Algorithms or Actions? A Study in Large-Scale Reinforcement Learning

Anderson Rocha Tavares[*1], **Sivasubramanian Anbalagan**[*2],
Leandro Soriano Marcolino[2], **Luiz Chaimowicz**[1]
[1] Computer Science Department – Universidade Federal de Minas Gerais
[2] School of Computing and Communications – Lancaster University
anderson@dcc.ufmg.br, siva.anbalagan@gmail.com,
l.marcolino@lancaster.ac.uk, chaimo@dcc.ufmg.br

Abstract

Large state and action spaces are very challenging to reinforcement learning. However, in many domains there is a set of algorithms available, which estimate the best action given a state. Hence, agents can either directly learn a performance-maximizing mapping from states to actions, or from states to algorithms. We investigate several aspects of this dilemma, showing sufficient conditions for learning over algorithms to outperform over actions for a finite number of training iterations. We present synthetic experiments to further study such systems. Finally, we propose a function approximation approach, demonstrating the effectiveness of learning over algorithms in real-time strategy games.

1 Introduction

Reinforcement learning aims at developing general agents, which learn by acting directly on the problem action space [Sutton and Barto, 1998]. However, as the state and action spaces grow large, learning agents struggle to attain high performance. On the other hand, many domains have existing algorithms, tailored to the specific problem, and an agent could rely on a pool of algorithms to act on its behalf [Rice, 1976].

Given limited computational resources, however, there is an important conflict: should we *learn over actions*, training a reinforcement learning agent to discover the best actions to take, or should we *learn over algorithms*, trying to discover the best algorithm to estimate the best action in each state?

Previous work on reinforcement learning with abstract actions [Sutton *et al.*, 1999; Dietterich, 2000] have shown that the optimal policy may not be attainable when learning over algorithms, although it may accelerate the reinforcement learning process. However, it is still unclear when each method should be preferred. Additionally, having a pool of algorithms may still not enable one to directly apply reinforcement learning techniques when the state space is also very large. In particular, Real-Time Strategy Games are a major challenge for Artificial Intelligence research, given their enormous action and state spaces [Ontañón *et al.*, 2013].

In this work we establish the conditions where learning over algorithms is helpful, evaluating the sufficient strength of available algorithms, the relation among algorithms and actions set sizes, and possible underlying algorithm creation processes. Synthetic experiments further develop our conclusions. Additionally, we introduce a function approximation approach for Real-Time Strategy Games, demonstrating the effectiveness of learning over algorithms in a complex domain.

2 Related Work

In algorithm selection, one finds a performance-maximizing mapping from problem instances to algorithms [Rice, 1976]. It has been applied to a variety of problems, including SAT [Xu *et al.*, 2008], sorting [Lagoudakis and Littman, 2000], and general video game playing [Bontrager *et al.*, 2016].

Learning over algorithms is also related to action abstractions in reinforcement learning (RL). In MAXQ [Dietterich, 2000], a MDP "calls" other sub-MDPs organized in a graph. Options on MDPs [Sutton *et al.*, 1999] are temporally-extended actions. Algorithms can be seen as one-step options: they can initiate in any state, act according to their internal policy and terminate after one transition. While Sutton *et al.* [1999] showed that the optimal policy is not attainable when the RL agent has no access to actions, they did not present a detailed study on the dilemma between learning over actions or over algorithms/options. Our theory provides new insights in terms of the required strength of algorithms, relations among algorithms and actions set sizes, and underlying algorithm creation processes.

The use of algorithms, or scripts, is widely adopted in real-time strategy games (RTS) research. For example, in script selection via Monte Carlo planning [Sailer *et al.*, 2007] or assignment of scripts directly to units [Churchill and Buro, 2013] or unit types [Lelis, 2017] via hill-climbing. However, these works are combat-oriented, and do not tackle RTS games as a whole. Recent planning-based approaches, on the other hand, address the full games: AHTN [Ontañón and Buro, 2015] combines hierarchical-task network (HTN) planning, with a minimax-like tree-search algorithm. PuppetSearch combines scripted behavior with game-tree search, by letting the scripts expose a restricted set of actions for the search algorithms to investigate. Two variations exist: PuppetAB [Barriga *et al.*,

* A. R. Tavares and S. Anbalagan are both first authors.

2015] and PuppetMCTS [Barriga *et al.*, 2018], which uses α-β pruning and Monte Carlo Tree Search as their search algorithms, respectively. StrategyTactics [Barriga *et al.*, 2017] uses a convolutional neural network to predict the output of PuppetSearch, allowing more time to be used by a tactical search algorithm. NaiveMCTS [Ontañón, 2017] also employs a Monte Carlo Tree Search, but uses a sampling strategy based on combinatorial multi-armed bandits (and thus does not use scripts). We test against all these approaches in our experiments.

All foregoing approaches require a forward model of the world to perform searches, which is not available in commercial RTS games and some real-world scenarios. Tavares *et al.* [2016] present a model-free approach that estimates algorithms' performance by running matches offline. However, a fixed algorithm must be chosen to play an entire match.

Our reinforcement learning approach with function approximation for algorithm selection in RTS games combines many strengths of related work: it tackles the game as a whole (not only combats), dynamically selects algorithms, and dismisses forward models.

3 Learning over Algorithms

We consider reinforcement learning (RL) tasks specified via Markov Decision Process (MDP), which are defined by a set of states $\mathbf{S}$, a set of actions $\mathbf{A}$, a reward function $\mathbf{R} : \mathbf{S} \times \mathbf{A} \to \mathbb{R}$, and a state transition function $\mathbf{T} : \mathbf{S} \times \mathbf{A} \times \mathbf{S} \to [0, 1]$. The aim in reinforcement learning is to discover the optimal action-value function $Q^*(s, a)$, which indicates the value of taking action a in state s and following the optimal policy thereafter. Q^* maximizes the expected sum of discounted rewards $E[\sum_{j=0}^{\infty} \gamma^j r_{t+j}]$, where t is the current time and r_{t+j} is the reward received j steps in the future. The discount factor $\gamma \in [0, 1]$ specifies how much the agent considers future rewards. RL agents also balance exploration and exploitation. As usual, we consider that when exploring (e.g., with ϵ probability) the agent chooses an action uniformly at random. When exploiting, the agent selects the action that maximizes $Q^*(s, a)$, with ties broken randomly.

Learning over actions is difficult in MDPs with large action sets, often requiring an impractical number of interactions with the environment to learn useful action-values. Thus, an agent may resort to existing algorithms, which could incorporate heuristics, search-based approaches, and/or domain knowledge, to act on its behalf. Formally, at each state, the agent selects an algorithm x from a set of algorithms $\mathbf{X}$, which then selects an action $a \in \mathbf{A}$ to affect the environment. The agent observes r and s', and chooses a new $x \in \mathbf{X}$ to act in this new state. Algorithms' performance may vary across different states, and thus it is necessary to learn which x to use at each state. We can apply usual RL methods, but an "action" corresponds to choosing an algorithm, and the algorithm outputs an action for the current state.

As in Marcolino *et al.* [2013], and Marcolino *et al.* [2014], we model each algorithm $x \in \mathbf{X}$ as a probability distribution function (pdf) over $\mathbf{A}$. That is, algorithms do not know in advance the action-values, and output the action that they estimate to be the best, according to their own decision procedures.

Algorithm / Action	a_1	a_2	a_3	a_4
x_1	0.8	0.2	0	0
x_2	0.2	0	0.7	0.1

Table 1: Algorithms' pdfs used in our simple example.

Given a state, there is a certain probability that the algorithm outputs the true best action, and a certain probability for other actions. Let p_a^x be the probability of x selecting an action a. Although we use the pdfs in our analysis, in general they may be unknown. Our analysis allows a deeper understanding of the conflict between learning over algorithms or actions, but a designer may still need to estimate the pdfs when taking a decision between both approaches. There are examples of estimating algorithms' pdfs in the literature [Marcolino *et al.*, 2013; 2014; Jiang *et al.*, 2014]. Our theoretical analysis is done by comparing the likelihood of selecting the (unknown) optimal action a^*, which maximizes the expected sum of discounted rewards. We consider two RL agents, P_1 and P_2, which reason over actions and algorithms, respectively.

Simple Example

Consider a single state, four actions $\{a_1, a_2, a_3, a_4\}$ and two available algorithms $\{x_1, x_2\}$. We assume a_1 is the optimal action, but that is not known in advance, and the algorithms select each action according to the probabilities shown in Table 1, which results from their reasoning procedures.

In the first iteration, P_1 picks a_1 with probability 0.25. P_2 picks x_1 with probability 0.5, which selects a_1 with probability 0.8. Hence, P_2 selects a_1 indirectly with probability at least $0.5 \cdot 0.8 = 0.4 > 0.25$. Thus P_2's expected performance is better than P_1's. In the next few iterations P_2 is even more likely to pick x_1, while P_1 may still need to explore further, until finally playing enough training iterations for the action-value of a_1 to be higher than the other actions.

When the number of training iterations becomes sufficiently large, P_1 learns to always select a_1, and P_2 to always select x_1. However, since x_1 selects a_1 with probability 0.8, it turns out that P_2 selects a_1 with probability $0.8 < 1$, and hence is outperformed by P_1 in the long run. Therefore, until a certain number of training iterations, a RL agent may perform better by learning over algorithms, depending on the algorithms' pdfs. In the long run, however, learning over actions will always perform better. We formalize this notion below.

3.1 Theoretical Analysis

Our main result is a sufficient condition for learning over algorithms to outperform learning over actions. That allows a formal guarantee when learning over algorithms, besides guiding in the number of algorithms used, as we discuss next.

P_1 or P_2 selects the best choice (best action a^* or best algorithm x^*, respectively) with a certain probability p_i. Let l be the current training iteration. As usual, we consider RL agents where $l \to \infty \Rightarrow p_i \to 1$. We model p_i by a learning curve given by the function $1 - (\xi_i + e^{l \times \beta_i})^{-1}$, where ξ_i and β_i are parameters defining the initial error and the convergence "speed", respectively. A training process is noisy by nature, but these functions model the average behavior over many training

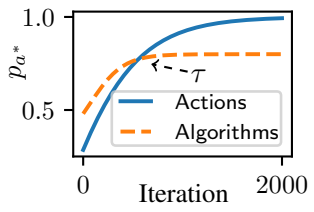


Figure 1: Theoretical learning curves.

events. They converge to 1 in a diminishing returns fashion, as it would be expected in a training process. We then have:

Theorem 1. *Let $\mathcal{P}_l^i$ be the probability that P_i picks a^* at iteration l. $\mathcal{P}_l^2 > \mathcal{P}_l^1$ for a finite number of iterations, if $\exists$ $x \in \mathbf{X}$, where $p_{a^*}^x > \frac{|\mathbf{X}|}{|\mathbf{A}|}$. If $l \to \infty$, however, $\mathcal{P}_l^1 \geq \mathcal{P}_l^2$.*

Proof. If P_2 selects x^*, it indirectly selects a^* with probability $p_{a^*}^{x^*}$. Hence, at iteration l it selects a^* with probability at least $(1 - (\xi_2 + e^{l \times \beta_2})^{-1}) \times p_{a^*}^{x^*}$. P_1, on the other hand, selects a^* with probability $1 - (\xi_1 + e^{l \times \beta_1})^{-1}$. Hence, if $p_{a^*}^{x^*} > \frac{1-(\xi_1+1)^{-1}}{1-(\xi_2+1)^{-1}}$, then P_2 selects a^* with higher probability than P_1 in the first iteration ($l = 0$). Eventually, however, P_1 outperforms P_2, since $\lim_{l \to \infty} 1 - (\xi_1 + e^{l \times \beta_1})^{-1} = 1$, and $\lim_{l \to \infty} (1 - (\xi_2 + e^{l \times \beta_2})^{-1}) \times p_{a^*}^{x^*} = p_{a^*}^{x^*} \leq 1$. As in the first iteration P_1 selects randomly, $1 - (\xi_1 + 1)^{-1} = \frac{1}{|\mathbf{A}|}$. Similarly, $1 - (\xi_2 + 1)^{-1} = \frac{1}{|\mathbf{X}|}$. Therefore, $\xi_1 = \frac{1}{|\mathbf{A}|-1}$ and $\xi_2 = \frac{1}{|\mathbf{X}|-1}$. Hence, if: $p_{a^*}^{x^*} > \frac{1 - \frac{1}{|\mathbf{A}|-1+1}}{1 - \frac{1}{|\mathbf{X}|-1+1}} \rightsquigarrow p_{a^*}^{x^*} > \frac{1 - \frac{|\mathbf{A}|-1}{|\mathbf{A}|}}{1 - \frac{|\mathbf{X}|-1}{|\mathbf{X}|}} \rightsquigarrow p_{a^*}^{x^*} > \frac{|\mathbf{X}|}{|\mathbf{A}|}$, then P_2 outperforms P_1 until a certain iteration τ. We only need one x such that $p_{a^*}^x > \frac{|\mathbf{X}|}{|\mathbf{A}|}$, since $p_{a^*}^{x^*} \geq p_{a^*}^x$. □

We show examples of P_1 and P_2's theoretical learning curves in Figure 1. τ is the training iteration where learning over actions starts to outperform learning over algorithms. Note that Theorem 1 gives us sufficient, but not necessary conditions. That is, if $p_{a^*}^x > \frac{|\mathbf{X}|}{|\mathbf{A}|}$, we have a *formal guarantee* that learning over algorithms is better than over actions until a certain training iteration τ. However, there could be cases where $p_{a^*}^x \leq \frac{|\mathbf{X}|}{|\mathbf{A}|}$, $\forall x$, and P_2 still outperforms P_1.

For instance, consider 2 actions and 10 algorithms, where $p_{a^*}^x = 0.99$, $\forall x$. In $l = 0$, P_1 picks a^* with probability 0.5, while P_2 with probability 0.99, even though $0.99 < \frac{|\mathbf{X}|}{|\mathbf{A}|} = 5$.

P_2 outperforms P_1 up to a certain training iteration τ. The previous theorem shows that $\tau > 0$ if $\exists x, p_{a^*}^x > \frac{|\mathbf{X}|}{|\mathbf{A}|}$. We can obtain a lower bound τ' for τ, by solving the following equation: $1 - (\xi_1 + e^{\tau' \times \beta_1})^{-1} = \left(1 - (\xi_2 + e^{\tau' \times \beta_2})^{-1}\right) \times p_{a^*}^{x^*}$, since the probability of P_2 selecting a^* is at least $(1 - (\xi_2 + e^{\tau' \times \beta_2})^{-1}) \times p_{a^*}^{x^*}$. Hence, up to training iteration τ', we have a *formal guarantee* that P_2 is better than P_1, if the theorem condition is satisfied. If x^* is unknown, we derive a less tight lower bound $\tau'' < \tau'$ by solving: $1 - (\xi_1 + e^{\tau'' \times \beta_1})^{-1} = \left(1 - (\xi_2 + e^{\tau'' \times \beta_2})^{-1}\right) \times p_{a^*}^x$, where x is any algorithm. Hence, if one is able to estimate $p_{a^*}^x$ (for at least one x) and β, then one would also have a lower bound for τ, leading to a formal guarantee for learning over algorithms up to that training iteration (ξ can be calculated given $|\mathbf{A}|$ and $|\mathbf{X}|$).

Additionally, note that Theorem 1 *does not* say that we must have one algorithm whose probability of playing a^* is higher than the probability of playing any other a. The condition $p_{a^*}^x > \frac{|\mathbf{X}|}{|\mathbf{A}|}$ can be valid, even if $\exists a \neq a^*$ such that $p_a^x > p_{a^*}^x$. In fact, we can show that learning over algorithms outperforms over actions in a very large action space, even if the probability of an algorithm selecting the best action is very small:

Corollary 1. *As $|\mathbf{A}| \to \infty$, P_2 is better than P_1 for a finite number of iterations, if and only if $\exists x$, where $p_{a^*}^x > 0$.*

Proof. Follows from $\lim_{|\mathbf{A}| \to \infty} \frac{|\mathbf{X}|}{|\mathbf{A}|} = 0$, thus $p_{a^*}^x > 0$ satisfies Theorem 1. The "only if" side is trivially true. □

Hence, if the number of actions is very large, we only need $p_{a^*}^x > 0$ for at least one algorithm. This result is very relevant even in domains where a designer cannot easily estimate $p_{a^*}^x$.

Interestingly, however, Theorem 1 seems to suggest that the higher the number of algorithms, the worse, as we have that:

Corollary 2. *If $|\mathbf{X}| = 1$, x only needs to play better than uniformly random. As $|\mathbf{X}|$ grows, however, the sufficient condition eventually is never satisfied, independent of $p_{a^*}^x$.*

Proof. Follows immediately from $\frac{|\mathbf{X}|}{|\mathbf{A}|} \to \frac{1}{|\mathbf{A}|}$ for $|\mathbf{X}| \to 1$, hence we need $p_{a^*}^x > \frac{1}{|\mathbf{A}|}$. Likewise, $\frac{|\mathbf{X}|}{|\mathbf{A}|} > 1$ for $|\mathbf{X}| > |\mathbf{A}|$, hence we would need $p_{a^*}^x > 1$, which is impossible. □

In fact, if there is a fixed algorithm x, where $p_{a^*}^x \geq p_{a^*}^{x'}$, $\forall x' \neq x$ in all states, then we should always pick x. Intuitively, however, it should be beneficial to have multiple algorithms to choose from. *Informally*, this may happen because different algorithms may perform better at different states, as discussed in Marcolino *et al.* [2013]. That is, in many domains we do not have a fixed algorithm x that has a higher probability of selecting a^* than the other algorithms in all states. Therefore, $|\mathbf{X}|$ may implicitly also affect the probability of P_2 selecting a^*, since $\mathcal{P}_l^2 \geq (1 - (\xi_2 + e^{l \times \beta_2})^{-1}) \times p_{a^*}^{x^*}$ (remember that x^* is the algorithm x with the highest $p_{a^*}^x$ across all $x \in \mathbf{X}$). Hence, *informally*, as the size n of $\mathbf{X}$ grows, we may have a greater chance of adding a new x_n that has a higher probability of playing a^* than the other algorithms (i.e., $p_{a^*}^{x_n} > p_{a^*}^{x_i}, \forall 0 \leq i < n$). Therefore, although adding a new algorithm may sacrifice initial performance, it may lead to a higher convergence point (i.e., a higher value for $\mathcal{P}_l^2 \geq (1 - (\xi_2 + e^{l \times \beta_2})^{-1}) \times p_{a^*}^{x^*}$ as $l \to \infty$). A larger value for $\mathcal{P}_l^2$ as $l \to \infty$ also increases the number of training iterations where the curve $\mathcal{P}_l^2$ is above $\mathcal{P}_l^1$. That is, we may have that $\tau_{\mathbf{X}'} > \tau_{\mathbf{X}}$, if $|\mathbf{X}'| > |\mathbf{X}|$. Hence, a larger $|\mathbf{X}|$ should increase the number of training iterations where P_2 still outperforms P_1.

Formally, however, it is not true that τ increases with $|\mathbf{X}|$ in general. The way $\mathcal{P}_l^2$ changes as new algorithms are added depends on the algorithms' pdfs. For example, if every new algorithm selects the worst action with probability 1, then P_2 just gets worse. If, however, we assume distributions over

$p^x_{a^*}$, then we can show that τ increases with $|\mathbf{X}|$. Similarly as before, this does not mean that an algorithm designer would directly sample a number from a distribution in order to "decide" $p^x_{a^*}$. We are just proposing to model the phenomenon of new algorithms being created as a distribution over $p^x_{a^*}$. For instance, given a set of algorithms $\mathbf{X}'$, one can calculate the average and standard deviation over all $p^x_{a^*}$, if one assumes that $p^x_{a^*}$ comes from a Gaussian distribution. As $|\mathbf{X}'|$ grows, the calculated average and standard deviation would approximate those of the true distribution. Hence, in order to *formally* study the effect of adding new algorithms, we evaluate different distributions for $p^x_{a^*}$. We analyze three possible cases below: (i) when $p^x_{a^*}$ comes from a uniform distribution; (ii) when $p^x_{a^*}$ comes from a Gaussian; (iii) when there is a fixed pool of algorithms $\tilde{\mathbf{X}}$ to choose from. Similar techniques could be employed to analyze the most appropriate distribution for a given domain.

The uniform distribution could model the case where there is not yet an established framework for developing "strong" algorithms (i.e., with a high $p^x_{a^*}$). Hence, the designer would not be able to develop an algorithm x with $p^x_{a^*}$ greater than some bound u; and given a certain state, the algorithm may be strong or weak with equal likelihood.

The Gaussian, on the other hand, models a situation with common knowledge or an established framework to develop strong algorithms (e.g., Monte Carlo Tree Search for computer Go). Then, we can expect that, in a set of algorithms, there will be a mean and a variance over $p^x_{a^*}$ (the variance resulting from different design decisions or parameter configurations).

Finally, we also consider the case where there is a fixed, previously known pool of algorithms available. That is, the designer must choose an algorithm $x' \in \tilde{\mathbf{X}}$ to include in $\mathbf{X}$.

We start by analyzing the uniform distribution:

Proposition 1. *If the underlying algorithm creation process originates x_i with $p^{x_i}_{a^*} \sim U(0,u)$, then: (i) $p^{x^*}_{a^*}$ grows with $|\mathbf{X}|$ in expectation; (ii) $\exists x$ where $p^x_{a^*} > \frac{|\mathbf{X}|}{|\mathbf{A}|}$ in expectation, if and only if $|\mathbf{X}| < u \times |\mathbf{A}| - 1$*

Proof. The expected value of the k-th order statistic of the uniform distribution with n samples is given by: $\frac{k \times u}{n+1}$. Hence, the expected maximum value for $p^x_{a^*}$ when $|\mathbf{X}| = n$ is $\frac{n \times u}{n+1}$ (which grows with n). In order for $\frac{n \times u}{n+1} > \frac{|\mathbf{X}|}{|\mathbf{A}|}$, we must have that $u \times |\mathbf{A}| > n + 1 \rightsquigarrow n < u \times |\mathbf{A}| - 1$. Conversely, if $|\mathbf{X}| = u \times |\mathbf{A}| - 1 + z$, for $z \geq 0$, we would have that: $\frac{(u \times |\mathbf{A}| - 1 + z) \times u}{u \times |\mathbf{A}| + z} > \frac{|\mathbf{X}|}{|\mathbf{A}|} \rightsquigarrow \frac{(u \times |\mathbf{A}| - 1 + z) \times u}{(u \times |\mathbf{A}| - 1 + z)} > \frac{u \times |\mathbf{A}| + z}{|\mathbf{A}|} \rightsquigarrow u \times |\mathbf{A}| > u \times |\mathbf{A}| + z$, which is not possible for $z \geq 0$. □

Since $p^{x^*}_{a^*}$ grows with $|\mathbf{X}|$, the proposition seems to indicate that we should use as large $\mathbf{X}$ as possible up to the upper bound $u \times |\mathbf{A}| - 1$. Interestingly, however, we show in synthetic experiments (Section 3.2) that performance still improves for $|\mathbf{X}| \geq u \times |\mathbf{A}| - 1$.

For the Gaussian, we find that:

Proposition 2. *If the underlying algorithm creation process originates algorithms x_i with $p^{x_i}_{a^*} \sim N(\mu, \sigma)$ (truncated to the interval $[0,1]$), then: (i) $p^{x^*}_{a^*}$ grows with $|\mathbf{X}|$ in expectation; (ii) $\exists x$, where $p^x_{a^*} > \frac{|\mathbf{X}|}{|\mathbf{A}|}$ in expectation, by following in order of priority: (a) $|\mathbf{X}| \geq 741$, if $|\mathbf{A}| > \frac{|\mathbf{X}|}{\mu + 3\sigma}$; (b) $|\mathbf{X}| \geq 44$, if $|\mathbf{A}| > \frac{|\mathbf{X}|}{\mu + 2\sigma}$ (c) $|\mathbf{X}| \geq 7$, if $|\mathbf{A}| > \frac{|\mathbf{X}|}{\mu + \sigma}$.*

Proof. From the "68–95–99.7" rule, we have: $p(p^x_{a^*} \geq \mu + \sigma) \approx 0.5 - \frac{0.6827}{2} = 0.15865$; $p(p^x_{a^*} \geq \mu + 2\sigma) \approx 0.5 - \frac{0.9545}{2} = 0.022275$; $p(p^x_{a^*} \geq \mu + 3\sigma) \approx 0.5 - \frac{0.9973}{2} = 0.00135$. Hence, in order to have in expectation at least one x such that $p^x_{a^*} \geq \mu + \sigma$, we need at least t_σ samples, where $t_\sigma \times 0.15865 = 1 \rightsquigarrow t_\sigma \approx 7$. Likewise, for $p^x_{a^*} \geq \mu + 2\sigma$, we need at least $t_{2\sigma} \approx 44$; and for $p^x_{a^*} \geq \mu + 3\sigma$, at least $t_{3\sigma} \approx 741$. $\mu + 3\sigma \geq \mu + 2\sigma \geq \mu + \sigma$ (the equality comes from $p^x_{a^*} > 1$ being equivalent to $p^x_{a^*} = 1$). Hence, $p^{x^*}_{a^*}$ grows with $|\mathbf{X}|$, in expectation. Now consider the sufficient condition $p^x_{a^*} > \frac{|\mathbf{X}|}{|\mathbf{A}|}$. For $p^x_{a^*} \geq \mu + 3\sigma$ in expectation, we need $\mu + 3\sigma \geq \frac{|\mathbf{X}|}{|\mathbf{A}|} \rightsquigarrow |\mathbf{A}| > \frac{|\mathbf{X}|}{\mu + 3\sigma}$. Likewise, for $p^x_{a^*} \geq \mu + 2\sigma$, we need $|\mathbf{A}| > \frac{|\mathbf{X}|}{\mu + 2\sigma}$; and for $p^x_{a^*} \geq \mu + \sigma$, $|\mathbf{A}| > \frac{|\mathbf{X}|}{\mu + \sigma}$. □

Hence, Proposition 2 allows a designer to estimate how many algorithms to use, even without an estimation of $p^x_{a^*}$ available. However, the proposition requires an estimation of μ and σ, which might come from previous knowledge designing and/or analyzing algorithms for the specific domain.

Fundamentally, however, even if all distribution parameters are unknown, Proposition 1 and Proposition 2 show that under distribution assumptions, one can expect $p^{x^*}_{a^*}$ to grow with $|\mathbf{X}|$. Since P_2 converges to $(1 - (\xi_2 + e^{l \times \beta_2})^{-1}) \times p^{x^*}_{a^*}$, then τ also grows with $|\mathbf{X}|$. We study this further in Section 3.2.

Next, we do not assume an underlying distribution. Instead, algorithms must be chosen from an existing pool $\tilde{\mathbf{X}}$ ($\mathbf{X} \subseteq \tilde{\mathbf{X}}$):

Proposition 3. *Let $\mathcal{P}_{x_i}$ be the probability that x_i has the highest $p^x_{a^*}$ (across all $x \in \tilde{\mathbf{X}}$) in a state s. Let p_e be the expected value of $p^x_{a^*}$ in $\tilde{\mathbf{X}}$. Then, in expectation: (i) $p^{x^*}_{a^*}$ grows with $|\mathbf{X}|$; (ii) $\exists x$, where $p^x_{a^*} > \frac{|\mathbf{X}|}{|\mathbf{A}|}$, if $|\mathbf{X}| < p_e \times |\mathbf{A}|$.*

Proof. Given n algorithms, the probability that at least one of them has the highest $p^x_{a^*}$ (across all $x \in \tilde{\mathbf{X}}$) is $p = 1 - \prod_{i=1}^{n}(1 - \mathcal{P}_{x_i})$. Clearly, $p \to 1$ as $n \to \infty$, and thus $p^{x^*}_{a^*}$ grows with $|\mathbf{X}|$. However, to satisfy the sufficient condition, we must have $p_e > \frac{|\mathbf{X}|}{|\mathbf{A}|} \rightsquigarrow |\mathbf{X}| < p_e \times |\mathbf{A}|$. □

Proposition 3 allows an estimation of the best $|\mathbf{X}|$ as $\lfloor p_e \times |\mathbf{A}| \rfloor$, given p_e. With a fixed $\tilde{\mathbf{X}}$, in some domains one could estimate p_e by experimentation over a set of states with a known ground truth. Fundamentally, however, it again shows that one should expect $p^{x^*}_{a^*}$ (and τ, consequently) to grow with $|\mathbf{X}|$. We study this further in the next section.

3.2 Synthetic Experiments

We ran several synthetic experiments, to better investigate the dilemma between learning over actions or algorithms. Each experiment consists of many simulations, where we randomly generate a single state MDP. That is, we sample the mean reward r_i from $N(0,1)$, for each action a_i. When playing a_i, the reward returned is sampled from $N(r_i, 0.25)$. For each

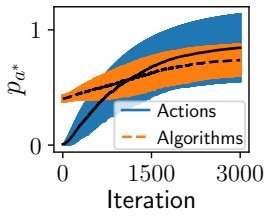


Figure 2: Example of p_{a^*} curves.

simulation, we create an agent that learns over actions, and another that learns over algorithms. For both we considered α and ϵ starting as 1, and decaying at the rate 0.999. We sample $p^x_{a^*}$ from different distributions, to simulate the creation of algorithms in a given domain.

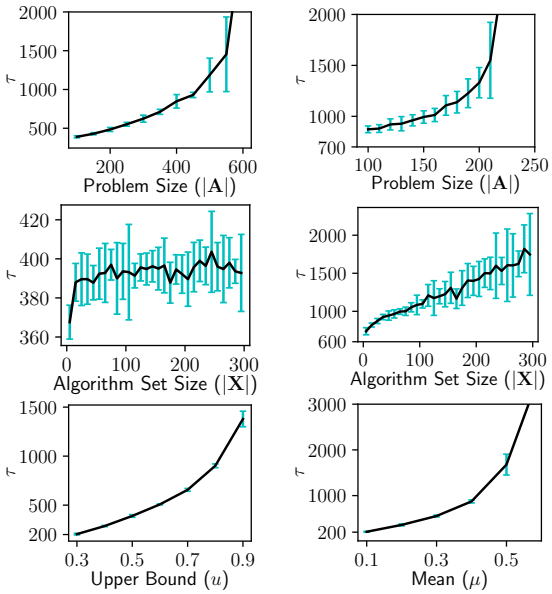
For each experiment, we run 1000 simulations of 10000 training iterations each. As an example, Figure 2 shows the probability of playing the best action (p_{a^*}) when learning over actions or algorithms, for a Gaussian model (black lines shows mean p_{a^*}, and colored areas indicate the standard deviation). Note how the curves follow a similar shape as the ones predicted by our theory (Figure 1). In the appendix[1], we show that the reward curves also follow a similar shape.

Our theoretical analysis does not yet give the exact number of iterations τ where learning over algorithms is better. Hence, in Figure 3 we study how τ changes as several parameters change (problem size $|\mathbf{A}|$, algorithm set size $|\mathbf{X}|$, u or μ), under a uniform or Gaussian model. A curve beyond the y-axis means that $\tau > 10000$. We repeat the whole procedure 5 times, and the error bars show the confidence interval ($p = 0.01$). When changing one parameter we fix the others (100 actions, 25 algorithms, $u = 0.5$, $\mu = 0.4$). We see that τ grows with statistical significance, under all parameters considered, for both models. When increasing u and μ we increase the overall expected performance of the algorithms, and hence this result is expected. It is interesting to note, however, that the curves tend to grow in an exponential fashion.

Concerning $|\mathbf{A}|$, the meeting point τ also tends to grow exponentially. Hence, it gets more advantageous to learn over algorithms as problems grow in complexity. This happens since it gets harder for a RL agent to find the best action, as it requires more exploration. On the other hand, we can still see τ increasing with $|\mathbf{X}|$. That is, even though it gets harder to find the best algorithm, $p^{x^*}_{a^*}$ tends to increase with $|\mathbf{X}|$, compensating for the harder exploration, as we discussed in our analysis. Based on Theorem 1, one may expect τ to drop when $|\mathbf{X}| > |\mathbf{A}|$. Interestingly, however, we see that τ tends to converge as $|\mathbf{X}|$ grows for both models, instead of dropping (remember that the theorem only gives sufficient conditions).

Our theory focused on p_{a^*}, but the actual reward obtained may be more significant. We evaluated the reward and cumulative reward, and found similar results (shown in the appendix[1]). In the uniform model reward curves, however, we notice that τ starts to drop when $|\mathbf{X}| \gg |\mathbf{A}|$.

[1]http://www.lancaster.ac.uk/staff/sorianom/ijcai18-ap.pdf



(a) Uniform Model (b) Gaussian Model

Figure 3: τ as number of actions, algorithms, u and μ grows.

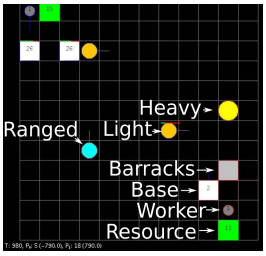


Figure 4: Screenshot of μRTS.

4 Learning over Algorithms in RTS Games

We evaluate learning over algorithms in a real-time strategy (RTS) game, which is a very complex domain, where directly learning over actions is impractical. The number of actions for a given state is estimated as over 10^{50} [Ontañón *et al.*, 2013]. Hence, we would need over 10^{50} training iterations just to explore a single time all the possible actions for a single state. The objective of this section, therefore, is to demonstrate that we can obtain a good performance when learning over algorithms in a complex and relevant domain.

RTS games are adversarial, normally involving resource management, construction, and combat between a large number of military units. They impose a great challenge for AI algorithms, since they have huge action and state spaces and require fast decisions. In this paper we use μRTS, a simplified RTS game developed for AI research[2].

A screenshot of μRTS is shown in Figure 4. In μRTS, entities are either *buildings*, *units* or *resources*. Buildings are either *bases*, which can produce *workers* or *barracks*, which

[2]https://github.com/santiontanon/microrts

produce military units. Units are either *workers*, which harvest resources and have limited combat ability; or military units. Military units are: *heavy* and *light*, which are strong but slow or weak but fast melee units, respectively; or *ranged*, which are long range attack units.

A set of four simple *rush* algorithms is available in μRTS: (i) *Worker*: create worker units, have one of them gather resources, and send all others to attack; (ii) *Ranged*: use a worker to gather resources. With enough resources, build a barrack and generate ranged units, sending them to attack. (iii) *Heavy*: same as Ranged, but creates the heavy unit instead; (iv) *Light*: same as before, but creates the light unit. In addition, we implemented two algorithms: (v) *BuildBarracks*: build a new barrack, allowing faster production of military units; and (vi) *Expand*: build a new base, increasing the production of worker units and faster gathering of multiple resources. All these compose our set $\mathbf{X}$. In order to handle the large state space, we propose next a Function Approximation approach.

4.1 Function Approximation (FA)

In this section, we say that we are taking an "action" $a \in \mathbf{A}$ in a state s even though we are selecting an algorithm $x \in \mathbf{X}$. This is to follow the traditional notation in RL literature. The main idea of FA is to learn a functional representation of the action-value function Q. This allows us to *generalize* Q for similar state-action pairs. We use SARSA [Rummery and Niranjan, 1994], with linear function approximation. Hence, a state s is represented by a feature vector $[k_1(s), \ldots, k_n(s)]$, and $Q(s,a)$ is approximated by $\tilde{Q}(s,a,w) = \sum_{i=1}^{n} k_i(s) \cdot w_i$, where $[w_1, \ldots, w_n]_a$ is a weight vector for action a. The learning problem is to find the best weights for each action. Each time the agent takes an action a, observes the next state s', and chooses an action a' in s', we update w_i with: $\Delta w_i = \alpha(r + \gamma\tilde{Q}(s',a',w) - \tilde{Q}(s,a,w)) \times k_i(s)$, where α is the training step size.

The features for a given state of μRTS are obtained as follows: we split the map into 3×3 quadrants of equal size. Within each quadrant, the number of units of each type owned by each player p is a separate feature. Thus, 9 quadrants, 7 unit types and 2 players lead to $9 \times 7 \times 2$ features. Additionally, the cumulative average health of each player's units within each quadrant is included, leading to 9×2 more features. We also include the resources harvested by each player, the current game time and the independent term, with value 1. Hence, given $\rho_p = \{u_1^1, \ldots, u_1^9, \ldots, u_7^1, \ldots, u_7^9\}$, where u_i^j is the number of units of type u_i in quadrant j for player p; and $\beta_p = \{h_1^1, \ldots, h_1^9, \ldots, h_7^1, \ldots, h_7^9\}$, where h_i^j is the cumulative average health of units of type u_i in quadrant j; the feature vector is: $k = [1, \rho_1, \rho_2, \beta_1, \beta_2, \omega_1, \omega_2, t]$, where ω_p is the amount of resources owned by player p, and t is the current game time. This linear combination of features is replicated for each $x \in \mathbf{X}$. Hence, we have $|\mathbf{X}|$ equations with $|k|$ features, leading to $|\mathbf{X}| \times |k|$ weights to adjust. We select an algorithm $x \in \mathbf{X}$ using exponentially decaying ϵ-greedy (decayed after every training game).

4.2 Results

We evaluate the performance of learning over algorithms using the proposed FA approach. We compare against the state of the art in μRTS: AHTN, PuppetMCTS, PuppetAB, NaiveMCTS and StrategyTactics. They are described in Section 2 and references therein. StrategyTactics won the 2017 μRTS competition, and NaiveMCTS was in the top 5. We used the map "basesWorkers24×24", and the best parametrization we found: $\alpha = 10^{-4}$, $\gamma = 0.9$, ϵ exponentially decaying from 0.2 against PuppetAB, PuppetMCTS and AHTN; and decaying from 0.1 for NaiveMCTS and StrategyTactics, after every game (decay rate ≈ 0.9984). All games have 3000 cycles at most, declared a draw on timeout. Rewards are -1, 0 or 1 for defeat, draw and victory, respectively.

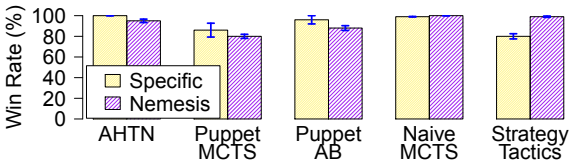


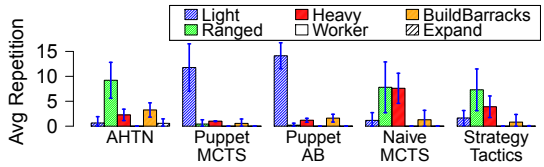
Figure 5: Learning over algorithms against state of the art players.

We perform two evaluations. In the first, named *Specific*, we trained FA in 500 games against PuppetAB, PuppetMCTS and AHTN; and in 100 games against NaiveMCTS and StrategyTactics. The resulting policy is tested against the same adversary that FA was trained against. In the second, named *Nemesis*, we: (i) trained FA against PuppetMCTS, fixing the resulting policy; and (ii) trained a new instance of FA against the resulting policy of (i), in 500 games. The single resulting policy of (ii) is tested against all adversaries (showing robustness). All tests have 100 games, with $\alpha = \epsilon = 0$. We ran 5 repetitions of all experiments, and the error bars show the 99% confidence interval. We consider statistical significance as $p \leq 0.01$. Figure 5 shows the results.

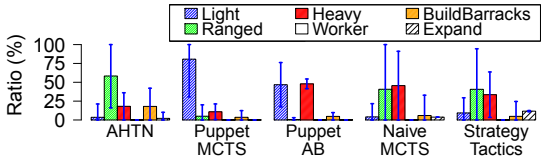
In both cases FA significantly defeats all opponents, with win rates higher than 80%. *Nemesis* and *Specific* have similar win rates, but *Nemesis* is significantly better against StrategyTactics. We believe this happens because *Nemesis* further elaborates on a policy that was already strong (the resulting policy of FA trained against PuppetMCTS).

Allowing algorithm switches at any state could have a negative effect: it could happen so frequently that algorithms would not be able to follow a course of action. Indeed, the agent may switch "too fast" during exploration, but eventually it learns a strong policy, and tends to pick a certain algorithm repeatedly if this leads to higher performance. On the other hand, the agent learns to switch to different algorithms when that is more profitable. Figure 6 confirms both situations with the *Specific* agents, by showing (a) the average number of times an algorithm is chosen consecutively and (b) the average percentage of selections (error bars indicate standard deviation). Hence, all algorithms tend to be chosen, but at different proportions depending on the adversary.

Finally, Figure 7 compares *Nemesis* and the individual algorithms ($x \in \mathbf{X}$), when playing against all opponents. We find that our performance is either significantly better than all individual algorithms, or not statistically different than the best algorithm (which is still relevant, since we may not know in advance which one to use), against each adversary. P-values when comparing against the best algorithm x are: 0.97,



(a) Repeated Selection



(b) Selection ratio

Figure 6: Average repeated selection of algorithms.

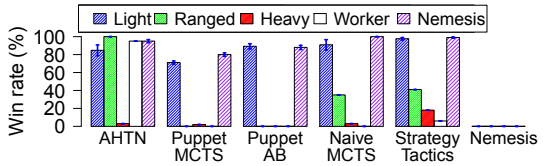


Figure 7: Algorithms and FA *Nemesis* against all AI players.

0.004, 0.61, 0.04, 0.14; for AHTN, PuppetMCTS, PuppetAB, NaiveMCTS and StrategyTactics, respectively. Additionally, the last set of bars shows that all algorithms are individually defeated by *Nemesis*.

5 Conclusion

Although action abstractions have been introduced before, our model for learning over algorithms gives novel guidelines backed by a theoretical analysis. Synthetic experiments demonstrate an increase in relative performance with action and algorithm set sizes. We also introduce a Function Approximation approach for learning over algorithms in RTS games, significantly outperforming state-of-the-art search-based players. The source code of synthetic and μRTS experiments are available at: https://github.com/andertavares/syntheticmdps and https://github.com/SivaAnbalagan1/micrortsFA, respectively.

Acknowledgments

We would like to thank Fapemig, CNPq, CAPES, and the School of Computing and Communications for their support. We also thank Tom McCracken for kindly checking our code.

References

[Barriga *et al.*, 2015] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games. In *AIIDE*, 2015.

[Barriga *et al.*, 2017] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Combining strategic learning and tactical search in real-time strategy games. In *AIIDE*, 2017.

[Barriga *et al.*, 2018] Nicolas A. Barriga, Marius Stanescu, and Michael Buro. Game Tree Search Based on Non-Deterministic Action Scripts in Real-Time Strategy Games. *IEEE Transactions on Games*, 10(1):67–77, 2018.

[Bontrager *et al.*, 2016] Philip Bontrager, Ahmed Khalifa, André Mendes, and Julian Togelius. Matching Games and Algorithms for General Video Game Playing. In *AIIDE*, 2016.

[Churchill and Buro, 2013] David Churchill and Michael Buro. Portfolio Greedy Search and Simulation for Large-Scale Combat in StarCraft. In *CIG*, 2013.

[Dietterich, 2000] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, 13:227–303, 2000.

[Jiang *et al.*, 2014] Albert Xin Jiang, Leandro Soriano Marcolino, Ariel D. Procaccia, Tuomas Sandholm, Nisarg Shah, and Milind Tambe. Diverse randomized agents vote to win. In *NIPS*, 2014.

[Lagoudakis and Littman, 2000] Michail G. Lagoudakis and Michael L. Littman. Algorithm selection using reinforcement learning. In *ICML*, 2000.

[Lelis, 2017] Levi H. S. Lelis. Stratified Strategy Selection for Unit Control in Real-Time Strategy Games. In *IJCAI*, 2017.

[Marcolino *et al.*, 2013] Leandro Soriano Marcolino, Albert Xin Jiang, and Milind Tambe. Multi-agent team formation: diversity beats strength? In *IJCAI*, 2013.

[Marcolino *et al.*, 2014] Leandro Soriano Marcolino, Haifeng Xu, Albert Xin Jiang, Milind Tambe, and Emma Bowring. Give a hard problem to a diverse team: Exploring large action spaces. In *AAAI*, 2014.

[Ontañón, 2017] Santiago Ontañón. Combinatorial multi-armed bandits for real-time strategy games. *JAIR*, 58, 2017.

[Ontañón and Buro, 2015] Santiago Ontañón and Michael Buro. Adversarial hierarchical-task network planning for complex real-time games. In *IJCAI*, 2015.

[Ontañón *et al.*, 2013] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 5(4):293–311, 2013.

[Rice, 1976] John R. Rice. The algorithm selection problem. *Advances in computers*, 15:65–118, 1976.

[Rummery and Niranjan, 1994] Gavin A. Rummery and Mahesan Niranjan. On-line Q-learning using connectionist systems. Technical report, Cambridge University, 1994.

[Sailer *et al.*, 2007] Frantisek Sailer, Michael Buro, and Marc Lanctot. Adversarial planning through strategy simulation. In *CIG*, 2007.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[Sutton *et al.*, 1999] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[Tavares *et al.*, 2016] Anderson Tavares, Hector Azpúrua, Amanda Santos, and Luiz Chaimowicz. Rock, Paper, StarCraft: Strategy Selection in Real-Time Strategy Games. In *AIIDE*, 2016.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *JAIR*, 32:565–606, 2008.