Multi-agent reinforcement learning for unit control in the programming strategy game Screeps

Ben Bartlett *
Department of Applied Physics, Stanford University

1 Introduction

Screeps is a multiplayer real-time strategy (RTS) game for programmers. [1] The core game objective is to expand your colony, gathering resources and fighting other players along the way. To manipulate your units, you write a script to control their behavior in JavaScript; everything from moving, harvesting, building, fighting, and trading is entirely driven by your code, which is executed on a single server that runs 24/7, populated by every other player and their armies of creeps.

Screeps has nearly limitless depth to it, with a variably-sized action space which, even for moderate numbers of agents, can be far larger than that of many popular single-agent games such as chess. It shares many similar mechanics with other well-studied multi-agent games, such as StarCraft, providing a convenient body of literature on which to base initial experiments [2, 3, 4, 5]. Because Screeps is a complex environment which can be interacted with entirely programmatically, it provides an ideal platform for studying multi-agent reinforcement learning concepts.

As in many RTS games, combat in Screeps is highly nuanced, with certain configurations of positions being favorable over others, allowing for coordinated movement strategies with complexity beyond what could feasibly be hand-coded. For this project, I implemented a training environment which allows a Python-based reinforcement learning agent to interact with many remote, vectorized instances of the Screeps backend, with the ultimate goal of being able to discover effective combat strategies. The training environment allowed for scalability and distribution of processing to obtain step throughputs which would otherwise be impossible running with a single, non-vectorized server. I performed two sets of experiments using the training environment I developed and was able to demonstrate desirable movement behavior and effective but preliminary combat behavior.

I describe some of the basic game mechanics in Section 2, detail the architecture and implementation of the training environment in Section 3, and discuss the feature extraction and results of the experiments run using this environment in Section 4.

2 Game mechanics

The game time is measured in `ticks` (cycles); every player's script instructs their units to perform actions synchronously every tick, and the game state is updated at the end of the tick. The unaltered Screeps server backend can typically run one room at approximately 30 ticks/sec, which is prohibitively slow for training, necessitating clever use of batching and vectorization.

Player scripts are executed within their own isolated virtual machines running inside of the server. The scripts have limited CPU time to complete their execution and have access to a 2MB persistent `Memory` object, which is serialized into the server database at the end of each tick. I directly read and write to the serialized `Memory` object to communicate between the RL agent and the isolated VM.

The game world consists of a map of interconnected rooms. A `Room` is a 50×50 lattice of `RoomPositions`, which have terrain that can hinder or block unit movement. The units of the game are `Creeps`, which have bodies specified by an array of parts, allowing the creep to perform

*email: benbartlett@stanford.edu

various actions, such as moving, building, attacking, ranged-attacking, and healing. The effectiveness of each action scales with the number of corresponding body parts.

There are many other game mechanics in Screeps, including room structures, spawning, and static defenses, but I omit the details of these because they were not addressed in any of the experiments performed in this project (although the training environment is capable of simulating them). In-game screenshots depicting some of the game objects discussed here are shown in Figure 1.

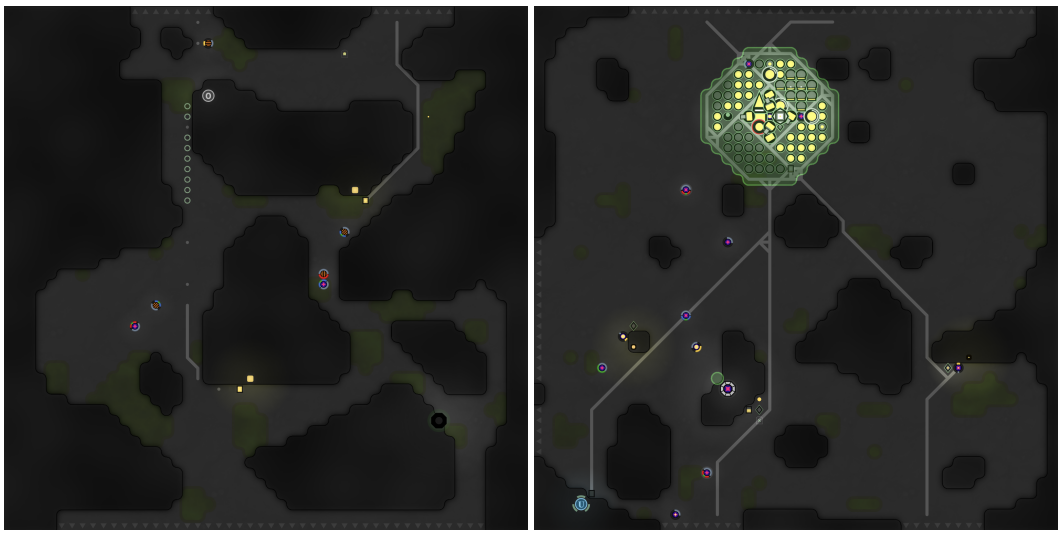

Figure 1: In-game screenshots depicting combat in a neutral room (left) and a completed base built in an owned room (right). The color of each tile indicates the type of terrain: black is impassible, dark green hinders but does not block movement, and grey is standard. Creeps are the multi-colored circles; their body parts are shown by the circular colored arcs around their perimeter.

3 Training environment architecture

The training environment I implemented for this project has three main components, encapsulated as separate packages: in increasing order of abstraction, `screeps-rl-backend`, a node.js package which exposes methods via remote procedure call (RPC) to controllably initialize, reset, and step the Screeps server and communicate with the underlying agents; `screeps_rl_env` which provides standardized `gym`- and `rllib`-style Python training environments for agents to interact with; and `models`, which leverages customized implementations of various RL algorithms within `rllib` to train within the environment and provides scripts for automatically deploying training to a dynamically scalable Google compute cluster. A UML diagram of the full training environment is shown in Figure 2, and I discuss each package in greater detail below.

3.1 Low-level interactions: `screeps-rl-backend`

The `screeps-rl-backend` is written in node.js and allows for low-level interaction with the Screeps server. The main logic for interfacing with the server is contained within the `ScreepsController` class, which takes a set of ports to start the server on as inputs and contains methods for initializing, running, and stepping the Screeps server. The server stores the game state using a LokiJS database; player scripts are executed within isolated VMs and have access to a serializable `Memory` object stored in the database. The controller class contains methods to read the game state from the database and to write serialized action commands directly to the `Memory` object. I used the `ActionParser` object within Overmind [6] (a popular open-source Screeps bot I developed) to interpret the serialized actions.

A `ScreepsController` class is instantiated by a remote server wrapper, which exposes the methods of the controller to remote procedure calls, allowing the backend to interface with the Python environment.

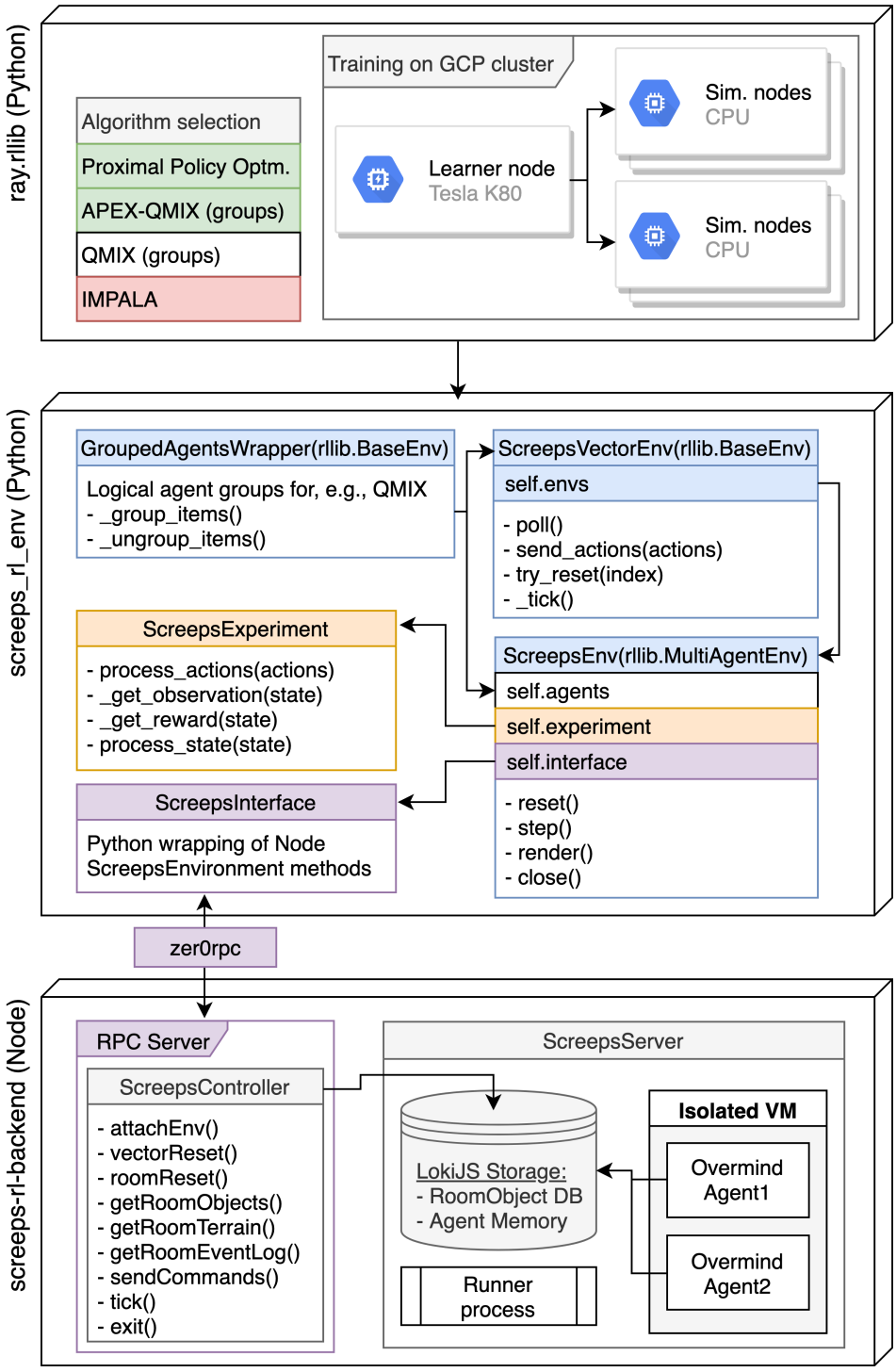

Figure 2: Structure of the training environment I developed for this project. A node.js backend provides low-level interaction with the Screeps server, relaying observations and actions by reading and writing to the storage database. A Python environment conforming to `rllib` standards allows for vectorization and execution of environments on remote nodes, providing high-level interaction with a reinforcement learning agent. Training algorithms were based on `rllib` implementations and were deployed on a scalable Google compute cluster.

3.2 High-level environments: `screeps_rl_env`

The `screeps_rl_env` package provides standardized environments for an RL agent to interact with. The environment communicates with the `ScreepsController` via zer0rpc [7] contained within the `ScreepsInterface` class.

There are multiple environments contained within the package, extending the `gym.Env`, `rllib.Env`, `rllib.VectorEnv`, and `rllib.MultiAgentEnv` classes, but the primary (and most full-featured) environment is the `ScreepsMultiAgentVectorEnv` class, extending `rllib.BaseEnv`. This environment contains several instances of the non-vectorized multi-agent environment class which contain a shared pointer to the same `ScreepsInterface`, allowing for "vectorization" as discussed in Sec. 3.2.1. Implementing multi-agent vectorization was a major architectural challenge in this project.

The environments take as an initialization argument a `ScreepsExperiment` class, which contains methods for processing actions and room states, as well as for generating observations and rewards at each step. This allows the same environment class to be run with multiple different experiments, which can select for different behaviors. The two experiments tested in this project are detailed in Sec. 4.

The vectorized multi-agent environment also has a method which wraps the environment in the `GroupedAgentsWrapper` class, which provides groupings for agents operating within the same logical super-agent, e.g. multiple units on the same team. This allows for leveraging some relevant explicitly-coordinated multi-agent algorithms like QMIX [8] during training.

3.2.1 Implementing multi-agent vectorization

The Screeps server adds a large amount of overhead, and it is more efficient to simulate many rooms on a single server than many single-room servers, which would be the default way of vectorizing the environment. This poses a challenge, however, as the native vectorization in `rllib` starts many environment instances in parallel and assumes that each can be stepped independently. This is not the case in this application, however, as all actions must be sent to each sub-environment (disconnected room), then the vector environment must be ticked, stepping all sub-environments at once, then all observations must be gathered. This necessitated working with the lowest-level `rllib` environment, `BaseEnv`, and implementing a custom `GroupedAgentsWrapper`, which natively did not support `BaseEnv`.

After implementing multi-agent vectorization, I observed an increase in tick throughput (defined as $\frac{\text{ticks}\times\text{rooms}}{\text{sec}\times\text{core}}$) of about an order of magnitude. The optimal number of rooms per server to obtain maximum throughput seems to be approximately 10.

3.3 Distributed training with `rllib`

To train agents in the Screeps environment, I used the `rllib` platform [9], a part of the `ray` project [10]. I configured the auto-scaling cluster architecture provided in `ray` to rent out a dynamically-sized Google compute cluster. The head node hosted the policy optimizers and was accelerated by an NVIDIA Tesla K80 GPU. The worker nodes were `n1-standard-2` instances which would periodically fetch the latest policy parameters from the head nodes and rollout episode data to a shared replay buffer, which would be used to train the policy. This allowed the training environment to attain a peak tick throughput of 1900 room·ticks/sec when running with 64 worker cores.

`rllib` includes a number of reinforcement learning algorithm implementations out of the box. Once I had implemented the training environment in a format which was compatible with the library, I was able to experiment with the algorithms to identify strong and weak performers. I found Proximal Policy Optimization [11] and QMIX [8] / APEX-QMIX [12] to be particularly effective. The latter two require the grouping logic implemented by the custom `GroupedAgentsWrapper` described in Sec. 3.2.

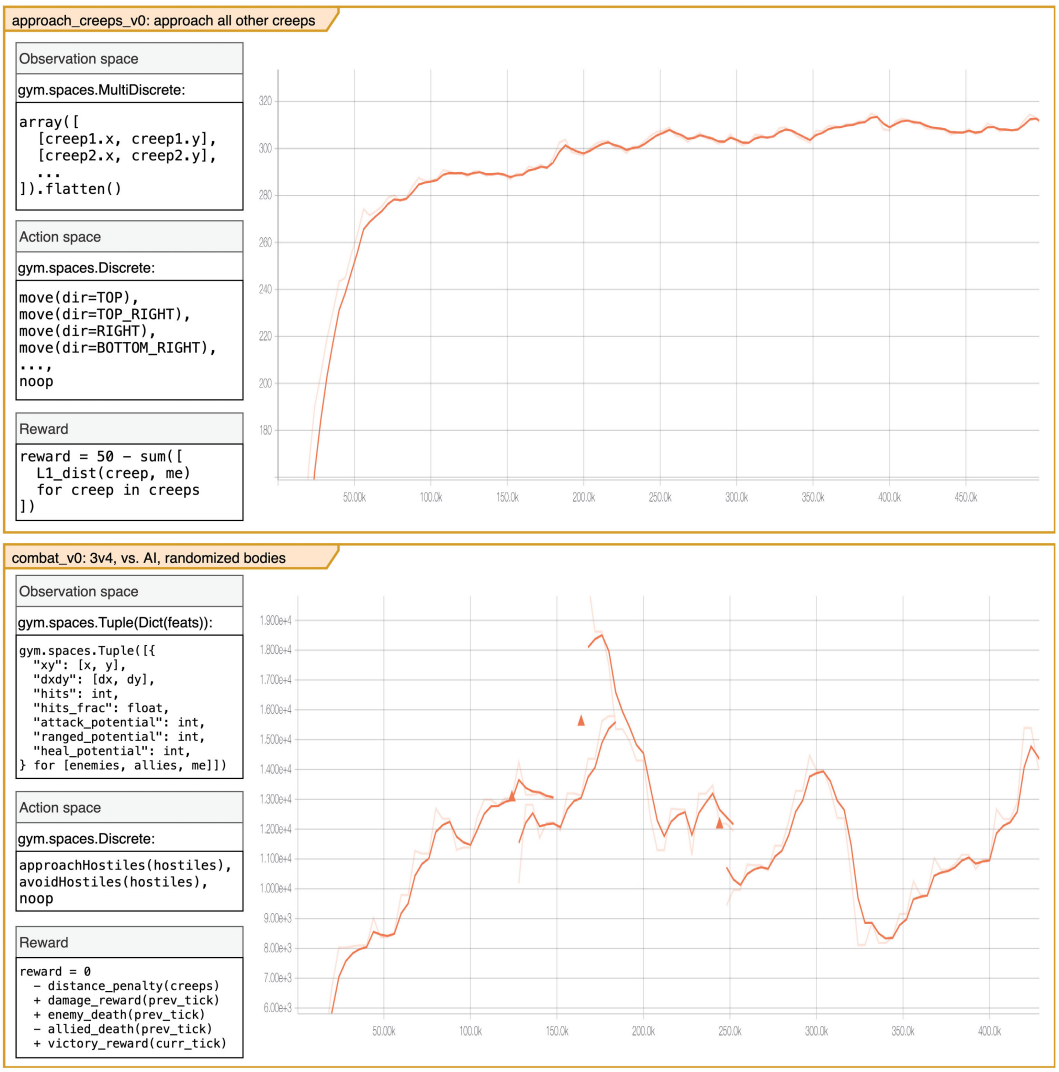

Figure 3: Observation space, action space, reward function, and training performance of the two experiments performed for this project. The top panel depicts an experiment to teach units to group together and move around the map, while the lower panel depicts teaching units rudimentary combat skills. Discontinuities in the reward plots are due to failures in worker nodes during training, causing the policy to revert to a recent checkpoint. The variability of the reward function for the `combat_v0` experiment is discussed below.

4 Experiments

I ran two sets of experiments using the Screeps training environment described in the previous section. The observation space, action space, reward function, and reward over training iterations for both experiments are depicted in Figure 3.

The first experiment, `approach_creeps_v0`, aimed to teach units to navigate toward each other and clump together. The units would spawn in random positions in the environment and were provided with a `MultiDiscrete` (or in the second version, a `Box`) observation space which contained the coordinates of all other units. The action space was a `Discrete` space consisting simply of raw `move()` actions (no access to higher-level pathfinding methods) in 8 directions and `noop`. The maximum reward for an episode depended on the initial spawn positions of the units, but is theoretically 376. Reviewing visualized rollouts of the trained behavior, I also qualitatively observed the desired grouping behavior, followed by random movement as a group around the map.

The second experiment, `combat_v0`, aimed to teach units rudimentary combat behavior. Two teams of units would spawn, each unit in a random position, and the units would have randomized body compositions. The RL model would control a team of 3 units, and the opposing team of 4 units were controlled by a purposefully simplistic script consisting of the logic "approach the nearest enemy and melee/ranged attack it, healing yourself if possible". (Notably, this logic omits decisions for when to retreat from hostiles or when to group up with allies.) The observation space for each unit was based on the data representation for StarCraft units in Ref. [3]; the space consisted of a `Tuple` of `Dict` spaces for each ally and enemy unit, each consisting of the coordinates of the unit, the displacement of the unit from self, the hits and maximum hits of the unit, and the number of melee, ranged, and healing parts on the unit. The action space was a `Discrete` space of approaching or avoiding hostiles and `noop`. To simplify this preliminary model, attack functions were handled procedurally by the script, while movement was handled by the policy. Each unit was given a reward that was penalized very slightly for distance from other units, rewarded for dealing damage the previous tick, rewarded highly for killing a unit the previous tick, penalized for an allied unit dying the previous tick, and rewarded very highly when all enemy units were dead on the previous tick.

The cumulative reward over training for this model was unstable, likely due to the randomization of body compositions giving one side a variably large advantage over the other. (In a future version of this experiment, I will choose random body compositions that sum to a fixed "power budget" for each side.) Discontinuities in the reward function shown in Figure 3 are due to failures in worker nodes during training, causing the policy to revert to a recent checkpoint and resume training. As with the first experiment, the maximum theoretical reward for `combat_v0` depends on spawning locations and on body compositions, but is approximately 25000.

5 Conclusions and future work

In this project I implemented a parallelizable and scalable training environment which allows for a reinforcement learning agent to interact with the Screeps game environment. Using distributed computing utilities provided by `rllib`, I was able to achieve a tick throughput which was suitably high for training purposes. I ran two experiments to test this environment, qualitatively and quantitatively observing that the units learned the desired behaviors in both cases.

I did not have much time to explore the experiment and data representation portion of the project, as implementing the training platform and learning the `rllib` framework consumed most of my time, so I would like to improve the models in the future. Specifically, I would like to incorporate spatial representations of both terrain data and unit data and leverage convolutional layers in the networks which estimate the Q values.

Source code

The source code for this project is available in a repository at `https://github.com/bencbartlett/Overmind-RL`, which is currently private. A temporary access token was included in the email where I submitted this paper.

References

[1] Artem Chivchalov. *Screeps: MMO strategy game for programmers*. 2014. URL: `https://screeps.com/`.

[2] Kun Shao, Yuanheng Zhu, and Dongbin Zhao. "StarCraft Micromanagement with Reinforcement Learning and Curriculum Transfer Learning". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 3.1 (2018), pp. 73–84. DOI: `10.1109/TETCI.2018.2823329`. URL: `http://arxiv.org/abs/1804.00810`.

[3] Kun Shao, Yuanheng Zhu, and Dongbin Zhao. "Cooperative reinforcement learning for multiple units combat in starCraft". In: *2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017 - Proceedings* 2018-Janua (2018), pp. 1–6. DOI: `10.1109/SSCI.2017.8280949`.

[4] Amirhosein Shantia, Eric Begue, and Marco Wiering. "Connectionist reinforcement learning for intelligent unit micro management in StarCraft". In: *Proceedings of the International Joint Conference on Neural Networks* (2011), pp. 1794–1801. DOI: 10.1109/IJCNN.2011.6033442.

[5] Peng Peng et al. "Multiagent Bidirectionally-Coordinated Nets: Emergence of Human-level Coordination in Learning to Play StarCraft Combat Games". In: (2017). URL: http://arxiv.org/abs/1703.10069.

[6] Ben Bartlett. *Overmind Screeps AI*. 2019. URL: https://github.com/bencbartlett/Overmind.

[7] 0rpc. *zer0rpc: An easy to use, intuitive, and cross-language RPC*. 2019. URL: https://www.zerorpc.io/.

[8] Tabish Rashid et al. "QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning". In: (2018). URL: http://arxiv.org/abs/1803.11485.

[9] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: (2017). URL: http://arxiv.org/abs/1712.09381.

[10] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: (2017). URL: http://arxiv.org/abs/1712.05889.

[11] John Schulman et al. "Proximal Policy Optimization Algorithms". In: (2017), pp. 1–12. URL: http://arxiv.org/abs/1707.06347.

[12] Dan Horgan et al. "Distributed Prioritized Experience Replay". In: (2018), pp. 1–19. URL: http://arxiv.org/abs/1803.00933.