SMAClite: A Lightweight Environment for Multi-Agent Reinforcement Learning

Adam Michalski
University of Edinburgh
Edinburgh, United Kingdom
contact@adammi.ch

Filippos Christianos
University of Edinburgh
Edinburgh, United Kingdom
f.christianos@ed.ac.uk

Stefano V. Albrecht
University of Edinburgh
Edinburgh, United Kingdom
s.albrecht@ed.ac.uk

ABSTRACT

There is a lack of standard benchmarks for Multi-Agent Reinforcement Learning (MARL) algorithms. The Starcraft Multi-Agent Challenge (SMAC) has been widely used in MARL research, but is built on top of a heavy, closed-source computer game, StarCraft II. Thus, SMAC is computationally expensive and requires knowledge and the use of proprietary tools specific to the game for any meaningful alteration or contribution to the environment. We introduce SMAClite – a challenge based on SMAC that is both decoupled from Starcraft II and open-source, along with a framework which makes it possible to create new content for SMAClite without any special knowledge. We conduct experiments to show that SMAClite is equivalent to SMAC, by training MARL algorithms on SMAClite and reproducing SMAC results. We then show that SMAClite outperforms SMAC in both runtime speed and memory.

KEYWORDS

Multi-agent Reinforcement Learning, Starcraft, Strategy, Game

1 INTRODUCTION

As we continue to make advancements in artificial intelligence research, it inevitably makes its way into our daily lives. Examples of autonomous agents popular in recent times range from robotic vacuum cleaners (e.g. [30]) and self-driving cars (e.g. [2]) to robots making lives easier from behind the scenes, such as warehouse optimization robots (e.g. [15]). With all this attention in research, a natural need arises for standardized benchmarks for the various types of artificial intelligence (AI) models.

Multi-agent reinforcement learning (MARL) is a branch of machine learning dealing with multiple autonomous AI entities – usually called *agents* – existing in the same environment. In this work, we are interested in *cooperative* agents, ones that work together to accomplish some goal. Crucially, there is currently no consensus in the research community about what a standard benchmark for this type of agent should be. Just by looking at a recent benchmarking paper [20] we can count five different benchmarking environments.

The Starcraft Multi-Agent Challenge (SMAC) [24] has been widely used in MARL research. It is built on top of a real-time strategy computer game Starcraft II (SC2) and makes use of an API – an interface between the game and the AI agents – made available by Vinyals et al. [32]. It presents a mini-game where each agent controls a single combat unit (e.g. a single soldier) in one of several available battle scenarios against an enemy team controlled by the game's built-in AI. SMAC presents a challenge where the solution is not straightforward – in most of the scenarios the most obvious strategy of running forward and attacking is not good enough and will result in a quick loss due to the enemy army having better units or more numbers.

While the idea is promising, we can spot several problems with SMAC if it is to become a universally accepted benchmark. The biggest issue we see is that SMAC uses SC2 as its key dependency, requiring a large-sized (ca. 3.7 GB) download and a complicated setup process for any training or inference, not to mention running SC2 alongside SMAC at all times, consuming extra CPU and memory resources. This is made worse by the fact that SMAC uses only a subset of the SC2 features – a lot of the required downloadable and computational resources are simply redundant, and due to that training agents on SMAC is more expensive than necessary for the task it offers.

On top of that, it remains highly inaccessible for people unfamiliar with the game it is based in. This manifests in many ways, e.g. to create custom scenarios or units for SMAC, one is required to use the official Starcraft II map editor, requiring people to learn an unusual and proprietary tool, and put in a lot of effort for a single benchmark.

We present a challenge very similar to SMAC, but completely decoupled from the Starcraft II dependency, and show that it preserves the challenging aspects of SMAC. We name this challenge Starcraft Multi-Agent Challenge lite (SMAClite). We also want to make the battle scenarios and units as easy to modify as possible – also allowing easy creation of completely new ones. Our environment preserves the outer interface of SMAC, only changing the inner workings, in order to allow AI developers to reuse their code for handling SMAC with minimal modifications. We make this environment open-source[1] and free to use.

We perform a series of experiments using models trained on SMAClite. Our experiments include quantitative analysis by comparing the return achieved by various MARL algorithms in SMAClite – we show that the algorithms achieve similar returns as on SMAC and that the relative ranking among them is preserved from SMAC. We also perform qualitative analysis – looking at the combat strategies employed by the agents on a case-by-case basis, and verifying they do indeed outsmart the handwritten enemy AI. On top of that, we take the models trained on SMAClite, and put them inside the original SMAC environment without any further training, to see how much potential for transfer learning there is between the environments – from this experiment, we conclude that training agents on SMAClite improves the returns achieved by them on SMAC, and therefore the challenges require similar skills.

[1] https://github.com/uoe-agents/smaclite

2 RELATED WORK

Popular benchmarks, accepted by the research community as the standard, do exist for the single-agent variant of reinforcement learning [5, 22]. The authors of the MARL benchmark paper [20] make available two MARL benchmarks, both based in simple 2D worlds: Level-Based Foraging, and Multi-Robot Warehouse. There are also several other non-Starcraft II multi-agent benchmarks, such as the Multi-Agent Particle Environment [19], or the Hanabi challenge [3]. Another multi-agent challenge based on a modern computer game is the OpenAI Five project [7], which put agents inside a full five versus five matches of the strategy game Dota 2, showing the impressive scale of the game and the trained models.

Training autonomous agents in Starcraft II started becoming popular upon the publication of its API [32], with one of the popular results being the AlphaStar model [31]. Our work is mostly based on the work of the creators of SMAC [24]. Since its publication, SMAC has been used as a benchmark for autonomous agents in numerous works (e.g. [20, 23, 33]), with its popularity being a sign that it does fill a niche.

There are several projects that are tangentially related to ours since they also aim to improve SMAC but take different approaches (e.g. SMACv2 [9], SMAC+ [14]). These projects present interesting additions to the SMAC paradigm, however, to our knowledge, none of them address the issues we wish to tackle: the environment's performance cost, and its closed-source nature. We believe the additions introduced by them are good ideas for the future development of SMAClite, but the scope of our project is to maintain the challenge's difficulty on the same level as the original.

3 BACKGROUND

3.1 Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement learning (MARL) allows for multiple autonomous agents to coexist in the same space. Formally, the setup of MARL consists of several *agents* that can perform specific actions, and the *environment* – a term that encompasses everything outside of the agents, that the agents can interact with. The specific formalisation of MARL that we use in our work is called **Decentralized Partially Observable Markov Decision Problems** (in short: Dec-POMDPs).

A Dec-POMDP is a cooperative process defined as a 7-element tuple $(\mathcal{N}, \mathcal{S}, \mathcal{A}, \mathcal{O}, \Omega, \mathcal{P}, \mathcal{R})$, where $\mathcal{N} = \{1 \dots N\}$ is the set of agents participating in the process. Agents interact with the environment in discrete timesteps $t \in \mathbf{N}$. In each timestep t the environment has some true active state $s_t \in \mathcal{S}$, and each agent i receives an observation $o_t^i \sim \Omega(i, s_t),\ o_t^i \in \mathcal{O}$. Each agent i then selects an action $a_t^i \in \mathcal{A}$. After each timestep t the agents receive a shared reward $\mathcal{R}(a_t^1, a_t^2, \dots, a_t^N, s_t) = r_{t+1} \in \mathbf{R}$, and the environment enters the next state $s_{t+1} \sim \mathcal{P}(a_t^1, a_t^2, \dots, a_t^N, s_t),\ s_{t+1} \in \mathcal{S}$.

In a Dec-POMDP, like in reinforcement learning in general, the agents' goal at each point in time t is to maximize the discounted cumulative reward (or the *return*) $\sum_{i=0}^{\infty} \gamma^i r_{t+i}$, where γ is a discount factor. When $\gamma = 1$, the discounted return is equal to the actual return $\sum_t r_t$ – we will report this sum when presenting evaluation results.

For the purpose of training and evaluating agents in the SMAClite environment, we will use the same set of algorithms, as well as their hyperparameters, that was used to test agents in SMAC in a recent MARL benchmark paper [20]. This includes 9 popular algorithms that can be used to solve various SMAC scenarios: IQL [28], IA2C [18], IPPO [25], MADDPG [17], COMA [10], MAA2C [20], MAPPO [33], VDN [27], and QMIX [23].

3.2 SMAC

3.2.1 Overview. In this section, we go over the SMAC environment to the extent that is relevant to our project. Our project only replicates SMAC in its default configuration, omitting any optional parameters, so we omit those here as well.

In SMAC, each agent controls one unit (we refer to units controlled by agents as *allied units* throughout this paper) and is tasked with defeating a group of enemy units controlled by Starcraft II's built-in AI opponent. SMAC defines several combat scenarios, differing in army compositions and terrain layout, and as result, in difficulty. We include visualizations of two SMAC scenarios in Figure 1.

Units are divided into several types with different attributes (health, attack, etc.) – but note that the SMAC environment only makes a distinction between unit types in the state/observation vectors if there is more than 1 unit type within a single team in the scenario – we will refer to this as the scenario "distinguishing unit types".

Some unit types, once hit, do not regenerate health in any way. Other units have innate health regeneration that is always active. Yet another group of unit types has special shields on top of their health points that regenerate after a period of not taking damage, and the shields have to be brought down to 0 before the units' health can be hit. Due to the limitations of Starcraft II, either all units in a team possess shields or none of them.

3.2.2 Actions. Each agent has access to several actions, which may or may not be available at any given timestep – the environment exposes a method to get currently available actions for each agent. If an unavailable action is chosen by the agent, SMAC raises an error and ceases execution. The possible actions are `no-op` – which has no effect and is only available to dead units, `stop` – orders the unit to stop in its place and do nothing, `moveN`, `moveE`, `moveS`, and `moveW` – orders the unit to move in the chosen cardinal direction (north, east, south, or west), `target1`, `target2`, … – orders the unit to target the unit with the specified team-specific ID – for damage-dealing units, this refers to targeting enemy units to attack, for healing units, to targeting allies to heal. SMAC defines a constant targeting range for agents, and this action is unavailable if the target is outside of this range.

3.2.3 State. The true state of the environment is a vector divided into three sections. The first section contains each ally unit in order of their IDs: its current health, its current cooldown, its X and Y coordinates (relative to the centre of the map), its current shields (only if allies have shields), and a one-hot vector representing its unit type (only if scenario distinguishes unit types). The second section contains each enemy unit in order of their IDs: its current health, its X and Y coordinates (relative to the centre of the map),

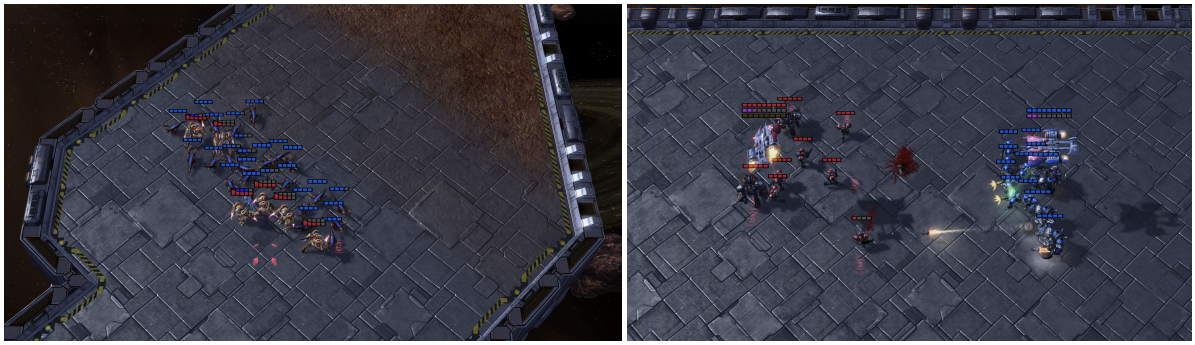

Figure 1: Visualizations of the SMAC environment. *(left)* The `corridor` scenario. *(right)* The `MMM2` scenario.

its current shields (only if enemies have shields), and a one-hot vector representing its unit type (only if the scenario distinguishes unit types). The final section contains each agent in order of their IDs: a one-hot vector representing the action taken by them in the previous timestep. Note that all features within the state vector are normalized to be between zero and one – for example, the health value is divided by the maximum health value of the given unit.

3.2.4 Observations. Each agent, in each time-step, receives an observation representing what is visible to the agent within the environment. SMAC defines a constant sight range for agents – if a unit is dead, or is further from the agents' own unit than this sight range, any information in the observation vector about this unit is completely zeroed out. The observation vector is divided into four sections. The first section includes, for each cardinal direction, whether the movement is possible in that direction. The second section includes, for each enemy unit, if it is alive: whether the agent's unit can attack it, its distance to the agent's unit, its X and Y coordinates relative to the agent's unit, its health, its shields (only if enemies have shields), and a one-hot vector representing its unit type (only if the scenario distinguishes unit types). The third section includes, for each ally unit: a literal 1 (to distinguish from units that are dead or too far), its distance to the agent's unit, its X and Y coordinates relative to the agent's unit, its health, its shields (only if allies have shields), and a one-hot vector representing its unit type (only if the scenario distinguishes unit types). The final section includes, for the agent's own unit: its health, its shields (only if allies have shields), and a one-hot vector representing its unit type (only if the scenario distinguishes unit types). Note that, similarly to the state features, all observation features are normalized to be between zero and one.

3.2.5 Rewards. After each time-step all agents receive a shared reward equal to the sum of health points and shield points removed from enemies in that timestep. A small bonus of 10 is added for each eliminated enemy unit and a bigger bonus of 200 is added for winning the scenario. The reward is normalized by dividing it by the sum of all health and shield points of enemy units and any possible bonuses, and multiplying by 20 – thus, the possible returns should be between zero and twenty. Do note, however, that the actual cumulative reward received by the agents by the end of an episode might exceed 20 due to health and shield regeneration.

3.3 Optimal Reciprocal Collision Avoidance

In Starcraft II, units move around the battlefield populated by other units and avoid collision by stepping to the side if they would get in the way of another unit. This makes the battlefield feel more realistic and physical and allows for some advanced strategies like body-blocking or surrounding (in essence, limiting other units' movement by positioning oneself strategically). We felt it was crucial to reproduce this behaviour in our environment. However, because Starcraft II is a proprietary, closed-source piece of software, we cannot use the exact algorithms used in SMAC. To fill this gap we chose the Optimal Reciprocal Collision Avoidance (ORCA) algorithm by Berg et al. [6].

The ORCA algorithm fills several criteria desirable for our environment. Much like in SC2, each unit is assumed to be a circle with a specific radius. Units can avoid other units, and they can also avoid static polygonal obstacles. On top of that, the units can not only avoid collisions but also move towards their own goal location at the same time.

The algorithm is parametrized by a time horizon τ. In each run of this algorithm, each unit A computes a set of half-planes (which we call *ORCA half-planes*) in 2D velocity space, each half-plane being the set of velocities safe to choose to avoid collision with some other unit or obstacle B for at least τ time.

Each unit A considers all units and obstacles in its immediate neighbourhood (i.e. within some radius r around it). For each neighbour B, the unit calculates the **velocity obstacle** induced by the neighbour on it – that is, the set of positions in velocity space that would make the unit collide (for units: get within the distance of $r_A + r_B$, where r_A and r_B are the units' own radii) with that neighbour within τ time.

Let $\mathbf{u}$ be the shortest vector from the relative velocity $\mathbf{v}_A - \mathbf{v}_B$ to the velocity obstacle's boundary – in other words, the smallest amount of change to the relative velocity required to prevent a collision within τ time. Then, the slope of the line (called the *ORCA line*) defining the ORCA half-plane for that neighbour is given by the slope of the outward normal of the velocity obstacle boundary at point $\mathbf{v}_A - \mathbf{v}_B + \mathbf{u}$. The line is anchored in a point given by $\mathbf{v}_A + \frac{1}{2}\mathbf{u}$. This gives the unit a half-plane of possible velocities that will avoid collision with the neighbour – note that the adjustment by $\frac{1}{2}\mathbf{u}$ is because the unit assumes the neighbour is following the same algorithm and will adjust by $-\frac{1}{2}\mathbf{u}$ (since from the neighbour's

perspective, everything is mirrored – hence the negation) – the adjustment is not halved for obstacle neighbours, only for unit neighbours.

Then, given all half-planes induced by neighbours, the algorithm solves a linear programming problem for each unit, to find a velocity that avoids all neighbours and is the closest to the unit's desired velocity. If avoiding collisions completely is not possible, the algorithm solves a different linear programming problem that minimizes the distance the unit crosses behind the ORCA lines.

We go into more detail about how we use this algorithm in SMAClite in Section 4. Note that this algorithm is not equivalent to a pathfinding algorithm – when faced with a wall, the units will often stop in front of it, and they will not look very far for a way to go around it.

4 SMACLITE

4.1 Environment implementation

In this section, we focus on our contribution and how we approached implementing SMAClite. Note that because of the closed-source nature of Starcraft II, we did not have access to any implementation details of it, algorithms contained in this section were designed with our knowledge of the game and with trial and error experiments, while also using some general information available on the Starcraft II Liquipedia [16].

We decided to implement the environment in the Python programming language [29] due to it being widely known in the machine learning community, where it is by far the most popular one. Most of the computations within the environment are performed using the Numpy library [12], and the rendering of the environment is handled by a script written by us using the Pygame [1] library. Wherever applicable, we used the default arguments of the SMAC environment and omitted any optional ones. The SMAClite environment uses the well-established OpenAI Gym framework [8] for creating reinforcement learning environments. We include visualizations of two scenarios within the environment in Figure 2[2].

At all steps of the implementation, we made sure the action, state, observation, and reward APIs are exactly aligned with SMAC. This also applies to individual unit attributes, for which we consulted Liquipedia [16]. This allowed us to conduct transfer learning experiments, such as the one we describe in Section 5.3.

Even though the map grid in Starcraft II allows triangles within the unitary squares, we chose for simplicity to only allow a grid of squares as the terrain for SMAClite. We found that this simplifying assumption does not detract from the environment's difficulty. Internally, we collapse adjacent squares containing obstacles into rectangles to lower the total number of obstacles for performance's sake. When defining the units' velocities or sizes, the base distance measurement unit is the side length of a single square in the terrain grid. This is consistent with SC2, and all scenarios available in both SMAC and SMAClite use a map size of 32 by 32 units.

To further improve environment performance, the units use K-D trees as available in the Scikit-learn library [21] to find their neighbours (e.g. finding units within sight range when generating observation vectors), as opposed to iterating over the entire unit list. Since K-D trees only support querying in a circular area and obstacles are always rectangles, when looking for obstacle neighbours, the units query an R-tree from the Python package `rtree` [11].

[2]We also make available videos showing SMAClite in action: https://drive.google.com/drive/folders/1-2YJicUqRzovTa7lRgJilwtecxCPrtVb

4.2 Base framework

SMAClite, much like SMAC, is defined mostly by the various combat scenarios available, as well as the units participating in those scenarios. As part of the SMAClite environment, we contribute a framework capable of reading both custom scenarios and units from JSON files – this means expertise in the Starcraft II map editor is no longer required to create new or modified challenges using the environment. This also means that with SMAClite it is easy to tell the difference between two different unit types – to compare "zergling" and "marine", all one needs to do is look at their respective JSON definitions, and see what the differences are.

All of the standard scenarios and units shipped with the environment are written using this framework. We give detailed specification of both the scenario and unit definition formats, as well as full examples of JSON files compatible with the framework, in the appendix to this paper.

4.3 Unit command types

At any given point in time, each unit in the environment is executing one of several types of commands. SMAClite supports five different command types, with the first four being exactly equivalent to the four action types available to the agents, as described in Section 3.2.

We introduce one more command type that is unavailable to agents but is key to the AI opponent's behaviour, called `attack_move` – this command orders the units to march toward a specified location, attacking any units encountered along the way, and then guard the location once it is reached. We based our implementation of attack-moving on the "Automatic targeting" article on the Starcraft II Liquipedia [16], but made a few judgement calls based on what yielded the most desirable behaviour, wherever their information was unclear or unavailable.

4.4 Environment loop

Each environment step starts with the agents' units being assigned commands corresponding to the actions chosen by the agents. Once that happens, we simulate eight game steps, and the reward returned from the environment step is the sum of the rewards earned within these game steps. Each game step is considered to last $\frac{1}{16}$th of a second for the purpose of calculating velocity, cooldowns, etc. – that does not mean this is its actual duration, as in fact, our environment can run much faster than real-time (see Section 5.4 for details).

Briefly, each game step consists of each unit following the logic associated with its command, e.g. moving, attacking, or waiting. We give a detailed description of each game step in the appendix.

4.5 Custom ORCA implementation

To facilitate installation simplicity, we rewrote the RVO2 library – the official implementation of ORCA in the C++ programming language from Berg et al. [6] – into NumPy, and we ship this module together with SMAClite. Because Starcraft II uses compiled (and,

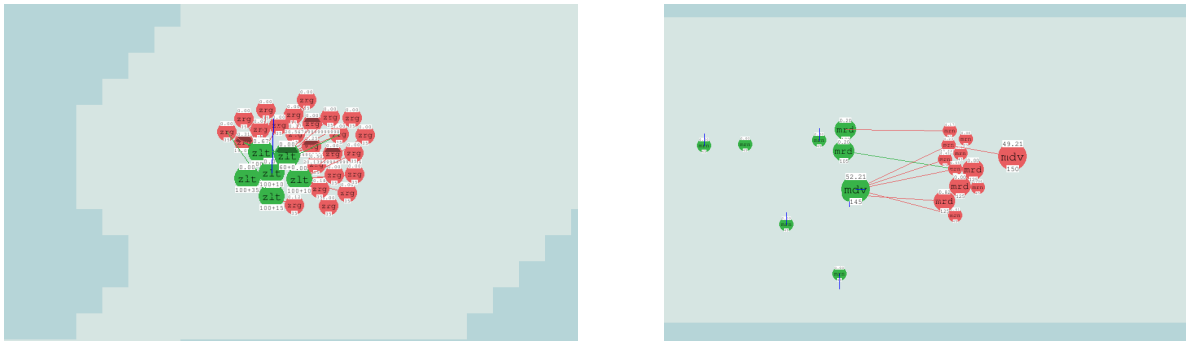

Figure 2: Visualizations of the SMAClite environment. *(left)* The `corridor` scenario. *(right)* The `MMM2` scenario.

we assume, highly optimized) C/C++ code, and Python code can be quite slow compared to it, we ran into performance issues early in the development of SMAClite. To address this issue, we also make available an addon for SMAClite[3], which uses the original C++ RVO2 library verbatim, using Python bindings written in the Cython [4] extension. The Python bindings were originally made available by Stüvel [26], though we implemented several modifications to suit our use-case – these modifications are also present in the Numpy version of RVO2.

The original RVO2 library has no way to remove units one by one or remove all units at once. These are key features for SMAClite, as we need to adjust the collision avoidance unit list whenever a unit dies or whenever we restart the environment. We added both of these features into RVO2 – we assume dead units disappear from the battlefield as soon as they are eliminated, so we do not want other units avoiding collisions with them.

The second set of adjustments considers static units – the original ORCA algorithm assumes fully cooperative units that will go out of their way to make the passage easier for other units. Our version, on the other hand, assumes that if a unit A is static (i.e. $||\mathbf{v}_A|| = 0$), it will *never* adjust its velocity. This makes it possible to surround other units and/or block their path, which is a valid strategy in Starcraft II. In the original implementation of ORCA, the blocking units would simply be "pushed" away. Note that to make up for this, any moving units will adjust their velocity by $\mathbf{u}$ instead of $\frac{1}{2}\mathbf{u}$ (see Section 3.3 for the definition of $\mathbf{u}$), when avoiding static units – this ensures moving units will not walk into static units.

Because the addon requires building and installing C++ files via `CMake`, which could potentially be problematic on some systems, we chose to make this RVO2 fork available as an optional dependency, for users who are willing to go through a more difficult installation process in order to boost environment performance. We call this version `SMAClite_plus`. Note that, because of differences in finding neighbours, unit behaviours will not be exactly the same between the two versions, but should remain functionally indistinguishable. If this addon is in use, custom K-D trees implemented in C++ for RVO2 are used for finding unit and obstacle neighbours for collision avoidance purposes, instead of Scikit-learn K-D trees or `rtree` R-trees. After it is installed, the addon can be enabled by setting the parameter `use_cpp_rvo2` to `True` when initialising the environment. Any attempt to set this argument to `True` without the addon installed will result in an error.

[3]https://github.com/uoe-agents/SMAClite-Python-RVO2

4.6 Opponent AI behaviour

The authors of SMAC [24] claim the opponent team is controlled by Starcraft II's built-in AI on the *very difficult* level. Because the enemy units' behaviours seemed very simple, we had our doubts about their strategic ability when watching combat inside the SMAC environment. For this reason, we performed the following test on the `MMM2`, `2c_vs_64zg`, and `corridor` SMAC scenarios.

First, we toggled the AI level inside SMAC across the various difficulty levels available, while keeping the random number generator seed constant. We then put the opponent AI against agents who pick randomly from the available actions. The opponent units' behaviour was always exactly the same, and we saw no difference at all in the rewards obtained by the random agents between difficulty levels – they were exactly equal to at least the tenth decimal place. Then, going one step further, we **removed** the AI opponent from the game, making the enemy units not controlled by any player. This **did not change the units' behaviour or the resulting rewards either**.

Based on these results, we are reasonably certain Starcraft's built-in AI never issues any orders to the enemy units in SMAC. The only order given to the units is hand-placed inside a script in each SMAC map file – it tells them to attack-move towards a specific point, usually where allied units initially appear. Following these results, we did not implement any custom opponent AI. All we do is, upon map initialization, set the enemy units' command to `attack_move` towards the `attack_point` specified in the map scenario file. The enemy units' command never changes throughout the encounter.

5 EXPERIMENTS

In this section, we describe the results of various experiments we performed in our environment. In all of the experiments, we use all of the scenarios used by Papoudakis et al. [20], with the addition of `bane_vs_bane`, which we included to feature a wider selection of unit types. Specifically, we used the `2sc_vs_1sc` (2 stalkers vs 1 spine crawler), `3s5z` (symmetric map with 3 stalkers and 5 zealots on each side), `MMM2` (a map with a medivac and some marines and marauders on both sides), `corridor` (6 zealots vs 24 zerglings in a narrow passageway), `3s_vs_5z` (3 stalkers vs 5 zealots),

and bane_vs_bane (a map with zeglings and banelings on both sides) scenarios. As optional material helpful to understand each scenario, we attach Table 3 in the appendix with natural-language descriptions of each scenario.

Through these experiments, we wish to show that our environment does indeed accomplish the goals we set for ourselves. First and foremost, we want to show that SMAClite poses a challenge equivalent to SMAC. We want to show that various MARL algorithms perform in it similarly well as in SMAC, and explain any discrepancies. We also want to prove that SMAClite is strictly cheaper to run than SMAC – the main metrics we are interested in are the time required to run it, and the RAM it takes up on the machine.

5.1 Agent learning curves

We trained several MARL algorithms on the selected scenarios. Each training was run for 4 million timesteps, and each training was repeated using 5 different random number generator seeds. All of the training were performed on the SMAClite_plus version of the environment and were performed solely using CPUs, each training using a single CPU core. The training were run on various nodes on a cloud cluster, most of them using Intel(R) Xeon(R) Gold 6138 CPUs @ 2.00GHz, and all others used CPUs of comparable computational capacity. Due to technical constraints, all training was under a strict 48-hour time limit. For all of the algorithms, we used hyperparameters listed as the best in SMAC as listed in the benchmark paper [20]. For the episodic algorithms, we used a buffer size of 5000, and for the parallel algorithms, we used a buffer size of 10. We used the EPyMARL framework [20] to run all of the training.

Some caveats need to be mentioned with regard to the training procedure. Firstly, the bane_vs_bane scenario is by far the slowest (as evidenced by Section 5.4), so we reduced the number of timesteps to 1 million for this scenario only. In addition, we found the MADDPG algorithm extremely slow during training, and so it never reached the required timesteps in two scenarios (MMM2 and bane_vs_bane) under the time limit, so we omit those two curves in our figure.

The resulting graphs of episodic return over training time can be found in Figure 3. In the remainder of this subsection, we compare the learning curves to those reported for SMAC in the benchmark paper.

First of all, many algorithms easily solve the 2s_vs_1sc scenario in SMAC, quickly reaching the maximum reward of 20, while in SMAClite the best performer (QMIX) only reaches an average return of about 15. We theorize that this is due to the fact that SMAClite does not simulate attack animations, and originally in Starcraft II the spine crawler has a very long attack animation (0.238 seconds versus 0.1193 seconds for the stalkers), so in SC2 it is much easier to dodge the spine crawler's attack in the last possible moment. Though the difference is less noticeable there, a similar situation occurs in 3s_vs_5z, and we believe it is for the same reason. Put simply, we believe *kiting* – alternating between running away and attacking without getting hit – is much more difficult in SMAClite than in SC2 because of the instant attacks.

Another major difference between the SMAC and SMAClite learning curves is in the MMM2 environment – in the case of SMAClite, three algorithms (MAPPO, VDN and QMIX) seem to have mastered the environment, with maximal rewards almost reaching 25 (for details about how the algorithms achieve this, see the next subsection), while in SMAC the best-performing algorithms barely reach a reward of 17.5 (but note that they are the same three algorithms, which shows the scenarios are still somewhat equivalent). Our hypothesis is that this is due to potentially unintended behaviour in the SMAC environment code. When calculating rewards, SMAC simply subtracts enemy units' new health values from their old health values, so any healing done by the enemies results in a negative reward being incurred by the agents, even though they never did anything wrong. We believe that this, combined with the fact that rewards should be gained for dealing damage, could confuse the agents during training. Judging by the wording in section 4 of the SMAC paper [24] we do not believe this is intentional, so the same behaviour is not present in SMAClite, where we explicitly define the reward as the sum of health points lost by the enemies due to attacks, and we do not penalize the agents for enemies healing.

Despite these few issues, we do observe that the overall shape of the training curves is comparable to those from SMAC in the benchmark paper [20] in all scenarios – some of them differ only in maximum return reached. The ranking of algorithms at the end of training time is also largely the same between SMAC and SMAClite. We believe that this is promising evidence pointing towards the environments being equivalent as far as learning is considered.

5.2 Evaluation of learned behaviours

In this section, we select one model for each scenario, and we describe the strategies employed by the agents to achieve high reward values, in order to demonstrate the complexity of strategies required in SMAClite. Note that the below descriptions are simply our interpretations of the behaviours demonstrated by the agents in the environment, informed by our knowledge of the game gained both by playing Starcraft II and working on SMAClite.

When making our selection, we wanted to showcase each of the algorithms, and each of them in a scenario where it did well – note that this means we might not use the best performer in all of the scenarios, but all of them are at least above average. However, because there are 9 algorithms and only 6 scenarios and because not all algorithms performed well, we decided to omit COMA, MADDPG, and IPPO, which were the overall weakest performers. In each case, we used the latest available (i.e. highest amount of training timesteps) checkpoint of agent parameters.

2s_vs_1sc – QMIX – **Mean test return: 16.22** – one of the stalkers is the "baiter" that gets the spine crawler's attention and they both attack until the baiter's health drops down to a very low value, at which point it backs out and lets the spine crawler target the other stalker. They then both hit the spine crawler until they die, which sometimes results in a victory and sometimes in defeat depending on the random attack order.

3s5z – VDN – **Mean test return: 20** – the stalkers and the zealots both focus on the enemy stalkers first, which deal more damage per second than the enemy zealots. The allied zealots move slightly north and the allied stalkers move slightly south, both to direct the

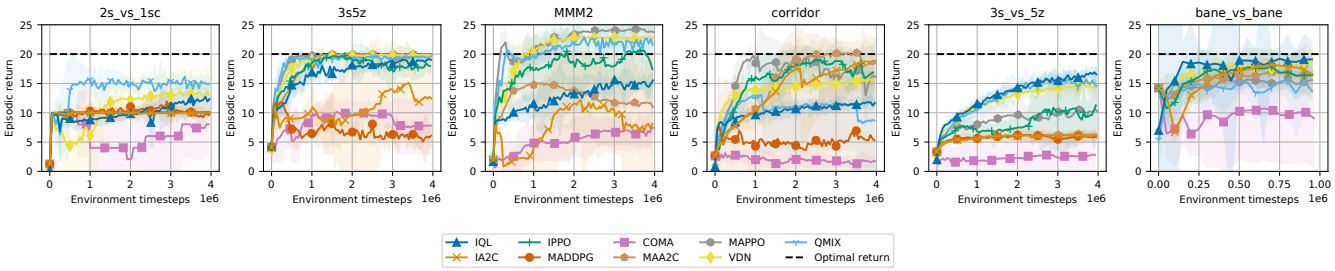

Figure 3: Test-time returns achieved by agents trained using the different algorithms over time during training. The graphs show the mean value, as well as the 95% confidence interval, from 5 trainings differing by random number generator seed.

attention of enemy stalkers (whose initial position is on the north side of the enemy army) to the allied zealots, who can take more hits, and also to allow allied stalkers plenty of room to maneuver while attacking. Also worth noting is the fact that the allied units are quite good at moving away right as their health drops to a dangerously low level, letting other nearby allies take the aggression.

`MMM2` – MAPPO – **Mean test return: 24.63** – all allied damage-dealers focus on the enemy marauders first, because they are the heavier hitters, while the ally medivac hides behind others as soon as it gets low since it is a priority target and would quickly be gunned down. What is interesting here is that the allied units leave the enemy medivac alive and kill it *last*, abusing the fact that enemy units getting healed results in a higher return due to the total damage dealt is higher – this is how the agents surpass the optimal return of 20 and reach values close to 25. They even stop hitting enemy damage-dealers and let them get healed, since the medivac cannot heal itself.

`corridor` – MAA2C – **Mean test return: 20.25** – the zealots fan out and form a horizontal line against the zerglings, blocking them making it harder for the zerglings to surround them. This causes the zerglings to crowd around the front of the zealot line, reducing the total amount of damage the zealots take over time.

`3s_vs_5z` – IQL – **Mean test return: 18.1** – the stalkers use the intended optimal strategy of kiting the zealots around the map. The lowest-health stalker always makes sure to stand behind the other two when attacking, in order to avoid dying. The stalkers do seem to get "lazy" with their kiting when the number of remaining zealots becomes low, probably because there is no penalty for dying, and it is easier to just stand and attack when the risk of death is low.

`bane_vs_bane` – IA2C – **Mean test return: 19.14** – the zerglings run away to the west to avoid the enemy banelings' explosions, while the ally banelings charge forward and explode when they become surrounded by enemy units. The enemy zerglings and banelings quickly die to multiple explosions. If any enemies remain after the allied banelings' explosions, the allied zerglings come out of hiding and attack.

5.3 Cross-environment zero-shot performance

In order to verify whether our environment does in fact require the same strategic reasoning as SMAC, we performed zero-shot transfer learning experiments on each scenario What we mean by this is, we trained the agents on SMAClite, and then without any retraining put them inside SMAC in the same scenario. We used the same models as in the previous experiment – to be specific, we first used their parameters from the first time the model was saved (at the very beginning of the training process), and then their parameters from the last time the model was saved (at the very end of the training process), and then compared the mean test-time returns. We present the results in Table 1.

We note that the mean return obtained has increased in *all* of the scenarios, with some exhibiting significant improvements such as doubling or tripling of the mean return. A noteworthy example is the `bane_vs_bane` scenario, where even the early version of the agents got close to the optimal return of 20, but was still improved upon by the later version. Because all of the scenarios exhibited improved returns upon training in SMAClite, we believe there is evidence to support the two environments requiring similar sets of skills, and that transfer learning from one to the other is a viable training strategy.

5.4 Environment Performance Benchmark

In this section, we consider the performance of the environments themselves, to confirm that SMAClite is indeed cheaper to run than SMAC. To obtain the data in this benchmark, we ran each scenario 20 times with agents picking randomly among the available actions. All of the below experiments were run on a computer with an AMD Ryzen 3700X CPU.

In Table 2 one can find the time each environment took per timestep (excluding logic not belonging to the environment, like action selection). We notice that the pure Python code is indeed slower than the original SMAC environment, but the environment becomes much faster when using the C++ RVO2 addon.

Note that these timings correspond to running a single environment step, which consists of 8 game steps in sequence, together with any environment-only logic (e.g. calculating rewards and determining observations). If we ever were to run experiments against human players (assuming SMAClite got some human control extension), we would want the environment to be capable of running in real-time. Both in SC2 and SMAClite, each game step, of which there are 8 in an environment step, is considered to last $\frac{1}{16}$ seconds, and most human players play SC2 at the "faster" in-game speed, which corresponds to a 40% speed-up [16]. Therefore, in order for the environments to run in real-time, they can afford to use $\frac{8}{16*1.4} \simeq 0.357$ seconds per environment step. Thus, all versions of the environment are more than capable of running all tested

Table 1: Mean test returns achieved by agents trained on SMAClite when put inside the original SMAC environment, achieved using the parameters from the first time they were saved during training, and from the last time they were saved during training. Late return on SMAClite is also included for reference.

Scenario	algorithm	early return	late return	late return on SMAClite
2s_vs_1sc	QMIX	0	11.4	16.22
3s5z	VDN	3.09	8.96	20
MMM2	MAPPO	1.87	6.70	24.63
corridor	MAA2C	3.35	4.48	20.25
3s_vs_5z	IQL	3.13	9.33	18.1
bane_vs_bane	IA2C	18.97	19.84	19.14

Table 2: Average seconds per timestep on the SMAC scenarios we used for training. Data was obtained by running the scenario 20 times against random agents.

Scenario	SMAC	SMAClite	Change	SMAClite_plus	Change
2s_vs_1sc	0.004	0.007	+75%	0.003	-25%
3s5z	0.013	0.018	+38%	0.007	-46%
MMM2	0.017	0.028	+64%	0.010	-41%
corridor	0.014	0.092	+557%	0.010	-28%
3s_vs_5z	0.006	0.013	+116%	0.005	-17%
bane_vs_bane	0.049	0.086	+76%	0.024	-51%

scenarios in real-time – the advantage of SMAClite_plus becomes the most obvious when running lengthy training which requires many executions of the environment.

While running the experiments described above, we also measured the amount of RAM used by each environment. This did not vary a lot by scenario and oscillated around 600 MB for the SMAC environment, and around 100 MB for the SMAClite and SMAClite_plus environments. Therefore, our lightweight version of the environment requires six times less memory than the original SMAC environment to run.

6 FUTURE WORK AND CONCLUSION

We presented SMAClite – a lightweight environment for MARL, consisting of a game engine emulating the Starcraft II minigame of SMAC, as well as a framework for easily creating new scenarios and units for this engine, using a familiar JSON format. We conducted experiments to show that SMAClite presents a challenge equivalent to SMAC, both by comparing learning curves and MARL algorithm rankings, as well as through a zero-shot learning experiment where training on SMAClite improved the agents' SMAC performance. We also showed that this challenge comes at a much-reduced cost, both in terms of required time and memory.

The SMAClite engine and framework are both very much open to extensions. Thanks to the fact that the environment is no longer bound by the Starcraft II dependency, developers could introduce game mechanics unrelated to Starcraft II, and could also go beyond the game's technical limitations. One could implement new unit types unseen in SC2 with new attack types or abilities, or easily create countless intricate puzzles for the agents to solve using the scenario framework.

It is also possible to treat SMAClite strictly as an extension of SMAC and work towards making it as close to SMAC as possible while maintaining the performance improvements it brings. This would likely mean focusing on transfer learning experiments such as the one in Section 5.3, and eliminating various differences present currently between the environments. One example might be a projectile/animation simulation, which is one of the major differences between the SMAC and SMAClite, and caused discrepancies in 2s_vs_1sc, as well as in 3s_vs_5z, during our experiments.

One noteworthy contribution to the SMAC ecosystem is SMAC v2 [9], which introduces procedurally generated scenarios that change episode-to-episode, and show that this stochasticity makes for more challenging scenarios and forces the agents' strategy to be more adaptable. This is certainly in interesting direction, and we will look into implementing similar improvements to SMAClite in the nearest future.

One could also work on improving the modified ORCA algorithm's combat capabilities – as evidenced by the corridor scenario, the SC2 zerglings were much better at surrounding the zealot wall as performed by our MAA2C agents than the SMAClite zerglings. This probably stems from the fact that RVO2 is mostly a general-purpose collision avoidance algorithm, while SC2's algorithm was handwritten for the best combat performance possible. Another feature idea we could use from SC2 is a pathfinding algorithm, since running in a straight line becomes a problem very soon when the terrain becomes any more complicated than the terrain in our experimental scenarios.

REFERENCES

[1] [n.d.]. Pygame. https://github.com/pygame/pygame

[2] Stefano V. Albrecht, Cillian Brewitt, John Wilhelm, Balint Gyevnar, Francisco Eiras, Mihai Dobre, and Subramanian Ramamoorthy. 2021. Interpretable Goal-based Prediction and Planning for Autonomous Driving. In *IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, May 30 - June 5, 2021*. IEEE, 1043–1049. https://doi.org/10.1109/ICRA48506.2021.9560849

[3] Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. 2020. The Hanabi challenge: A new frontier for AI research. *Artif. Intell.* 280 (2020), 103216. https://doi.org/10.1016/j.artint.2019.103216

[4] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39. Publisher: IEEE.

[5] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *J. Artif. Intell. Res.* 47 (2013), 253–279. https://doi.org/10.1613/jair.3912

[6] Jur van den Berg, Stephen J. Guy, Ming C. Lin, and Dinesh Manocha. 2009. Reciprocal \emphn-Body Collision Avoidance. In *Robotics Research - The 14th International Symposium, ISRR 2009, August 31 - September 3, 2009, Lucerne, Switzerland (Springer Tracts in Advanced Robotics, Vol. 70)*, Cédric Pradalier, Roland Siegwart, and Gerhard Hirzinger (Eds.). Springer, 3–19. https://doi.org/10.1007/978-3-642-19457-3_1

[7] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *CoRR* abs/1912.06680 (2019). http://arxiv.org/abs/1912.06680 arXiv: 1912.06680.

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[9] Benjamin Ellis, Skander Moalla, Mikayel Samvelyan, Mingfei Sun, Anuj Mahajan, Jakob Nicolaus Foerster, and Shimon Whiteson. 2022. SMACv2: A New Benchmark for Cooperative Multi-Agent Reinforcement Learning. (June 2022). https://openreview.net/forum?id=pcBnes02t3u

[10] Jakob N. Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. 2018. Counterfactual Multi-Agent Policy Gradients. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 2974–2982. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17193

[11] Gillies, Sean, Butler, Howard, Pedersen, Brent, Matthias, and Adam Stewart. [n.d.]. Rtree: Spatial indexing for Python. https://github.com/Toblerity/rtree

[12] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[13] Peter Hart, Nils Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. https://doi.org/10.1109/tssc.1968.300136 Publisher: Institute of Electrical and Electronics Engineers (IEEE).

[14] Mingyu Kim, Jihwan Oh, Yongsik Lee, Joonkee Kim, Seonghwan Kim, Song Chong, and Se-Young Yun. 2022. The StarCraft Multi-Agent Challenges+ : Learning of Multi-Stage Tasks and Environmental Factors without Precise Reward Functions. *CoRR* abs/2207.02007 (2022). https://doi.org/10.48550/arXiv.2207.02007 arXiv: 2207.02007.

[15] Aleksandar Krnjaic, Jonathan D. Thomas, Georgios Papoudakis, Lukas Schäfer, Peter Börsting, and Stefano V. Albrecht. 2022. Scalable Multi-Agent Reinforcement Learning for Warehouse Logistics with Robotic and Human Co-Workers. _eprint: 2212.11498.

[16] Liquipedia. [n.d.]. Starcraft II Liquipedia. https://liquipedia.net/starcraft2/Main_Page

[17] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 6379–6390. https://proceedings.neurips.cc/paper/2017/hash/68a9750337a418a86fe06c1991a1d64c-Abstract.html

[18] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 1928–1937. http://proceedings.mlr.press/v48/mniha16.html

[19] Igor Mordatch and Pieter Abbeel. 2018. Emergence of Grounded Compositional Language in Multi-Agent Populations. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 1495–1502. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17007

[20] Georgios Papoudakis, Filippos Christianos, Lukas Schäfer, and Stefano V. Albrecht. 2021. Benchmarking Multi-Agent Deep Reinforcement Learning Algorithms in Cooperative Tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a8baa56554f96369ab93e4f3bb068c22-Abstract-round1.html

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[22] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. 2018. Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research. *CoRR* abs/1802.09464 (2018). http://arxiv.org/abs/1802.09464 arXiv: 1802.09464.

[23] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. 2018. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning*. PMLR, 4295–4304.

[24] Mikayel Samvelyan, Tabish Rashid, Christian Schröder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philip H. S. Torr, Jakob N. Foerster, and Shimon Whiteson. 2019. The StarCraft Multi-Agent Challenge. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019*, Edith Elkind, Manuela Veloso, Noa Agmon, and Matthew E. Taylor (Eds.). International Foundation for Autonomous Agents and Multiagent Systems, 2186–2188. http://dl.acm.org/citation.cfm?id=3332052

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). http://arxiv.org/abs/1707.06347 arXiv: 1707.06347.

[26] Sybren A. Stüvel. [n.d.]. Python bindings for Optimal Reciprocal Collision Avoidance. https://github.com/sybrenstuvel/Python-RVO2

[27] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinícius Flores Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. 2018. Value-Decomposition Networks For Cooperative Multi-Agent Learning Based On Team Reward. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, Elisabeth André, Sven Koenig, Mehdi Dastani, and Gita Sukthankar (Eds.). International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2085–2087. http://dl.acm.org/citation.cfm?id=3238080

[28] Ming Tan. 1993. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*. 330–337.

[29] Guido Van Rossum and Fred L Drake Jr. 1995. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam.

[30] Florian Vaussard, Julia Fink, Valerie Bauwens, Philippe Rétornaz, David Hamel, Pierre Dillenbourg, and Francesco Mondada. 2014. Lessons learned from robotic vacuum cleaners entering the home ecosystem. *Robotics Auton. Syst.* 62, 3 (2014), 376–391. https://doi.org/10.1016/j.robot.2013.09.014

[31] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. 2019. Grandmaster level in StarCraft

II using multi-agent reinforcement learning. *Nature* 575, 7782 (Nov. 2019), 350–354. https://doi.org/10.1038/s41586-019-1724-z Number: 7782 Publisher: Nature Publishing Group.

[32] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John P. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. 2017. StarCraft II: A New Challenge for Reinforcement Learning. *CoRR* abs/1708.04782 (2017). http://arxiv.org/abs/1708.04782 arXiv: 1708.04782.

[33] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. 2021. The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games. *arXiv preprint arXiv:2103.01955* (2021).

Appendix A SMACLITE FRAMEWORK DETAILS

A.1 Scenario definition

Our framework accepts scenario JSON files containing a single JSON object. Each scenario should have a `name` to identify it, and should specify the numbers of allied and enemy units with the `num_allied_units` and `num_enemy_units` parameters. The units in the scenario should be listed with the `groups` parameter, each group being a JSON object with `x` and `y` parameters specifying the group's center, a `faction` parameter (`ALLY` or `ENEMY`) to specify which team the units are on, and a `units` parameter with an object of unit types together with their counts. Each group will initially be laid out in the shape of a square around their specified location. Note that these groups have no impact beyond unit positioning – once initial unit placement is complete, neither the agents nor the units have any information about what group they came from.

The framework supports two ways of specifying unit types. In order to use a standard unit type, its uppcercase name should be used (e.g. `ZERGLING`). The other way to specify a unit type is to provide a path to a JSON file with its specification. The framework also supports an optional `custom_unit_path` parameter – if specified, this path will be prepended to all unit types specified as a path. Note that if the `.json` extension is missing the framework reattaches it automatically, so if the custom unit path is `path/to` and the unit type specification is `type`, the framework will look for the file `custom/unit/type.json` file. The framework also requires an `attack_point`, which is typically near to the initial positions of the allied units – this point is where the enemy units will be marching towards throughout the scenario (see Section 4.6 for details).

The framework also supports two ways of defining terrain – one way is to pick a standard terrain present by supplying the `terrain_preset` parameter with its uppercase name (e.g. `CORRIDOR`). Terrain can also be provided in the scenario definition file itself using the `terrain` argument, which should consist of a list of strings forming a rectangular 2D array. The framework supports two types of terrain: `_` for walkable, and `X` for non-walkable. The framework also always requires a `width` and `height` to be specified for the scenario, which should match the terrain dimensions – note that all scenarios adapted from SMAC have the dimensions of 32 by 32.

Finally, the framework requires some general details about units participating in the scenario. The parameters `ally_has_shields` and `enemy_has_shields` specify which teams, if any, should have shields active on them, and the parameters `num_unit_types` and `unit_type_ids` specify what IDs the agents will receive in observations for the various unit types. The former should be a number (note that scenarios adapted from SMAC use 0 for all scenarios where both teams are homogenous, i.e. only have one unit type each), and the latter should be a map from unit type specifications (same as above) to numbers from 0 to `num_unit_types` minus one. The length of the map must be equal to `num_unit_types`.

We ship several standard scenarios with this environment as separate OpenAI Gym environments (e.g. `smaclite/2s3z-v0`). To use a custom scenario file, one should use the `smaclite/custom-v0` Gym environment, and provide a path to the scenario file via the `map_file` parameter.

A.2 Unit definition

Each unit is assigned a specific **type**, with a set of different attributes impacting the environment mechanics. Our framework supports several attributes in the JSON files defining the various unit types, in order to allow for their easy customization.

Firstly, the framework supports several attributes defining the units' resources, including their maximum health and health regeneration via the `hp` and `hp_regen` attributes, respectively. Their shields, if any, via the `shield` attribute, and their maximum and initial energy via the `energy` and `initial_energy` attributes. The units' size can be specified by providing their diameter in the `size` attribute, and their speed (in distance per second) via the `speed` parameter.

Then, the framework supports several attributes defining the units' combat abilities. Their `combat_type` (always one of `DAMAGE` or `HEALING`) defines their main role on the battlefield, and their `damage` defines how strong of a hitter they are, while their `armor` defines their defensive capabilities. Each unit has an `attack_range`, which defines how close (measured boundary to boundary) the unit has to get to a target in order to attack or heal it[4]. Note that while a numeric value is usually expected for this attribute, the special value of `MELEE` is also accepted to signify that the unit should have the standard melee range, i.e. only attack from up close. Some units can deal damage multiple times per attack – this can be achieved using the `attacks` attribute (e.g. 2 for attacking twice at once), and after attacking, each unit has to wait their specific `cooldown` before attacking again. Each unit type should also define a `minimum_scan_range`, which will govern how far the units will look when searching for targets.

In SMAClite, each unit resides in a specific plane, with three currently supported: `GROUND`, `AIR`, and `COLOSSUS`. When moving, the units only avoid collisions with units in the same plane as them, and only ground units are affected by static obstacles. All of the units in SMAClite can only target units which reside in the planes listed in their `valid_targets` – for example, if `AIR` is not in this list for some unit, then it can never attack airborne units. But note that, regardless of their `valid_targets`, all units can target units int he `COLOSSUS` plane.

In order to support making certain unit types stronger against specific other types, the framework also supports a system of unit `attributes` (e.g. `BIOLOGICAL`) and attribute-relative `bonuses`, defined as a map from attribute to bonus value. For example, if a unit type has a bonus of 20 against `ARMORED` units, it will deal 20 bonus damage with each attack against units with that attribute.

Finally, the framework supports several different attack types for units, defined by the `targeter` and `targeter_kwargs` attributes. The most common attack type is `STANDARD`, which simply has the unit attacking one target at a time. The other attack types are only used by one standard unit each, but can easily be reused for custom unit types. The `KAMIKAZE` attack type has the unit explode when attacking, dealing damage in a circle with a specified `radius` around itself and dying the process. The `LASER_BEAM` type fires a laser in a line perpendicular to the line between the attacker and their target –

[4]Note that this is different from the agent targeting range mentioned in Section 3.2 – that range only affects the agents' observations, while this one actually governs when an attack can happen.

Table 3: Combat scenarios used in our experiments.

Name	Allied Units	Enemy Units	Description
2s_vs_1sc	2	1	Two stalkers – powerful but fragile ranged units – face off against one spine crawler – very strong unit with no movement capabilities. The stalkers need to abuse the spine crawler's immobility to bring its health down.
3s5z	8	8	A symmetrical map with each team having three stalkers and five zealots – melee units that deal less damage but can take a lot of hits before dying.
MMM2	10	12	Each team has some marines – fragile ranged units that attack quickly, some marauders – more durable units that have stronger attacks, but can't hit flying units, and one medivac – a flying healer unit.
corridor	6	24	The allied team only has 6 zealots to hold off 24 zeglings – extremely fragile but very quick units with potential to overwhelm unprepared enemies with numbers and surround them.
3s_vs_5z	3	5	Three stalkers need to keep the quick zealots at bay while outnumbered, using their range to their advantage.
bane_vs_bane	24	24	Each team has some zerglings and some explosive kamikaze banelings, each of which can easily take out several zegrlings with one explosion.

more specifically, the laser line is a rectangle with a specified `width` and `height`. Lastly, the `HEAL` attack type is used by healer units.

Appendix B SMACLITE FRAMEWORK EXAMPLES

B.1 Example of a valid scenario file

The following is an example custom scenario similar to the built-in scenario `10m_vs_11m`, but using custom units.

```
{
    "name": "10m_vs_11m",
   "custom_unit_path": "smaclite/env/units/smaclite_units",
    "num_allied_units": 10,
    "num_enemy_units": 11,
    "groups": [
        {
            "x": 9,
            "y": 16,
            "faction": "ALLY",
            "units": {
                "example_custom_unit": 10
            }
        },
        {
            "x": 23,
            "y": 16,
            "faction": "ENEMY",
            "units": {
                "example_custom_unit": 11
            }
        }
    ],
    "attack_point": [9, 16],
    "terrain_preset": "NARROW",
    "num_unit_types": 0,
    "ally_has_shields": false,
    "enemy_has_shields": false
}
```

B.2 Exmaple of a valid unit file

The following is a custom unit similar to the built-in `MARINE` unit, but with a much larger (effectively global) scan range

```
{
    "hp": 45,
    "armor": 0,
    "damage": 6,
    "cooldown": 3,
    "speed": 3.15,
    "attack_range": 3,
    "size": 3,
    "attributes": ["LIGHT", "BIOLOGICAL"],
    "minimum_scan_range": 100,
    "valid_targets": ["GROUND", "AIR"]
}
```

Appendix C DETAILS ON THE SMACLITE GAME LOOP

In this section we describe in detail the logic of each game step, of which there are several within each environment step. Each game step consists of several phases. These phases were developed by us and we have no information about whether the game steps inside Starcraft II follow any sort of similar order – this structure was simply what yielded the most reasonable game ruleset that is resembles Starcraft II. Before the next phase can begin, all units must execute the logic for the previous phase. This is necessary to ensure the units' perceptions of other units (note: unrelated to the agents' observations) remain consistent throughout the execution of the game step.

C.1 Target clean-up

First of all, each unit might lose the target it was attacking or healing in the previous game-step, according to command-specific logic. It is necessary for all units to execute this logic before proceeding with declaring their preferred velocity for this game step because other units might access the target information when computing their own preferred velocity.

Units with the `noop`, `stop`, and `move` commands immediately lose any target they had, since these are non-combat commands. Units with the `target` command acquire the target dictated by their command, or retain it they are already targeting it. With the `attack_move` command, the units consider several factors. They always lose a dead target and they never lose a target who attacked them in the last game step. Otherwise, they lose their target if it is outside of their attack range.

C.2 Velocity preparation

In this phase all units declare their preferred velocity for this game step. It is necessary that they all do this before proceeding with velocity adjustment via the ORCA algorithm, because units can perceive each other's preferred velocity and make collision avoidance decisions based on that information.

In the case of `noop` and `stop` commands, the preferred velocity of the units is always $\mathbf{0}$. In the case of the `move` command, if the unit's maximum velocity is v_{max}, its current position is $\mathbf{x}$, and the position where it wants to move is $\mathbf{y}$, where $\mathbf{x} \neq \mathbf{y}$, then the unit always declares that its preferred velocity is $(\mathbf{y}-\mathbf{x})\frac{v_{max}}{||\mathbf{y}-\mathbf{x}||}$. If $\mathbf{x} = \mathbf{y}$, then the preferred velocity is instead $\mathbf{0}$. Put simply, the unit always wishes to move in a straight line towards its destination – this is different to Starcraft II's engine which has a built-in pathfinding algorithm based on the A* algorithm [13] – but we decided to omit it, since it only matters in one of the many scenarios offered by SMAC (`2c_vs_64zg`).

If the unit's command is `target`, the unit considers its distance from the target unit. Let the unit's attack range be d_{max}, its radius r_A, and its target radius r_B, and let the distance between it and the target be defined as $d(A, B)$. Then, in the case where $d(A, B) > r_A + d_{max} + r_B$, the unit declares it is moving in a straight line towards its target, and proceeds as with a `move` command. Otherwise, it declares it is attacking or healing, and its preferred velocity is $\mathbf{0}$.

Finally, if the unit's command is `attack_move`, the unit first finds the valid targets within its scan range. It is important to note that if the unit is a damage-dealer, it considers any enemy healers *priority targets*, and all other enemy units as non-priority targets – if the unit still has a target after the target clean-up phase, the only way it will switch targets at this point is if its current target is not a priority target, and there is a priority target within its scan range. If it does not have a target, or needs to switch, it picks the closest target with the highest available priority among the valid targets. The process is slightly different with healers, who consider valid targets any non-healer units in their team who are below full health, or who are attacking another unit. The healer unit picks as its target the lowest-health unit among its valid targets. After the target selection process finishes, the unit proceeds as with a `target` command if it has a target, and as with a `move` command toward its attack-move destination position if it does not. Note that healers will adjust their maximum velocity for any given game step to match the slowest allied unit within their scan range – we implemented this behaviour in order to stop them getting in front of their army if they are the fastest units.

There are a few intentional changes from SC2 in this phase – therein, it is not true that healers can target any non-healers, and can never target other healers or themselves. This is only true with the limited set of units present in SMAC, and we felt it makes for a nice simplification of the SC2 ruleset that does not change anything in SMAC. It is also not true that damage-dealers consider any healers as priority targets in SC2 – they only do in SMAC due to the modifications made to the map files by its authors – again, we felt this is a nice simplification aligned with SMAC, but not with SC2.

C.3 Velocity adjustment

Each unit A uses the ORCA algorithm to determine its actual velocity that avoids collisions, taking into account its own preferred velocity, as well as its neighbour units and obstacles. For the purpose of collision avoidance, the unit considers its neighbours all units within the radius of $(r_A + r_{max})\tau$ of itself, where r_{max} is the maximum radius of any unit in the scenario and τ is the time horizon; and all obstacles within the radius of $r_A + \tau v_{A,max}$ of itself. In SMAClite, we always use $\tau = 1$, i.e. the units want to guarantee avoiding collisions for 1 second, or 16 game steps. The ORCA algorithm returns for each unit its actual velocity, given its preferred velocity. For details on our implementation of the ORCA algorithm, refer to Section 4.5.

C.4 Game step execution

This is the final phase of the game step, and because order matters here (a unit might be eliminated before it gets to execute its command), the units execute their respective logic for this phase *in random order*, controlled by the environment randomness seed, with the order being re-randomized in each game step. Note that we do not have any data about whether Starcraft II also randomizes game step execution order, this simply felt like a decent compromise between fairness and simplicity.

Each unit, regardless of its command type, begins this phase by performing several standard updates. It updates its position according to its actual velocity computed in the previous phase (i.e. $\mathbf{x}_{new} = \mathbf{x} + \frac{\mathbf{v}_{actual}}{16}$, since velocity is always defined in distance per second), it reduces its cooldown if applicable (i.e. if it is greater than 0, it gets reduced by $\frac{1}{16}$th of a second), and it regenerates health, energy, and/or shields, if applicable. It then proceeds with command-specific logic.

The only case where any command-specific logic is performed at this point is if the unit's command is `target`, or `attack_move` with a non-empty target, and the unit declared during the velocity preparation phase that it is attacking or healing, and its current cooldown is 0 – in all other cases the unit's game step logic ends here. In the single actionable case, it attacks or heals its target.

Damage from attacks is dealt first to the enemy shields, then to the enemy health, with any damage dealt to the target's health being reduced by its armor. If the unit's `attacks` attribute is greater than 1, then it deals damage multiple times in a row during this

step. If the unit did attack, its cooldown is then set to its maximum value, defined by the unit's type. Healing units heal their target at a rate of 9 health per second, spending $\frac{1}{3}$ of an energy point per health healed.

Do note that units might attack or heal units that are technically outside of their range at this point, because attack/heal declaration happened during the velocity preparation phase, and the target might have moved away slightly since then. This is intentional and is meant to prevent endless chases when the units' velocities are similar. Again, we do not have data on whether the Starcraft II engine features any similar simplifications, but this rule works well for SMAClite.

In addition, there is an intentional change from SC2 in here as well, because for simplicity we do not simulate attack animations or attack projectiles – every attack happens instantly in the timestep when the unit declares it is attacking. This causes some small discrepancies from SC2, but we believe the margin of error is acceptable.