# Screeps Lite: An Environment of Exploration, Emergence, Greediness, and Sabotage

Eren Guendelsberger, Jake Israel, and James Capuder

Spring 2017

**Abstract**

Q-Learning, and the numerous extensions thereof, has become a ubiquitous reinforcement learning technique. To explore the effectiveness of Q-Learning in training multiple agents to farm resources, a training environment was implemented Python (3.5). The modular nature of the developed environment allowed for the simulation of various scenarios and reward structures, and for the possibility of future extensions.

## 1 Introduction

The primary goal of this project was to explore the phenomenon of emerging intelligence in multi-agent systems. Using an environment containing multiple non-communicating agents allowed for the observation of the strategies they developed to maximize individual utility. By testing different environments and different types of agents, global patterns of cooperation and competition emerged.

Initially, Screeps seemed like a good environment to perform these experiments in. Once we tried to implement Q-learning however, several problems emerged. Because the Screeps world has a fixed amount of time between steps and it is very difficult to save and restore specific board states, it would have taken far too long for any meaningful learning to occur. In addition, the state space is so large that any single state would likely have never been encountered again. Finally, many of the interesting problems such as pathfinding were already solved.

Because of this, it made the most sense to do this project in a simplified Screeps-like environment without the limitations of a preexisting online game. The goal of a standard agent is to gather resources from a source and bring them back to a spawn. In later environments we narrowed the focus of each agent, dividing them up into gatherers, carriers, and thieves. This allowed us to observe how the agents would act when they were forced to work together and how they would handle threats. By creating this simple clone of Screeps, we obtained more direct control over our experiments. The ability to reload and modify the starting environment and do thousands of episodes of learning in a very short time allowed for the unobstructed observation of the performance of learning methods in a multiagent system.

## 2 Prior Research

As discussed above, one shortcoming of *Screeps* for this project was the lack of any prior Artificial Intelligence research related to the game (perhaps foreshadowing the conclusions that lead to our shift in focus). Reinforcement Learning as a technique for building bots to play games, however, is a common research subject. Though our environment and learning methods were relatively simple, resources providing background on Reinforcement Learning offered valuable supplemental information to that covered in lecture. These sources are included in the References sections.

# 3 Implementation

This section details the implementation of the scripts and classes used in our framework: *tile.py, state.py, source.py, spawn.py, agent.py, grid.py, environment.py,* and finally *qlearning.py.*

## 3.1 Tile

The class used to represent each square on the grid in our simulation is `Tile`.

The constructor, `def __init__(self, x, y, tile_type)`, takes as arguments the $x$ and $y$ positions of the `Tile` on the grid, and a character representing what type of tile to initialize. These are saved as instance variables; $x, y$ are static, but `tile_type` along with a variable `self.init_type` to save the initial type of the tile so that the grid can be reset to it's initial configuration. The following table contains the mapping of characters to the tile types the represent.

Table 1: Character to type mapping

| Character | Type |
|---|---|
| o | Empty |
| s | Spawn |
| r | Source |
| a | Agent |
| x | Obstacle |
| g | Gatherer |
| c | Carrier |
| t | Thief |

Additionally, each `Tile` instance keeps track of adjacent tiles to the north, east, south, and west of the instance in a dictionary. This dictionary cannot be constructed until all tiles are initialized, so the method to do so, `initAdjacent(self, grid)`, is not called on instance initialization.

## 3.2 State

The class `State` simply holds constants that are used to clarify the indexing of the three-tuple representing each `Agent`'s state, which is discussed below.

## 3.3 Source

A source[1] is an object from which agents can collect energy to return to the spawn.

The constructor, `__init__(self, x, y, capacity=100000)`, takes $xy$ coordinates and a capacity, which is the maximum amount that can be farmed from the source. The `gather(self)`

[1] refered to as such in the code, but as a resource in the following results section

method is called whenever an agent gathers from this source, and returns a boolean value indicating whether the action was successful (false if `self.capacity==0`, true otherwise). Similarly to `Tile`, `Source` has a reset method that reverts `self.capacity` to the value with which it was initialized.

### 3.4 Spawn

`Spawn` objects are those to which agents return gathered resources.

The constructor, `__init__(self, x,y)`, takes only the *xy* position of the `Spawn` instance on the grid. The `dropoff(self)` method increments a counter representing the total number of resources returned to this `Spawn` whenever an agent performs that action, and the reset method reverts that counter to 0.

### 3.5 Agent

`Agent` is a more substantial class than those previously discussed.

The constructor, `__init__(self,environment, x, y, carry, agent_type)`, accepts an environment object, the starting *xy* position on the grid, a boolean `carry`, and the type of agent to be initialized. The agent's initial state is then constructed as the three-tuple of `(x, y, carry)`. Instance variables include the agent's Q-table as a dictionary, cumulative reward, capacity (number of resources the agent can carry, always 1), a boolean representing whether the agent's reward should be reduced (for use in the prisoner's dilemma scenario), two initially null variables indicating the agent's victim (for the thief scenario), and the agent to hand resources to (in the collaborative scenario).

`Agent` has methods:

- `update_position(self, x, y)`, which changes the coordinates in the instance's state;
- `do_action(self, action, alpha, discount_factor, reward_modifier)` simulates an agent picking an action, calculating the reward, updating the Q-table, and returning the reward;
- `mock_action(self, action)`, which makes use of the environment's mock action method to return the reward for an action without actually updating the Q-table or the agent's state;
- `get_q(self, state, action)` which extracts an entry from the the Q-table, or 0 if no entry exists;
- `update_q(self, state, action, reward, alpha, discount_factor)`, which is called by `do_action(...)` to perform the calculations on the inputs needed to update the Q-table;
- `get_explorative_action(self)` which returns a random action from the set of possible actions;
- `get_exploitative_action(self)`, which returns the action with the highest Q given the agent's current state;
- `pick_action_epsilon(self, epsilon)`, which selects an action by the epsilon greedy approach;
- `pick_action_softmax(self, tau)`, which selects an action by the softmax approach;

- get_actions(self), which calls the environment's get_agent_actions() method to get the available actions to this agent;
- and finally reset(self) to revert to the original state and reset the cumulative reward to 0.

### 3.6 Grid

Grid is a relatively simple class responsible for building and managing a board of tile objects.

The constructor for Grid, __init__(self, fpath, environment), takes an environment object and a file path pointing to a text file containing an initial board configuration. It initializes lists of agents, spawns, and sources, splits the contents of the text file into a 2D list, infers self.width, self.height from that list, and finally builds up the actual board by reading each character in the 2D list, and constructing a new Tile object for each one with the appropriate type and $xy$ position.

Grid implements the following methods:

- initTileAdjacent(self), which is called after self.board is constructed. This method populates the agent, spawn, and sources lists, and calls each tile's initAdjacent() method.
- get_tile(self, x, y) returns the tile at self.board[y][x].
- reset(self) which calls the appropriate reset method on all elements of the three lists discussed above, as well as the reset method of each tile.
- reset_all(self, components) which simply takes a list of elements and calls that element's reset method.

### 3.7 Environment

Now that we have the various components of our simulation, we cover the Environment class, which effectively serves as the shared point between Grid and Agent, helping to define how the agents are allowed to move around the grid.

The constructor for Environment, __init__(self, fname), takes just a file name, which is used to initialize a Grid instance. An important note is that the constructors for both Grid and Agent both take an environment object, so the environment from which this Grid is created, is propagated to all of these objects. The only instance variables of Environment are the Grid and a boolean value indicating whether the current episode is finished.

Environment contains the following methods:

- get_all_agents(self), which returns the list of agents stored in self.grid
- get_agent_actions(self, agent), which takes an agent and returns a list of legal moves the agent can take. This list is constructed by a series of boolean checks on the type of tile the agent is on and the agent's state, and can modify the agent's victim and handoffee variables when appropriate.
- get_agent_on_tile(self, tile) returns the agent on the specified tile, if one exists. This is used by get_agent_actions(...) when interaction between agents is allowed.

- adjacent_resource(self, agent) returns a tile adjacent to agent that has a source, or simply agent's current tile if no adjacent source exists.

- do_action(self,agent,action) takes an agent and an action, and returns the reward for that action. The difference between this and Agent.do_action(...) is that this method performs the state changes on agents, the modification of the board, and the setting of instance variables only, and relies on the agent to perform all additional calculations to transform the raw value of the reward returned here into what actually ends up in the agent's Q-table.

### 3.8 Qlearning

This is the script that actually runs the episodes, plots the data, and manages the basic GUI (enabled by setting the gui_enabled boolean to True). ENV_FILE is set to the name of a text file containing the initial configurations for a scenario, only one of which can be uncommented at a time. NUM_EPISODES, NUM_STEPS define the number of episodes to run for each Q-learning method, and the number of steps per episode, respectively. EPSILON_MODE, EPSILON1, EPSILON2 are, in order: a 0 or 1, representing whether the epsilon greedy approach should be performed, the first value for epsilon to use for simulation, and the second epsilon value to use for simulation. SOFTMAX_MODE determines whether the softmax approach should be performed, and TAU specifies the tau value for use in softmax.

The main method calls qlearning(...) for each enabled learning mode, parameterized by the flags set at the beginning of the script. This method in turn runs each episode by calling qlearn_episode(agents,discount_factor, mode, epsilon, episode), which returns a list of the cumulative rewards earned by each agent to qlearning(...). Each episode's results are saved in a list that is returned to main, and the information is then plotted with matplotlib.

## 4 Results

We tested our multi-agent q-learning implementation on a number of Screeps-like environments. We found some interesting and emergent results.

### 4.1 A Better Reward Lies Farther Away

For the first environment, we started the agent near a resource that was far away from a spawn. Way down south, there was another resource and spawn pair very close to each other. The map looks as follows.

```
soooaooor
ooooooooo
ooxoooooo
ooooooooo
ooooooooo
ooooooooo
ooooooooo
oooooosor
```

"o" means empty tile, "s" means spawn, "r" means resource, "a" means agent starting point, and

”x” means blockade. We get the following results for the agent when running it for 1000 episodes in this environment.

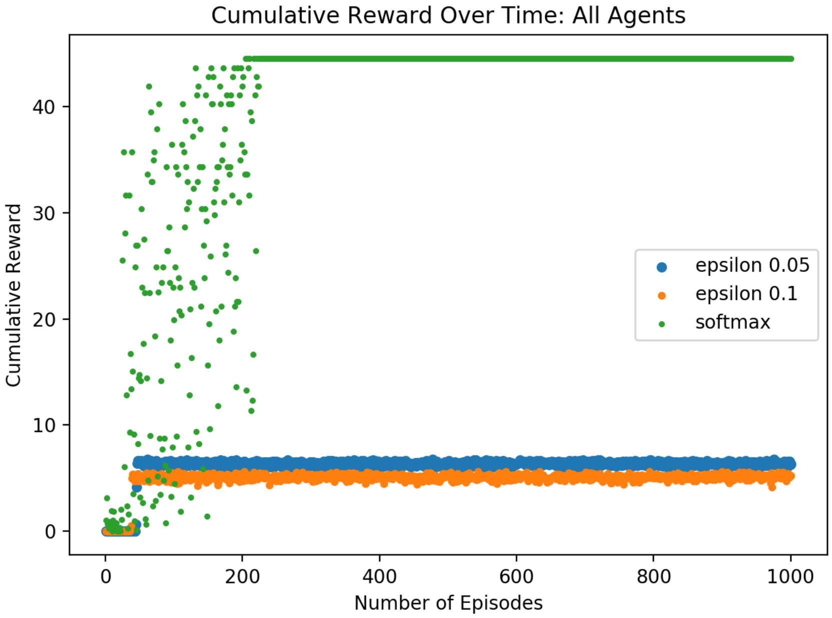


Figure 1: Single agent cumulative reward, 1000 episodes.

We notice that the two epsilon-greedy action selectors perform similarly, with a slight preference to an epsilon of 0.05. Once they find the nearby resource and spawn and learn what to do, they immediately jump up significantly in reward, thanks to the lack of unpredictable results with the actions. However, they never find the farther away resource that has a close spawn. Since they only explore less than 10% of the time, they never stray far enough away from the initial reward sequence they find to get to the other one. The variance is quite low for both of the epsilon-greedy algorithms. The little variance we notice after the improvement is due to the exploration. As we would expect, the results with an epsilon of 0.05 has less variance than the results with an epsilon of 0.1.

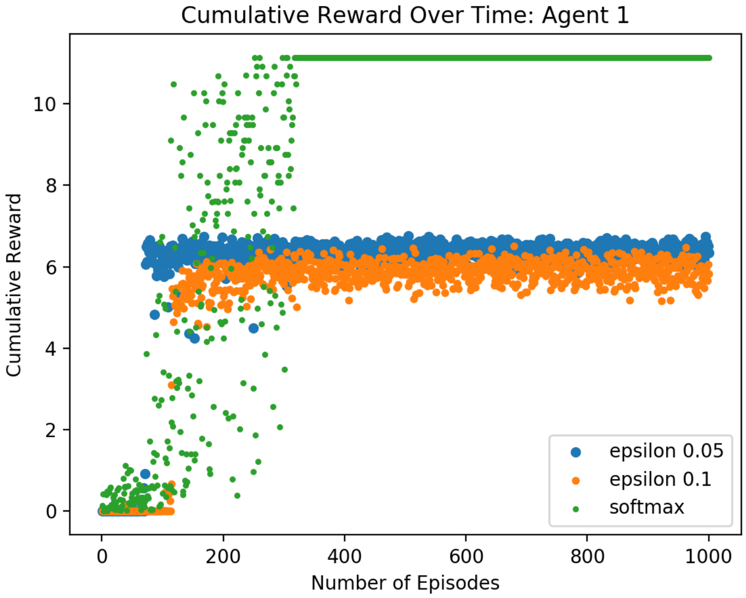
On the other hand, the softmax algorithm finds the better reward. The frequency of exploration for softmax depends on how well the algorithm is currently doing, that is to say based on how high the Q-values are. When the Q-values for the exploitative actions eclipses those of the other possible actions, the algorithm will rarely pick non-exploratory actions. The softmax algorithm is not satisfied by the resource, spawn pair that are far apart, as the q-values do not get high enough to significantly reduce exploration. This causes a significant amount of variance at the beginning of the episode. Eventually, the algorithm finds the close resource, spawn pair and learns to go there immediately at the start of the episode. Since it receives a reward every other turn (as they are right next to each other) it is very unlikely for it to explore. As a result, the variance is almost zero once the algorithm converges. As it is learning, however, the variance is significantly higher than the maximum variance interval of either of the epsilon-greedy algorithms.

## 4.2 A Dilemma

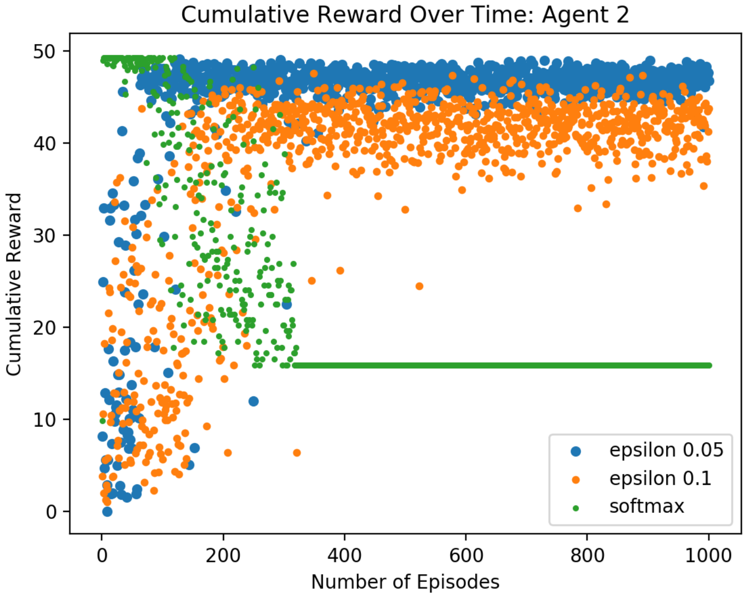
For this environment, we have two agents. The first agent starts near the far away resource, spawn pair, and the second agent starts next to the close resource, spawn pair. Note that if multiple agents mine the same resource simultaneously, they both get less than 50% of what they would otherwise due to overcrowding.

```
soooaooor
ooooooooo
ooxoooooo
ooooooooo
ooooooooo
oooooooos
ooooooooa
oooooosor
```

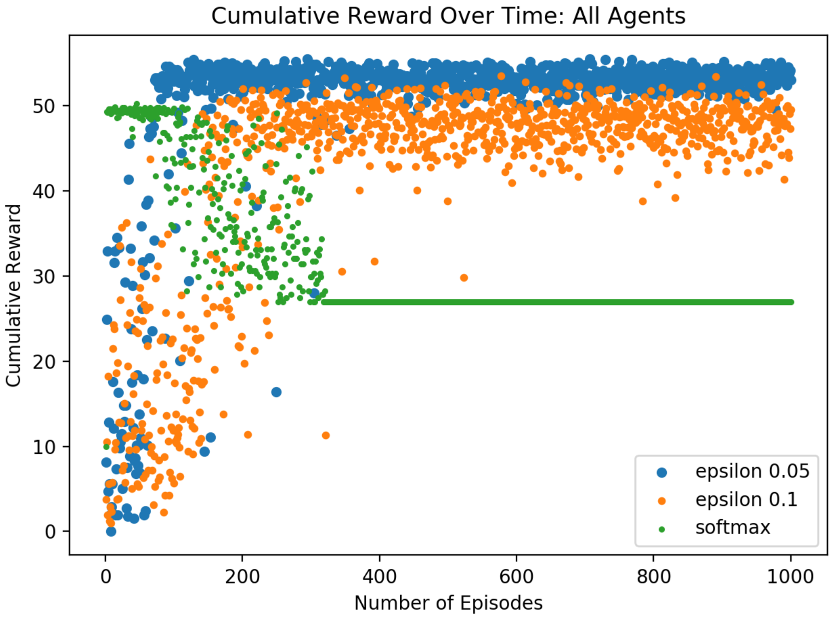
We generated graphs for each agent individually and for the environment as a whole. The purpose of this was to analyze whether greedy behavior hurts overall utility, and it does. We first present Agent 1's graph, the agent who started near the far away source, resource pair.



For the epsilon algorithms, agent 1 again never finds the closer resource, spawn pair, and its performance is mediocre. However, this also means that it does not hurt the other agent. The softmax algorithm is again able to find the closer pair. This time, the agent does not gain as much reward as before due to overcrowding, but it still gains more than when it was at the farther away resource. We now analyze agent 2 individually.



For the epsilon algorithms, agent 2 learns pretty quickly that staying where it started is the best way of earning reward. However, the variance is quite high due to the constant exploration rate. Thankfully, for these algorithms, agent 1 never finds the closer resource, so agent 2's performance stays consistent once it learns. On the other hand, softmax learns more quickly to stay where agent 2 started. However, once agent 1 finds the resource, it becomes overcrowded and agent 2's performance significantly worsens. This triggers softmax to start exploring more often, and we have a period of high variance for agent 2 from episodes 150-350. Agent 2 quickly discovers that the farther pair does not improve the results and returns to the overcrowded resource. Agent 2 cannot do anything about the agent 1's greediness. Once softmax realizes, this, agent 2's variance decreases significantly. We now analyze the overall utility.



Softmax starts out performing way better than epsilon-greedy thanks to agent 2 starting out in the perfect location. However, once agent 1 discovers the closer pair, the overall performance goes down for softmax. We notice that for the epsilon algorithms, when agent 1 never finds the location that is better for itself individually, the overall utility is better than that of softmax. This is similar to the classic prisoner's dilemma problem. We present the following game matrix with approximated values.

| | Far Pair | Close Pair |
|---|---|---|
| Far Pair | 1, 1 | 1, 4 |
| Close Pair | 4, 1 | 1.5, 1.5 |

The dominant strategy is for both agents to select the close pair, which is what happens when we use softmax for action selection. However, when one agent cannot find the close pair (when we use epsilon-greedy), overall utility is increased.

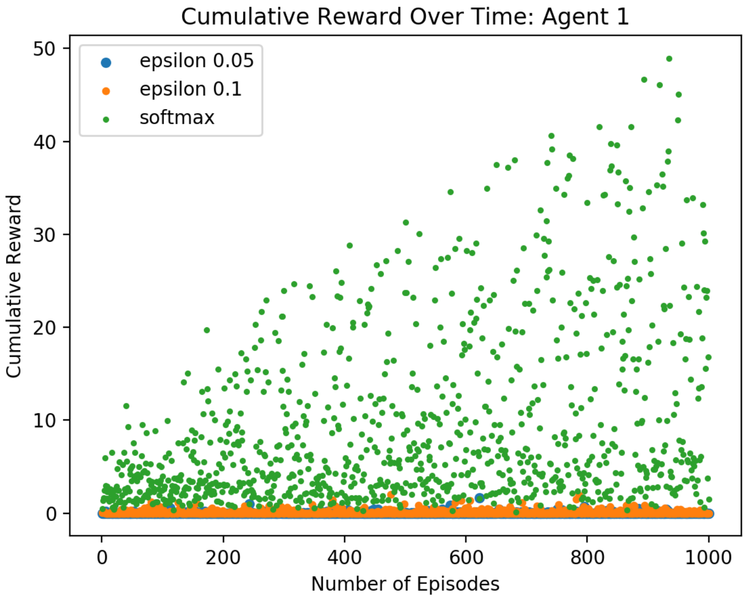
### 4.3 Working Together

We now present our results for the environment where agents have different roles. Recall that some agents are "gatherers", that is to say they can mine resources. Other agents are "carriers", and only they know the procedure for dropping them off at the spawn. All agents can carry resources around, but the gatherers must give their resources they mine to a carrier to drop off at a spawn. When we create a simple environment with only one dimension, the agents learn pretty quickly what to do. However, the agents never seem to learn for more complicated environments. We notice that at some points, the reward spikes up a bit only for it to go back down immediately.

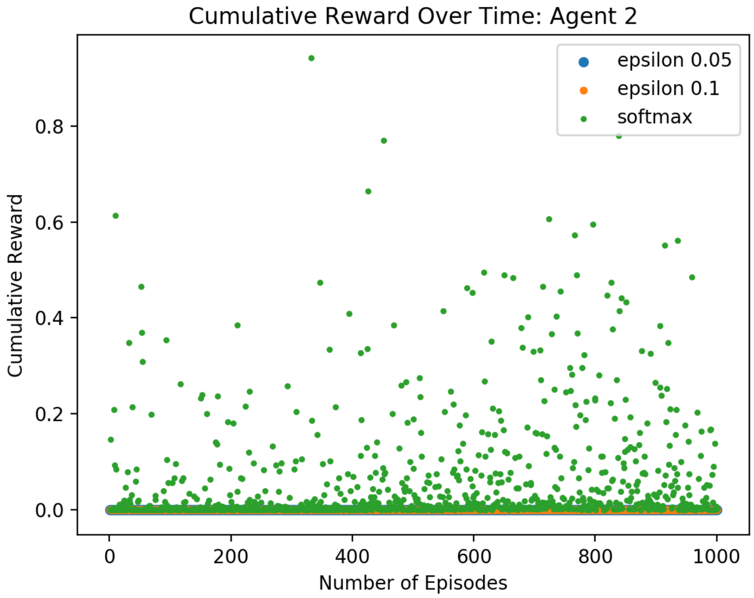
We believe this occurs due to the fact that both agents are moving. It it is difficult for them to both be in the place they expect each other to be at the right time, and when they aren't, the q-value decreases, and they eventually find each other (randomly) somewhere else. This increases the q-value for the new meeting point, but they don't end up there at the same time again. Since the agent's don't have the other agent locations as part of their state, they see the state as the same regardless of whether or not the agents are next to each other. When the desired action is not available, the value of the state decreases due to the fact that the state takes into account the reduced future reward for it taking longer to find the other agent again.

## 4.4 Sabotage

In this environment, we add a thief (agent 1) who's goal is to steal resources from the other agent (agent 2). The only source only has one access point, as the others are blocked by either a blockade or an edge. The goal of the thief could theoretically make it impossible for agent 2 to stow any resources, but the thief won't get any reward unless it chases after agent 2 and steals resources. First, we present the results for the thief.



We notice that the thief is only successful with the softmax algorithm. Even then, the variance increases over time despite the evident learning. This is perhaps due to the unpredictable nature of the other agent. Based on our observation of the "good" agent, it alternates between running away and trying to (mostly unsuccessfully) stow a resource. We can see in the graph of agent 2 that the thief has a huge advantage here.



Ultimately what makes this environment interesting, is to think about the agent's incentives. If the thief tries chasing agent 2 around, then agent 2 might be able to run around the thief and get to the source. On the other hand, if it stays by the source, agent 2 may learn to never come near, resulting in both agents to receive minimal reward. If only the agents could agree to alternate between letting one steal and the other stow. That would cause maximum overall utility but cannot occur when they are both greedy, not to mention untrustworthy. Ultimately, the agents are making the right decision given their inability to communicate. In a larger environment, we could encounter zugzwang, a concept often used in chess to describe the situation where whoever makes the first move loses. If the thief moves away from the spawn, the agent can get there. But if the agent tries going to the spawn before the thief moves, the thief will get the agent. Whoever makes the first move loses.

## 5 Future Work

The next step we would take in this project is to add in shared Q-tables. This would change the motivation of each agent, and instead of being greedy they would try to maximize the system's total utility. This could result in new emergent behaviors such as cooperation between thieves and their victims, where they alternate between stealing the resource and turning it in. It would also greatly increase the cooperation between gatherers and carriers, assuming we modified their states to include the location of the other.

## 6 Conclusion

In this project, we build a simplified clone of Screeps that allowed for direct control over our experiments. We applied Q-learning techniques in a number of environments and observed the patterns of behavior that emerged. In the first environment which contained only one agent, we found that the softmax approach was able to find the most optimal solution while the epsilon-greedy approaches only found a sub-optimal solution. In the prisoners dilemma environment, we found that the system's total utility was lower when the agents favored exploration over exploitation. This is because they are greedy and non-communicating, so each will choose whichever strategy maximizes its own utility. Given the opportunity, the agent which starts far away from the optimal source-spawn path will go there even at the expense of the other.

In the environment which divides up the responsibility between gatherers and carriers, we found that cooperation is extremely hard without communication. Because each agent doesn't contain the other agent's location within its state, its Q-table fails to direct it towards the nearest potential handoff but instead towards the spot where it has most often handed off in the past. We found that when the agents met, it was usually a coincidence, indicating that little meaningful learning occurred. With the introduction of a thief agent, a whole new layer of complexity was added. Because the thief and its victim are both using Q-learning, their greedy approaches lead to a stalemate. If the thief starts camping the spawn, then its victim will learn to avoid the spawn. The behavior that emerges is "the first to move loses", where making the first move gives the other agent the advantage. This makes perfect sense in a pair of non-communicating greedy agents, because neither is willing take a negative reward by making a move that its opponent can exploit.

In the end, we created a system that allowed for the unobstructed observation of the performance of Q-learning methods in multiagent systems. Using this system, we modeled a number of different scenarios and explored the emergent behavior of non-communicating greedy agents.

## 7 References

A review of AI in real time strategy games.

This paper provides a survey of current research on AI applied to real-time strategy games, with a focus on the game Starcraft. Most real-time strategy games, including Starcraft, have some similarities in the objectives and possible actions, so we had hoped to apply some of the information to Screeps before switching to our own implementation. Some of the techniques discussed involve observing good human players and scraping data to input to learning algorithms. Unfortunately, we cannot apply those methods to our environment.

Reinforcement Learning

This article discusses the relative merits of supervised learning, unsupervised learning, and reinforcement learning as methods of machine learning. It then delves into reinforcement learning, giving a detailed overview before moving on to a pair of research articles. The first goes over Google Deep Minds attempts to use reinforcement learning with a neural network to beat Atari games. The second discusses Alpha Go and how it used reinforcement learning to beat the worlds greatest Go players.

A Survey of Emergent Behavior and Its Impacts in Agent-based Systems

This paper is about emerging intelligence and helped us improve our understanding of how emergence works and how it can be used in multi-agent systems.