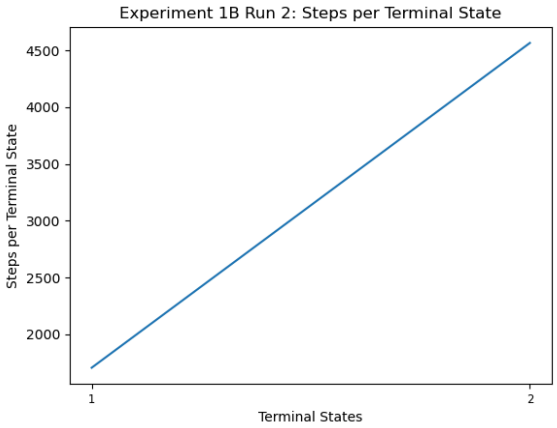
Experiment 1b:

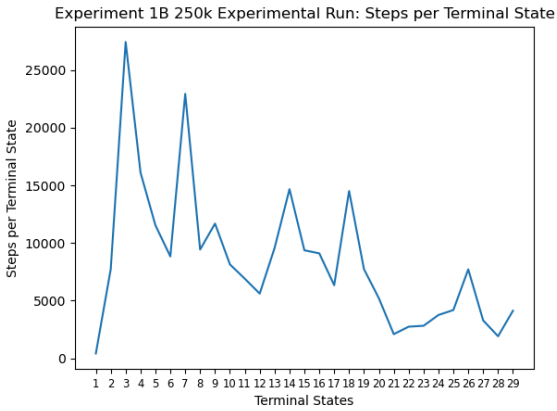
In this experiment, we ran 500 steps using the traditional q-learning algorithm with the policy PRANDOM, then ran 8500 more steps using PGREEDY.

Both of our runs only reached terminal states twice. Run 1 reached its first terminal state after 295 steps, with rewards of [-136, -96, -2], and its second after a further 1270 steps, with rewards of [-307, -236, -401]. Run 2 reached its terminal states much later, with its first at 1707 steps with rewards of [-555, -504, -391], and its second after a further 4565 steps with rewards of [-1583, -1108, -1422]. Both runs reached their first terminal state relatively early on, with Run 1 reaching one before its runs of PRANDOM finished, and only reached another terminal state comparatively later, with Run 1 reaching its last terminal state less than 2/9ths through its steps and Run 2 notably taking more than half of the remaining steps to reach its last terminal state. The agent rewards for each step do reflect the number of steps taken, with higher steps per terminal state reflecting in much higher magnitudes of negative reward. It is likely that the reason our PGREEDY policy ran so poorly was because of inefficiencies in its ability to learn the data. As a result, our 1B results are significantly worse than our 1A ones, something that's quite surprising as normally we'd expect a greedy policy to run significantly more efficiently than a purely random policy.

In order to investigate whether our PGREEDY policy was learning correctly, we decided to run Experiment 1B a third time with PGREEDY running for 249500 steps instead of 8500 steps. The results of this extra run are shown below in the graph labeled "Experiment 1B 250k Experimental Run: Steps per Terminal State", which displays how, over 29 terminal states, the steps per terminal state gradually begins to decrease, peaking at a high of 27435 steps per state at terminal state 3 but eventually reducing to a low of 1918 steps per state at terminal state 28. We believe this trend does indicate that our PGREEDY algorithm is slowly learning, albeit at a rate that's unreasonably slow for the constraints of the problem we were given.



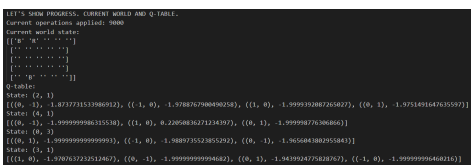


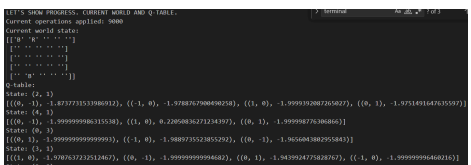


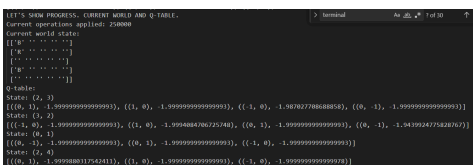
RUN 1:
Steps per Terminal State: [295, 1270]
Rewards per Terminal State: [[-136, -96, -2], [-307, -236, -401]]

RUN 2:
Steps per Terminal State: [1707, 4565]
Rewards per Terminal State: [[-555, -504, -391], [-1583, -1108, -1422]]

EXPERIMENTAL RUN:
Steps per Terminal State: [426, 7743, 27435, 16102, 11537, 8840, 22936, 9439, 11690, 8152, 6908, 5617, 9580, 14672, 9372, 9108, 6346, 14514, 7733, 5212, 2102, 2827, 3763, 4191, 7718, 3296, 1918, 4132]
Rewards per Terminal State: [[-134, -198, -179], [-777, -1085, -1022], [-2693, -2378, -2716], [-1297, -1417, -1430], [-1778, -1247, -1302], [-1299, -1854, -1208], [-2030, -1926, -1959], [-1285, -1158, -1094], [-1600, -1522, -1875], [-1178, -1377, -1555], [-1100, -1098, -1046], [-883, -682, -861], [-1715, -1458, -1500], [-1072, -1133, -1214], [-1242, -1256, -1168], [-1520, -1424, -1396], [-778, -594, -736], [-1488, -1392, -1616], [-1248, -1511, -1546], [-899, -775, -1212], [-146, -98, -224], [-308, -422, -342], [-426, -568, -403], [-516, -606, -604], [-1022, -912, -910], [-403, -388, -488], [-126, -230, -270], [-392, -600, -584]]



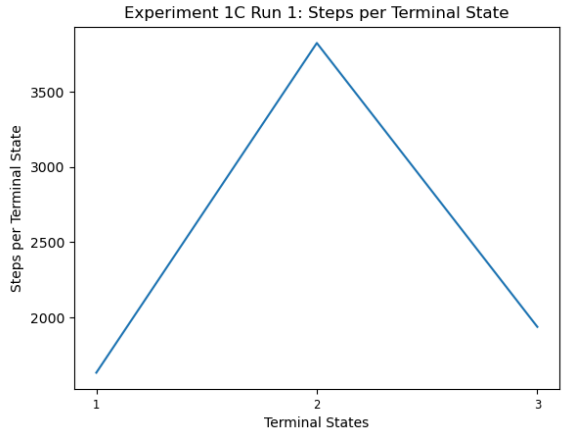


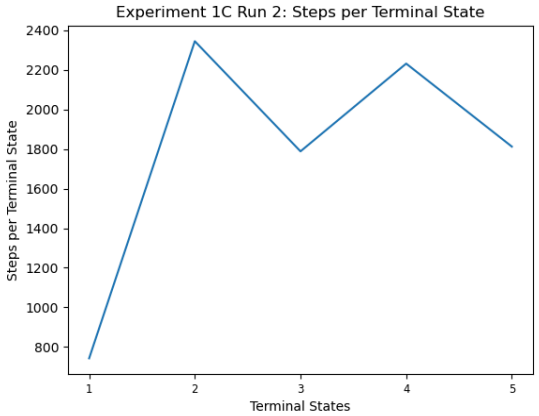


Experiment 1c:

In this experiment, we first ran the traditional q-learning algorithm with policy PRANDOM for 500 steps, then ran the policy PEXPLOIT for 8500 more steps.

Our Run 1 reached a terminal state 3 times, while our Run 2 reached a terminal state 5 times. Both runs started out with a lower steps per terminal state, with Run 1 initially having 1632 steps per terminal state and Run 2 having 743 steps per terminal state, before jumping to a much larger steps value with terminal state 2, with Run 1 having 3826 steps per terminal state and Run 2 having 2344 steps per terminal state. Afterwards, Run 1 dropped significantly lower for its third and final terminal state compared to Run 2's; Run 1 dropped to 1937 steps per state compared to Run 2's 1788, with a drop of 1889 versus 556. Finally, Run 2 jumps up to 2231 steps per state before ending with a drop to 1812 steps per state. Both runs follow the same initial pattern as the 2 Experiment 1B runs, with severe initial steps per state increases within the first 2 terminal states, but both also immediately diverged with a noticeable drop. After observing the trends from the experimental 1B run, we assume that the divergence was likely caused by the 20% chance for PEXPLOIT to use PRANDOM, making up for our weaker PGREEDY policy, allowing 1C to produce much better results than 1B. Rewards for the agents still mostly reflect the magnitude of their associated steps per terminal state, but notably all the rewards for Run 1 terminal state 3 are less than that for Run 1 terminal state 1 despite the former having a larger steps per terminal state count than the latter.

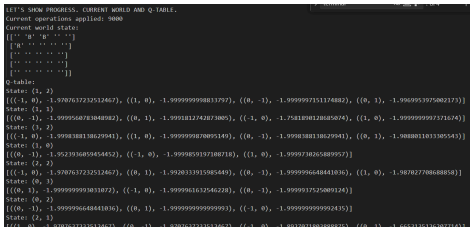


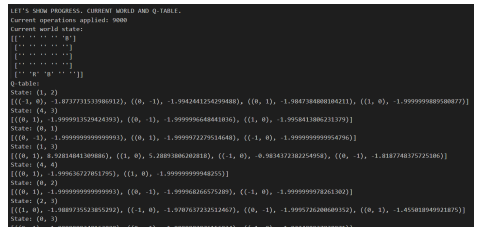


RUN 1:
Steps per Terminal State: [1632, 3826, 1937]
Rewards per Terminal State: [[-712, -672, -609], [-1142, -1211, -1298], [-252, -623, -497]]

RUN 2:

Steps per Terminal State: [743, 2344, 1788, 2231, 1812]
Rewards per Terminal State: [[-466, -247, -292], [-510, -485, -864], [-693, -434, -374], [-570, -814, -617], [-720, -316, -446]]



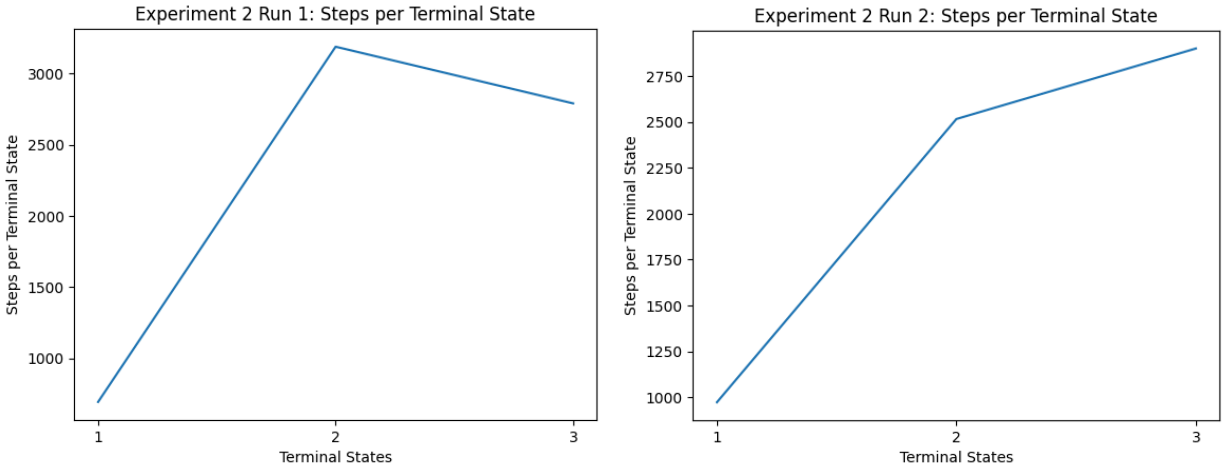


Experiment 2:

In this experiment, we first ran the SARSA q-learning algorithm with policy PRANDOM for 500 steps, then ran the policy PEXPLOIT for 8500 more steps for a total of 9000 steps.

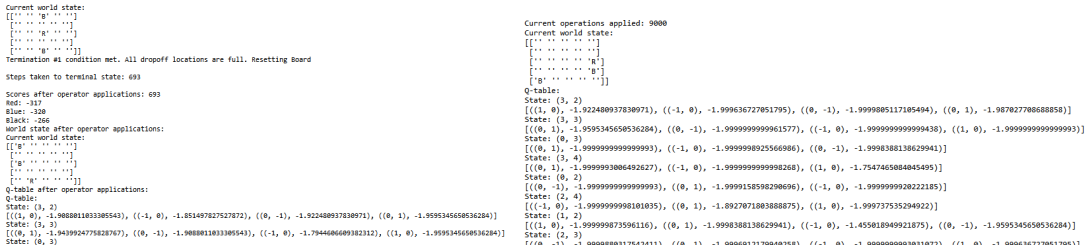
For the first run, we see the number of steps increase then decrease. For the second run, we just see it increasing. These fluctuations can be due to the randomness that comes with the exploration, which is a direct cause to the exploitation during PEXPLOIT. Using SARSA yielded negative rewards results like using a traditional q-learning, however, on average the SARSA rewards were greater than the traditional method. In both of the runs for this experiment, the terminal state was reached a total of three times, showing the consistency. The range of steps to reach a terminal state was 693-3188, as well as the different reward distributions, showing the difference in the efficiency of the agents' learning processes and their exploration strategies. Comparing these results to 1C, which used the traditional q-learning method, for both experiments, the number of steps to reach a terminal state decreased when running the experiment for a second time. This shows that the agents' efficiencies are increasing. The SARSA method had lower steps on average. Q-learning's rewards ranged from -1211 to -247, while SARSA's rewards ranged from -973 to -317. SARSA generally had better scoring rewards and better performance. Comparing these two experiments, SARSA is the method that performed better when looking at the number of steps to reach a terminal state, as well as the rewards of each agent.

Looking at just three terminal states didn't show us much information on if our algorithm was actually learning, so we increased the number of operations from 9000 to 50000. By doing this, we were able to encounter triple the terminal states. It also shows that our agents are learning with the decreasing trend after the second terminal state. This shows the agents are learning and improving their coordination with time, becoming more efficient in reaching terminal states.

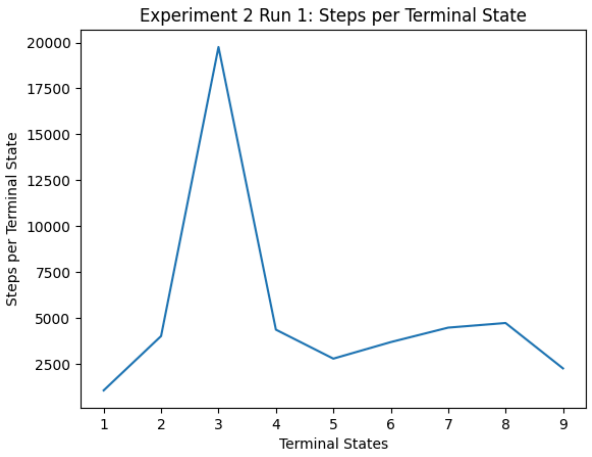


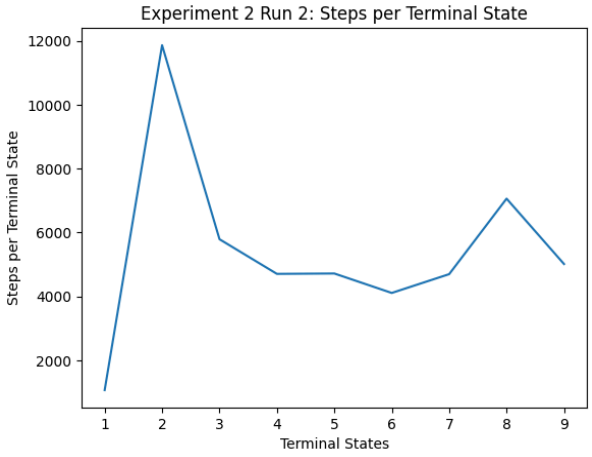
Run 1:
Terminal States reached: 3
Steps per Terminal State: [693, 3188, 2790]
Rewards per Terminal State: [(-317, -320, -266), (-891, -973, -674), (-352, -398, -513)]

Run 2:
Terminal States reached: 3
Steps per Terminal State: [973, 2516, 2900]
Rewards per Terminal State: [(-392, -448, -440), (-543, -438, -462), (-748, -835, -814)]



Experimental: 50000 steps





Run 1:
Terminal States reached: 9
Steps per Terminal State: [1057, 4014, 19754, 4363, 2785, 3686, 4474, 4726, 2254]
Rewards per Terminal State: [(-496, -399, -405), (-790, -700, -575), (-2969, -3045, -3020), (-563, -581, -556), (-369, -430, -269), (-557, -578, -509), (-804, -837, -793), (-751, -703, -846), (-248, -385, -323)]

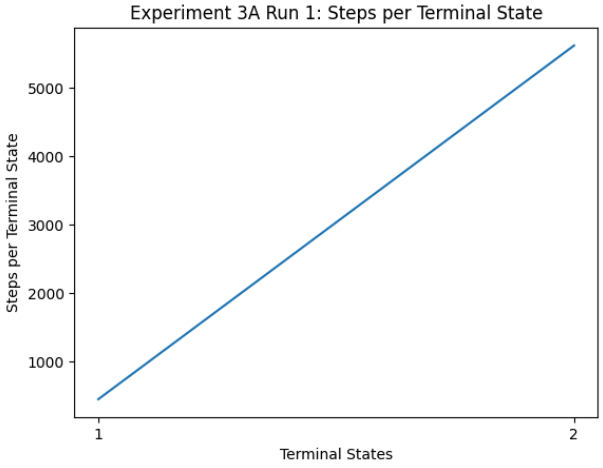
Run 2:
Terminal States reached: 9
Steps per Terminal State: [1068, 11859, 5788, 4704, 4716, 4107, 4695, 7058, 5013]
Rewards per Terminal State: [(-378, -322, -290), (-2424, -2672, -2745), (-2150, -1924, -2050), (-1186, -1300, -1226), (-1334, -1399, -1527), (-1150, -1033, -1069), (-1436, -1563, -1021), (-1926, -1946, -2013), (-1522, -1205, -1246)]

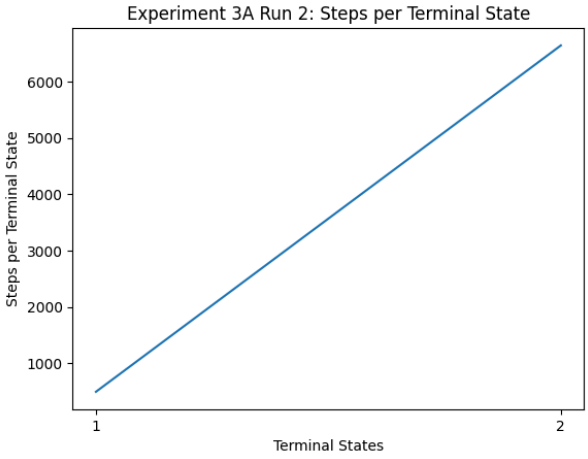
Experiment 3:

In this experiment, we first ran the SARSA q-learning algorithm with policy PRANDOM for 500 steps, then ran the policy PEXPLOIT for 8500 more steps.

Here we will be looking at the effects of changing the learning rate. Looking at 3A that had a learning rate of .15, for both runs, two terminal states were achieved. With 3B, which had a learning rate of .45, both runs experienced three terminal states. 2, which had a learning rate of .3, also had three terminal states. With a lower learning rate, the less terminal states were reached with the same number of operations. It is hard to compare the rewards of the agents between the runs and different experiments since they vary greatly. However, we can make some conclusions by looking at the number of steps to reach terminal states. Looking at the numbers, Experiment 3B had lower steps than experiment 2, while experiment 2 had a lower number of steps than experiment 3A. The learning rate that follows those experiments goes from greatest to least. This shows that with a higher learning rate, a terminal state will be reached quicker on average. With this knowledge, it might be possible to achieve a lower number of steps per terminal state if we make the learning rate greater than .45.

Experiment 3a (alpha = .15) :

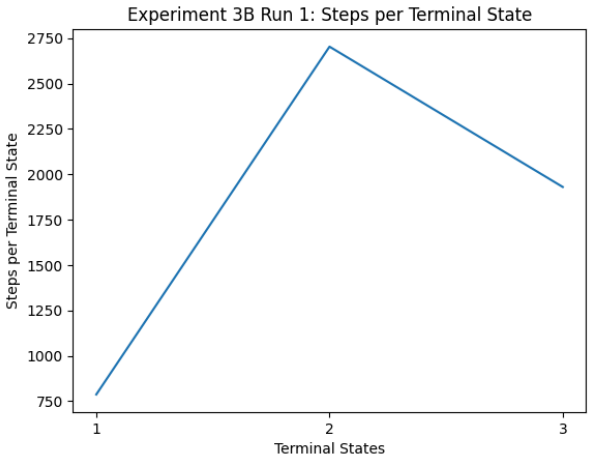


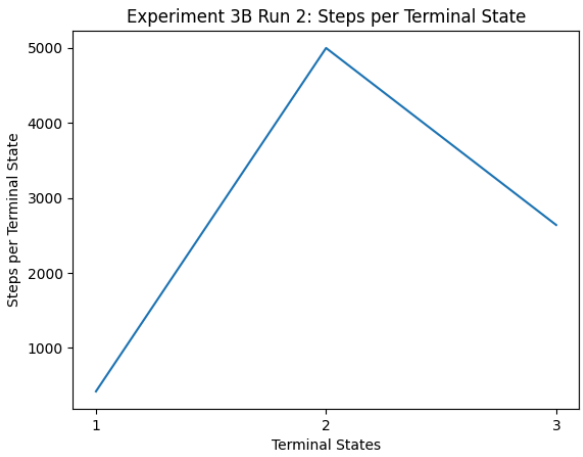


Run 1:
Terminal States reached: 2
Steps per Terminal State: [452, 5616]
Rewards per Terminal State: [(-239, -116, -195), (-922, -940, -821)]

Run 2:
Terminal States reached: 2
Steps per Terminal State: [495, 6642]
Rewards per Terminal State: [(-240, -164, -254), (-2034, -1922, -2206)]

Experiment 3b (alpha = .45):

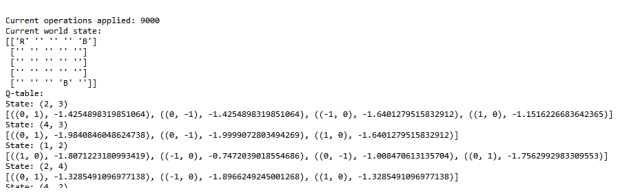


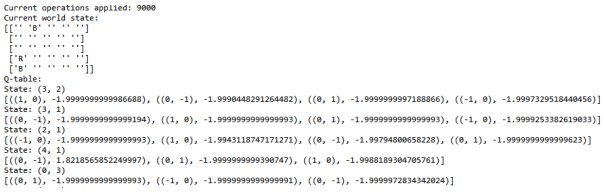


Run 1:
Terminal States reached: 3
Steps per Terminal State: [787, 2703, 1930]

Rewards per Terminal State: [(-415, -286, -357), (-807, -892, -820), (-456, -536, -577)]

Run 2:
Terminal States reached: 3
Steps per Terminal State: [422, 4995, 2638]
Rewards per Terminal State: [(-161, -147, -208), (-834, -860, -839), (-342, -280, -349)]

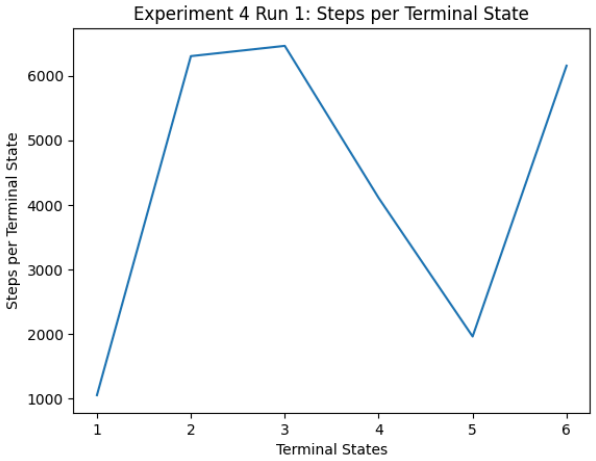


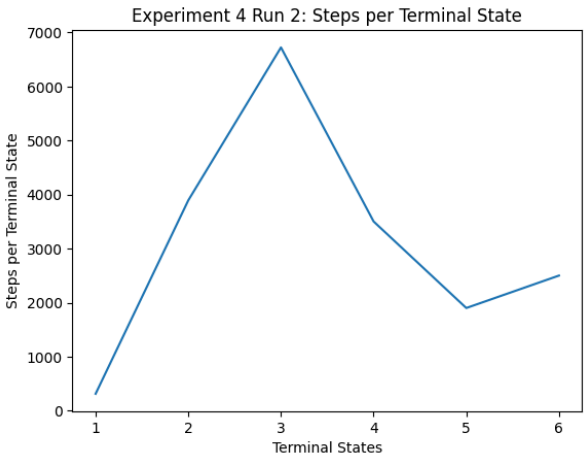


Experiment 4:

In this experiment, we first ran the SARSA q-learning algorithm with policy PRANDOM for 500 steps, then ran the policy PEXPLOIT until a total of 6 terminal states was reached.

Comparing the two runs, on average, run 2 had a lower number of steps to reach a terminal state than run 1 had. Both runs also seem to follow the same trend throughout the number of terminal states. The trend of these runs was that it would increase in steps until three terminal states were reached. Then for the fourth and fifth it would decrease but then go back up again for the final state of termination. When we look at these results next to experiment 2, it is hard to compare since only three terminal states were reached then. If we look at the experimental run we did for experiment 2 where we increased the number of operations, the learning rate there is a lot more efficient. With this experiment, since we are completely changing the pickup locations to something new, it takes the agents some time to adjust. If we were to extend the maximum number of terminal states reached, the graph will probably show the number of steps decreasing with more terminal states being reached. This shows that it takes a bit for our agents to get readjusted to forgetting about the old paths and to learn some new ones.

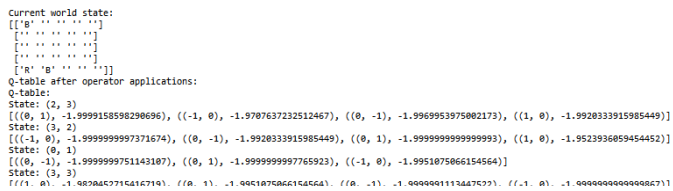




Run 1:
Terminal States reached: 6

Steps per Terminal State: [1053, 6301, 6460, 4105, 1961, 6152]
Rewards per Terminal State: [(-336, -371, -386), (-1309, -1328, -1266), (-1976, -1332, -1337), (-687, -712, -649), (-582, -446, -561), (-1322, -1100, -1203)]

Run 2:
Terminal States reached: 6
Steps per Terminal State: [315, 3894, 6724, 3503, 1902, 2502]
Rewards per Terminal State: [(-131, -80, -76), (-1173, -828, -1024), (-1715, -1646, -2070), (-1230, -704, -965), (-313, -170, -456), (-500, -610, -585)]



Pseudocode of Experiments:

Experiment 1a:

```
def experiment_1A():
    step = 0
    step_limit = 9000
    while step < 500:
        for agent in agents:
            actions = [N,S,E,W]
            action = PRANDOM(action)
             if agent.move(action, agents):
                update_q_value(agent.position, action, agent.movement, agent.position, action)
        if all(value == 5 for value in dropoffLocations.values()):
            reset_board()
    while step < step_limit:
        for agent in agents:
            actions = [N,S,E,W]
            action = PRANDOM(action)
             if agent.move(action, agents):
                update_q_value(agent.position, action, agent.movement, agent.position, action)
        if all(value == 5 for value in dropoffLocations.values()):
            reset_board()
```

Experiment 1b:

```
def experiment_1B():
```

```
step = 0
step_limit = 9000
while step < 500:
    for agent in agents:
        actions = [N,S,E,W]
        action = PRANDOM(action)
         if agent.move(action, agents):
            update_q_value(agent.position, action, agent.movement, agent.position, action)
    if all(value == 5 for value in dropoffLocations.values()):
        reset_board()
while step < step_limit:
    for agent in agents:
        actions = [N,S,E,W]
        action = PGREEDY(action)
         if agent.move(action, agents):
            update_q_value(agent.position, action, agent.movement, agent.position, action)
    if all(value == 5 for value in dropoffLocations.values()):
        reset_board()
```

Experiment 1c:

```
def experiment_1C():
    step = 0
    step_limit = 9000
    while step < 500:
        for agent in agents:
            actions = [N,S,E,W]
            action = PRANDOM(action)
             if agent.move(action, agents):
                update_q_value(agent.position, action, agent.movement, agent.position, action)
        if all(value == 5 for value in dropoffLocations.values()):
            reset_board()
    while step < step_limit:
        for agent in agents:
            actions = [N,S,E,W]
            action = PEXPLOIT(action)
             if agent.move(action, agents):
                update_q_value(agent.position, action, agent.movement, agent.position, action)
        if all(value == 5 for value in dropoffLocations.values()):
            reset_board()
```

Experiment 2:

```
def experiment_2():
```

```
agent = SARSA(alpha,gamma)
step = 0
step_limit = 9000
while step < 500:
   for agent in agents:
      actions = [N,S,E,W]
      action = PRANDOM(action)
       if agent.move(action, agents):
          update_q_value(agent.position, action, agent.movement, agent.position, action)
   if all(value == 5 for value in dropoffLocations.values()):
      reset_board()
while step < step_limit:
   for agent in agents:
      actions = [N,S,E,W]
      action = PEXPLOIT(action)
       if agent.move(action, agents):
          update_q_value(agent.position, action, agent.movement, agent.position, action)
   if all(value == 5 for value in dropoffLocations.values()):
      reset_board()
```

Experiment 3a:

```
def experiment_3A():
   agent = SARSA(alpha = 0.15 ,gamma)
   step = 0
   step_limit = 9000
   while step < 500:
      for agent in agents:
         actions = [N,S,E,W]
         action = PRANDOM(action)
          if agent.move(action, agents):
             update_q_value(agent.position, action, agent.movement, agent.position, action)
      if all(value == 5 for value in dropoffLocations.values()):
         reset_board()
   while step < step_limit:
      for agent in agents:
         actions = [N,S,E,W]
         action = PEXPLOIT(action)
          if agent.move(action, agents):
             update_q_value(agent.position, action, agent.movement, agent.position, action)
      if all(value == 5 for value in dropoffLocations.values()):
         reset_board()
```

Experiment 3b:

```
def experiment_3B():
    agent = SARSA(alpha = 0.45,gamma)
    step = 0
    step_limit = 9000
    while step < 500:
        for agent in agents:
            actions = [N,S,E,W]
            action = PRANDOM(action)
             if agent.move(action, agents):
                update_q_value(agent.position, action, agent.movement, agent.position, action)
        if all(value == 5 for value in dropoffLocations.values()):
            reset_board()
    while step < step_limit:
        for agent in agents:
            actions = [N,S,E,W]
            action = PEXPLOIT(action)
             if agent.move(action, agents):
                update_q_value(agent.position, action, agent.movement, agent.position, action)
        if all(value == 5 for value in dropoffLocations.values()):
            reset_board()
```

Experiment 4:

```
def experiment_4():
    agent = SARSA(alpha,gamma)
    step = 0
    terminal states reached = 0
while terminal states reached < 6
    if terminal states reached = 3
        update and change pickup locations
    while step < 500:
        for agent in agents:
            actions = [N,S,E,W]
            action = PRANDOM(action)
             if agent.move(action, agents):
                update_q_value(agent.position, action, agent.movement, agent.position, action)
        if all(value == 5 for value in dropoffLocations.values()):
            reset_board()

    for agent in agents:
         actions = [N,S,E,W]
         action = PEXPLOIT(action)
         if agent.move(action, agents):
```

```
            update_q_value(agent.position, action, agent.movement, agent.position, action)
        if all(value == 5 for value in dropoffLocations.values()):
            reset_board()
```

Future Changes:

As our program is far from perfect, there are a lot of things that we learned from this project and would like to add in order to better our Reinforcement Learning Agents. Mainly, we would rewrite the code in order to accept and run the PGreedy and PExploit constraints for action selection better. We believe that this was due to creating the base of the code using the MAPF algorithm rather than jumping straight into implementing Q-Learning. If we started with Q-Learning Agent first, we may have been able to cut out a middle man and simply adjust the algorithm to match the project's qualifications. However, we were hyper focused on the 3 agent path finding. Originally, this code ran with an A* search algorithm and we had adjusted it to work with Q-Learning SARSA. However, creating this program was a big learning opportunity and if we were to do it again, we would instead use Q-learning as a base and modify it to work with 3 agents, rather than the opposite like how we did here.

Resources:

The base of our code was a multi-agent pathfinding algorithm with added Q-learning and SARSA aspects. As it is our first time coding an RL algorithm like this, it took a lot of debugging and language help which resulted in us using chatGPT to see if we were writing our code correctly. Here are the resources that helped us in completion of the program.

MAPF Algorithm and Research:
https://jiaoyangli.me/research/mapf/
https://en.wikipedia.org/wiki/Multi-agent_pathfinding
https://kei18.github.io/lacam/
https://www.researchgate.net/publication/336611576_Multi-Agent_Path_Finding_-_An_Overview

Q-Learning and SARSA:
https://www.geeksforgeeks.org/sarsa-reinforcement-learning/
https://www.geeksforgeeks.org/q-learning-in-python/?ref=previous_article
https://medium.com/codex/temporal-difference-learning-sarsa-vs-q-learning-c367934b8bcc

Debugging and Python Help:
https://chat.openai.com/

WRITTEN BY TEAM URSA:

Ly Ha 1920058
Afina Apayeva 2368692
Dalila Nguyen 1843024
Rey Ghozali 2064920