Asymmetric Action Abstractions for Multi-Unit Control in Adversarial Real-Time Games

Paper #567

Abstract

Action abstractions restrict the number of legal actions available during search in multi-unit adversarial games, thus allowing algorithms to focus on a set of promising actions during search. Optimal strategies derived from un-abstracted spaces are guaranteed to be no worse than optimal strategies derived from action-abstracted spaces. In practice, however, due to real-time constraints and the state space size, one is only able to derive good strategies in un-abstracted spaces in small-scale games. In this paper we introduce a novel scheme we call asymmetric action abstraction that retains the un-abstracted spaces' theoretical advantage over regularly abstracted spaces while still allowing algorithms to derive effective strategies, even in large-scale games. Empirical results on combat scenarios that arise in a real-time strategy game show that algorithms using asymmetric abstractions are able to substantially outperform state-of-the-art algorithms.

Introduction

In real-time strategy (RTS) games the player controls dozens of units to collect resources, build structures, and battle the opponent. RTS games are excellent testbeds for Artificial Intelligence methods because they offer fast-paced environments, where players act simultaneously, and the number of legal actions grows exponentially with the number of units the player controls. Also, the time allowed for planning is on the order of milliseconds. In this paper we assume deterministic and perfect information games, and focus on the problem of real-time control of combat units.

A successful family of algorithms for controlling combat units uses what we call action abstractions to reduce the number of legal actions available during the match. In RTS games, player actions are represented as a vector of unit moves, where each entry in the vector represents a move for a unit controlled by the player. Action abstractions reduce the number of legal actions a player can perform by reducing the number of legal moves each unit can perform.

Churchill and Buro (2013) introduced a method for building action abstractions through scripts. A script $\bar{\sigma}$ is a function mapping a game state s and a unit u to a move m for u. A set of scripts $\mathcal{P}$ induces an action abstraction by restricting the set of legal moves of all units to moves returned by the scripts in $\mathcal{P}$. We call an action abstraction created with Churchill and Buro's scheme a uniform abstraction.

In theory, players searching in un-abstracted spaces are guaranteed to derive optimal strategies that are no worse than the optimal strategies derived from action-abstracted spaces. This is because the former has access to actions that are not available in action-abstracted spaces. Despite its theoretical disadvantage, uniform abstractions are successful in large-scale games (Churchill and Buro 2013). This happens because the state space of RTS combat scenarios can be very large, and the problem's real-time constraints often allow search algorithms to explore only a small fraction of all legal actions before deciding on which action to perform next—uniform abstractions allow algorithms to focus their search on actions deemed as promising by the set of scripts $\mathcal{P}$.

Our main contribution is the introduction of asymmetric action abstractions (asymmetric abstractions for short) for multi-unit adversarial games. In contrast with uniform abstractions that restrict the number of moves of all units, asymmetric abstractions restrict the number of moves of only a subset of units. We show that asymmetric abstractions retain the un-abstracted spaces' theoretical advantage over uniformly abstracted ones while still allowing algorithms to derive effective strategies in practice, even in large-scale games. Another advantage of asymmetric abstractions is that they allow the search effort to be distributed unevenly amongst the units. This is important because some units might benefit more from finer strategies (i.e., strategies computed while accounting for a larger set of moves) than others (e.g., in RTS games it is advantageous to provide finer control to units with low hit points so they survive longer).

Another contribution of this paper is the introduction of two algorithms that search in asymmetrically abstracted spaces. Our algorithms are based on Portfolio Greedy Search (PGS) (Churchill and Buro 2013) and Stratified Strategy Selection (SSS) (Lelis 2017), two state-of-the-art algorithms. Empirical results on RTS combats show that our algorithms are able to substantially outperform PGS and SSS.

Related Work

Justesen et al. (2014) proposed two variations of UCT (Kocsis and Szepesvári 2006) for searching in uniformly abstracted spaces: script-based and cluster-based UCT. Wang et al. (2016) introduced Portfolio Online Evolution (POE) a local search algorithm also designed for uniformly abstracted spaces. Wang et al. showed that POE is able to outperform Justesen's algorithms, and Lelis (2017) showed that PGS and SSS are able to outperform POE. Justesen et al.'s and Wang et al.'s algorithms can also be modified to search in the asymmetrically abstracted spaces we introduce in this paper. We use PGS and SSS in this paper because they are the current state-of-the-art search-based algorithms for RTS combat scenarios (Lelis 2017).

Before the invention of action abstractions induced by scripts, state-of-the-art algorithms included search methods for un-abstracted spaces such as Monte-Carlo (Chung, Buro, and Schaeffer 2005; Sailer, Buro, and Lanctot 2007; Balla and Fern 2009; Ontañón 2013) and Alpha-Beta (Churchill, Saffidine, and Buro 2012). Due to the large number of actions available during search, Alpha-Beta and Monte-Carlo methods perform well only when controlling a small number of units. Some search algorithms cited are more general than the algorithms we consider in this paper, e.g., (Ontañón 2013; Ontañón and Buro 2015). This is because such algorithms can be used to control a playing agent throughout a complete RTS game. By contrast, the algorithms we consider in this paper are specialized for combat scenarios.

Another line of research uses learning to control combat units in RTS games. State-space search algorithms need an efficient forward model of the game in order to plan by searching. By contrast, learning approaches do not necessarily require an efficient forward model of the game to be available as they learn from their past experiences. Examples of learning-based approaches to unit control in RTS combat scenarios include the work by Usunier et al. (2016) and Liu et al. (2016). Likely due to the use of an efficient forward model, search-based algorithms tend to scale more easily to large-scale combat scenarios than learning-based methods. While search-based algorithms can effectively handle battles with more than 50 units on each side, learning-based ones are usually tested on battles with no more than 25 units on each side. For a review of RTS research, see the work by Ontañón et al. (2013).

Background

Combat scenarios that arise in RTS games, which we also call **matches**, can be described as finite zero-sum two-player games with simultaneous and durative moves. Matches can be defined by a tuple $(\mathcal{N}, \mathcal{S}, s_{init}, \mathcal{A}, \mathcal{R}, \mathcal{T})$, where, $\mathcal{N} = \{i, -i\}$ is the set of **players** (i is the player we control and $-i$ is our opponent); $\mathcal{S} = \mathcal{D} \cup \mathcal{F}$ is the set of **states**, where $\mathcal{D}$ denotes the set of **non-terminal states** and $\mathcal{F}$ the set of **terminal states**. Every state $s \in \mathcal{S}$ includes the joint set of **units** $\mathcal{U} = \mathcal{U}_i \cup \mathcal{U}_{-i}$, for players i and $-i$; $s_{init} \in \mathcal{D}$ is the start state and defines the initial position of the units $\mathcal{U}$; $\mathcal{A} = \mathcal{A}_i \times \mathcal{A}_{-i}$ is the set of joint **actions**. $\mathcal{A}_i(s)$ is the set of legal actions player i can perform at state s. Each action $a \in \mathcal{A}_i(s)$ is denoted by a vector of n **unit moves** $(m_1, \cdots, m_n)$, where $m_k \in a$ is the move of the k-th **ready unit** of player i. A unit u is not ready at s if u is performing a move. We denote the set of ready units of players i and $-i$ as $\mathcal{U}_i^r$ and $\mathcal{U}_{-i}^r$. For $k \in \mathbb{N}^+$ we write $a[k]$ to denote the move of the k-th ready unit. Also, for unit u, we write $a[u]$ to denote the move of u in a. $\mathcal{R}_i : \mathcal{F} \rightarrow \mathbb{R}$ is a **utility function** with $\mathcal{R}_i(s) = -\mathcal{R}_{-i}(s)$, for any $s \in \mathcal{F}$. The **transition function** $\mathcal{T} : \mathcal{S} \times \mathcal{A}_i \times \mathcal{A}_{-i} \rightarrow \mathcal{S}$ determines the sucessor state for a state s and the set of joint actions taken at s.

We denote the set of unit moves as $\mathcal{M}$, which includes moving up (U), left (L), right (R) and down (D), waiting (W), and attacking an enemy unit. We write $\mathcal{M}(s, u)$ to denote the set of legal moves of unit u at s. Every unit $u \in \mathcal{U}$ has properties such as its x and y coordinates, attack range ($r(u)$), attack damage ($d(u)$), unit move duration, hit points ($hp(u)$), and weapon cool-down time, i.e., the time the unit has to wait before repeating an attack action ($cd(u)$). The amount of damage u can cause per frame of the game is defined as $dpf(u) = \frac{d(u)}{cd(u)+1}$ (we add one to the denominator to ensure a valid operation, even if $cd = 0$). Finally, the attack value of a unit is defined as $av(u) = \frac{dpf(u)}{hp(u)}$. A unit u has a large attack value if u is able to cause lots of damage per frame and/or if it has very few hit points.

A **decision point** of player i is a state s in which i has at least one ready unit. In the framework we consider in this paper, a search algorithm is invoked at every decision point. The algorithm is allowed 40 milliseconds to search and return an action vector with one move for each ready unit.

The **game tree** of a match is a tree in which each node in the tree represents a state in $\mathcal{S}$ and every edge represents a joint action in $\mathcal{A}$; the tree is rooted at s_{init}. For states $s_k, s_j \in \mathcal{S}$, there exists an outgoing edge from s_k to s_j if and only if there exists $a_i \in \mathcal{A}_i$ and $a_{-i} \in \mathcal{A}_{-i}$ such that $\mathcal{T}(s_k, a_i, a_{-i}) = s_j$. Nodes representing states in $\mathcal{F}$ are leaf nodes. We assume the tree to be bounded, i.e., the match finishes after a finite number of decision points. We denote as Ψ the **evaluation function** used by search algorithms while traversing the game tree. Ψ receives as input a state s and returns an estimate of the end-game value of s.

A **player strategy** is a function $\sigma_i : \mathcal{S} \times \mathcal{A}_i \rightarrow [0, 1]$ for player i, which maps a state s and an action vector a to a probability value, indicating the chance of taking action a at s. We denote as Σ_i and Σ_{-i} as the set of all player strategies for players i and $-i$. A **player strategy profile** $\sigma = (\sigma_i, \sigma_{-i})$ defines the strategy of both players. We denote as $\pi_s^\sigma(z)$ as the probability of reaching terminal state $z \in \mathcal{Z}$ while following σ from state s. The optimal **value of the game** rooted at state s for player i can be computed by solving the following equation.

$$V_i(s) = \max_{\sigma_i \in \Sigma_i} \min_{\sigma_{-i} \in \Sigma_{-i}} \sum_{z \in \mathcal{Z}} \pi_s^{(\sigma_i, \sigma_{-i})}(z) \cdot \mathcal{R}_i(z) . \qquad (1)$$

The strategy profile $\sigma = (\sigma_i, \sigma_{-i})$ that solves Equation 1 is known as a Nash equilibrium (NE) profile, which guarantees a utility of $V_i(s)$ for player i in expectation. A NE profile is referred to as an optimal profile. Backward induction methods (Ross 1971) can be used to find optimal profiles by searching in the game tree. However, due to the problem's size and real-time constraints, this approach is impractical for most RTS combats. State-of-the-art algorithms employ action abstraction schemes to reduce the game tree size and then derive player strategies from the smaller tree.

Algorithm 1 Portfolio Greedy Search

```
Require: state $s$, available units $\mathcal{U}_i^a = \{u_1^i, \cdots, u_{n_i}^i\}$ and
    $\mathcal{U}_{-i}^a = \{u_1^{-i}, \cdots, u_{n_{-i}}^{-i}\}$ in $s$, unit strategies $\mathcal{P}$, time
    limit $t$, and evaluation function $\Psi$
Ensure: action vector $a$ for player $i$'s units
 1: $\bar{\sigma}_i \leftarrow$ choose a script from $\mathcal{P}$ //see text for details
 2: $\bar{\sigma}_{-i} \leftarrow$ choose a script from $\mathcal{P}$ //see text for details
 3: $a_i \leftarrow \{\bar{\sigma}_i(u_1^i), \cdots, \bar{\sigma}_i(u_{n_i}^i)\}$
 4: $a_{-i} \leftarrow \{\bar{\sigma}_{-i}(u_1^{-i}), \cdots, \bar{\sigma}_{-i}(u_{n_{-i}}^{-i})\}$
 5: while time elapsed is not larger than $t$ do
 6:     for $k \leftarrow 1$ to $|\mathcal{U}_i^a|$ do
 7:         for each $\bar{\sigma} \in \mathcal{P}$ do
 8:             $a_i' \leftarrow a_i$; $a_i'[k] \leftarrow \bar{\sigma}(s, u_k^i)$
 9:             if $\Psi(\mathcal{T}(s, a_i', a_{-i})) > \Psi(\mathcal{T}(s, a_i, a_{-i}))$ then
10:                 $a_i \leftarrow a_i'$
11:         if time elapsed is larger than $t$ then
12:             return $a_i$
13: return $a_i$
```

Uniform Action Abstractions

We define a **uniform abstraction** for player i as a function mapping the set of legal actions $\mathcal{A}_i$ to a subset $\mathcal{A}_i'$ of $\mathcal{A}_i$. In RTS games, action abstractions are constructed from a collection of scripts. A **script** $\bar{\sigma}$ is a function mapping a state s and a unit u in s to a legal move m for u. A script $\bar{\sigma}$ can be used to define a player strategy σ_i by applying $\bar{\sigma}$ to every unit in the state. We write $\bar{\sigma}$ instead of $\bar{\sigma}(s, u)$ whenever the state and the unit are clear from the context.

Let the **action-abstracted legal moves** of u at state s be the moves for u that is returned by a script in $\mathcal{P}$, defined as,

$$\mathcal{M}(s, u, \mathcal{P}) = \{\bar{\sigma}(s, u) | \bar{\sigma} \in \mathcal{P}\} .$$

Definition 1 *A uniform abstraction Φ is a function receiving as input a state s, a player i, and a set of scripts $\mathcal{P}$. Φ returns a subset of $\mathcal{A}_i(s)$ denoted $\mathcal{A}_i'(s)$. $\mathcal{A}_i'(s)$ is defined by the Cartesian product of moves in $\mathcal{M}(s, u, \mathcal{P})$ for all u in $\mathcal{U}_i^r$, where $\mathcal{U}_i^r$ is the set of ready units of i in s.*

Algorithms using a uniform abstraction Φ search in a game tree whose the legal actions are limited to $\mathcal{A}_i'(s)$ for all s. This way, algorithms focus their search in the actions deemed as promising by the scripts in $\mathcal{P}$, as the actions in $\mathcal{A}_i'(s)$ are composed of moves returned by the scripts in $\mathcal{P}$.

Two scripts commonly used for inducing uniform abstractions are known as NOKAV and Kiter (Churchill and Buro 2013). NOKAV assigns a move to a unit u so that u does not cause more damage than that required to reduce an enemy's unit hp to zero. Kiter allows a unit to attack and then move away from the target.

Searching in Uniformly Abstracted Spaces

Churchill and Buro (2013) introduced PGS, a method for searching in uniformly abstracted spaces. Algorithm 1 presents PGS's pseudocode. PGS receives as input a state s, player i's and $-i$'s set of ready units for s ($\mathcal{U}_i^r$ and $\mathcal{U}_{-i}^r$), a set of scripts $\mathcal{P}$, a time limit t, and an evaluation function Ψ. PGS returns an action vector a for player i to be executed in s. PGS selects the script $\bar{\sigma}_i$ (resp. $\bar{\sigma}_{-i}$) from $\mathcal{P}$ (lines 1 and 2) that yields the largest Ψ-value assuming player i executes an action composed of moves computed with $\bar{\sigma}_i$ for all units in $\mathcal{U}_i^r$ (resp. $\mathcal{U}_{-i}^r$). Action vectors a_i and a_{-i} are initialized with the moves computed from $\bar{\sigma}_i$ and $\bar{\sigma}_{-i}$.

Once a_i and a_{-i} have been initialized, PGS iterates through all units u_k^i in $\mathcal{U}_i^r$ and tries to greedily improve the move assigned to u_k^i in a_i, denoted by $a_i[k]$. Since PGS only assigns moves to units given by scripts in $\mathcal{P}$, it considers only actions in the space induced by a uniform abstraction. PGS evaluates a_i for each possible move $\bar{\sigma}(s, u_k^i)$ for unit u_k^i. PGS keeps in a_i the action found during search with the largest Ψ-value. This process is repeated until PGS reaches time limit t. PGS then returns a_i.

The action vector a_{-i} does not change after its initialization (see line 4). Although in PGS's original formulation one alternates between improving player i's and player $-i$'s action vectors (Churchill and Buro 2013), Churchill and Buro suggest to keep player $-i$'s action fixed after initialization as that leads to better results in practice.

Lelis (2017) introduced Stratified Strategy Selection (SSS), a hill-climbing algorithm for uniformly abstracted spaces similar to PGS. The main difference between PGS and SSS is that the latter searches in the space induced by a partition of units called a type system. SSS assigns moves returned by the same script to units of the same type. For example, all units with low hp-value (type) move away from the battle (strategy encoded in a script). In terms of pseudocode, SSS initializes $\bar{\sigma}_i$ and $\bar{\sigma}_{-i}$ with the NOKAV script (lines 1 and 2). Instead of iterating through all units as PGS does, SSS iterates through all types q of units in line 6 of Algorithm 1 and assigns the move provided by $\bar{\sigma}$ to all units of type q before evaluating the resulting state with Ψ. Finally, SSS uses a meta-reasoning method to automatically select the type system to be used during search. In this paper we call SSS what Lelis (2017) called SSS+.

Asymmetric Action Abstractions

Uniform abstractions are restrictive in the sense that all units have their legal moves reduced to those specified by scripts. In this section we introduce an abstraction scheme we call asymmetric abstraction that is not as restrictive as uniform abstractions but still uses the guidance of the scripts for selecting a subset of promising actions. The key idea behind asymmetric abstractions is to reduce the number of legal moves of only a subset of the units controlled by player i; the sets of legal moves of the other units remain unchanged. We call the subset of units that do not have their set of legal moves reduced the **unrestricted units**; the complement of the unrestricted units are referred as the **restricted units**.

Definition 2 *An asymmetric abstraction Ω is a function receiving as input a state s, a player i, a set of unrestricted units $\mathcal{U}_i' \subseteq \mathcal{U}_i^r$, and a set of scripts $\mathcal{P}$. Ω returns a subset of actions of $\mathcal{A}_i(s)$, denoted $\mathcal{A}_i''(s)$, defined by the Cartesian product of the moves in $\mathcal{M}(s, u, \mathcal{P})$ for all u in $\mathcal{U}_i^r \setminus \mathcal{U}_i'$ and of moves $\mathcal{M}(s, u')$ for all u' in $\mathcal{U}_i'$.*

Algorithms using an asymmetric abstraction Ω search in a game tree whose the legal actions are limited to $\mathcal{A}_i''(s)$

for all s. If the set of unrestricted units equals the set of units controlled by the player, then the asymmetric abstraction is equivalent to the un-abstracted space, and if the set of unrestricted units is empty, the asymmetric abstraction is equivalent to the uniform abstraction induced by the same set of scripts. Asymmetric abstractions allow us to explore the action abstractions in the large spectrum of possibilities between the uniformly abstracted and un-abstracted spaces.

The following theorem shows that an optimal strategy derived from the space induced by an asymmetric abstraction is at least as good as the optimal strategy derived from the space induced by a uniform abstraction as long as both abstractions are defined by the same set of scripts.

Theorem 1 *Assume abstractions Φ and Ω defined with the same set of scripts $\mathcal{P}$ and a match with start state s. Let $V_i^{\Phi}(s)$ be the optimal value of the game computed by considering the space induced by Φ; define $V_i^{\Omega}(s)$ analogously. We have that $V_i^{\Omega}(s) \geq V_i^{\Phi}(s)$.*

The proof for Theorem 1, which is provided in the appendix, hinges on the fact that a player searching with Ω has access to more actions than a player searching with Φ. This guarantee can also be achieved by considering a larger set $\mathcal{P}$ to induce Φ. The problem of increasing the size of $\mathcal{P}$ is that new scripts might not be readily available as they need to be either handcrafted or learned. By contrast, one can easily create a wide range of asymmetric abstractions by changing the size of the set of unrestricted units. Also, depending on the combat scenario, some units might be more important than others (e.g., units with low hp-values), and asymmetric abstractions allow one to assign finer strategies to these units. Similarly to what human players do, asymmetric abstractions allow algorithms to focus on a subset of units at a given time of the match. This is achieved by considering all legal moves of the unrestricted units during search.

A corollary of Theorem 1 is that optimal strategies derived from the un-abstracted space are at least as good as the optimal strategies derived from action-abstracted spaces, as the un-abstracted space is a special case of the asymmetric abstraction. Similar to un-abstracted spaces, asymmetric abstractions have the theoretical advantage over uniform abstractions while still reducing the size of the action space, thus allowing one to find effective strategies in practice.

Searching with Asymmetric Abstractions

We introduce Greedy Alpha-Beta Search (GAB) and Stratified Alpha-Beta Search (SAB), two algorithms for searching in asymmetrically abstracted spaces. GAB and SAB hinge on a property of PGS and SSS that have hitherto been overlooked. Namely, both PGS and SSS may come to an early termination if they encounter a local maximum. PGS and SSS reach a local maximum when they complete all iterations of the outer for loop in Algorithm 1 (line 6) without altering a_i (line 10). Once a local maximum is reached, PGS and SSS are unable to further improve the move assignments, even if the time limit t was not reached.

GAB and SAB take advantage of PGS's and SSS's early termination by operating in two steps. In the first step GAB and SAB search for an action vector in the uniformly abstracted space with PGS and SSS, respectively. The first step finishes either when (i) the time limit is reached or (ii) a local maximum is encountered. In the second step, which is run only if the first step finishes by encountering a local maximum, GAB and SAB fix the moves of all restricted units according to the moves found in the first step, and search in the asymmetrically abstracted space for moves for all unrestricted units. If the first step finishes by reaching the time limit, GAB and SAB return the action determined in the first step. GAB and SAB behave exactly like PGS and SSS in decision points in which the first step uses all time allowed for planning. We explain GAB and SAB in more detail below.

We also implemented a variant of PGS for searching in asymmetric spaces that is simpler than the algorithms we present in this section. In this PGS variant, during the hill-climbing search, for a given state s, instead of limiting the number of legal moves of all units u to $\mathcal{M}(s, u, \mathcal{P})$, as PGS does, we consider all legal moves $\mathcal{M}(s, u)$ for unrestricted units, and the moves $\mathcal{M}(s, u, \mathcal{P})$ for restricted units. We do not present the empirical results of this PGS variant because it did not perform as well as GAB and SAB.

Greedy Alpha-Beta Search (GAB) In its first step GAB uses PGS to search in a uniformly abstracted space induced by $\mathcal{P}$ for deriving an action a that is used to fix the moves of the restricted units during the second search. In its second step, GAB uses a variant of Alpha-Beta that accounts for durative moves (Churchill, Saffidine, and Buro 2012) (ABCD). Although we use ABCD, one could also use other algorithms such as UCTCD (Churchill and Buro 2013). ABCD is used to search in a tree we call **Move-Fixed Tree** (MFT). The following example illustrates how the MFT is defined; MFT's definition follows the example.

Example 1 *Let $\mathcal{U}_i = \mathcal{U}_i^r = \{u_1, u_2, u_3\}$ be player i's units in state s (all units are ready), $\mathcal{P} = \{\bar{\sigma}_1, \bar{\sigma}_2\}$ be a set of scripts, and $\{u_1, u_3\}$ be the unrestricted units. Also, let $a = (W, L, R)$ be the action vector returned by PGS while searching in the uniformly abstracted space induced by $\mathcal{P}$. GAB's second step performs a search in the MFT.*

The MFT is rooted at s, and all legal action vectors in s is obtained by fixing $a[u_2] = L$ and considering all legal moves of u_1 and u_3. That is, if $\mathcal{M}(s, u_1) = \{W, U\}$ and $\mathcal{M}(s, u_3) = \{R, D\}$, then the set of legal actions in s is:

$$\mathcal{A}_i(s) = \{(W, L, R), (W, L, D), (U, L, R), (U, L, D)\}\,.$$

s For all descendants states s' of s in the MFT, if $\mathcal{M}(s', u_1) = \{W, U\}$ and $\mathcal{M}(s', u_3) = \{R, D\}$, then the set of legal actions in s' is:

$$\begin{aligned}\mathcal{A}_i(s') = &\{(W, \bar{\sigma}_1(s', u_2), R), (W, \bar{\sigma}_1(s', u_2), D),\\ &(U, \bar{\sigma}_1(s', u_2), R), (U, \bar{\sigma}_1(s', u_2), D)\}\,.\end{aligned}$$

Here, $\bar{\sigma}_1 \in \mathcal{P}$ is what we call the default script of the MFT.

Definition 3 (Move-Fixed Tree) *For a given state s, a subset of unrestricted units of $\mathcal{U}_i$ in s, a set of scripts $\mathcal{P}$, and an action a returned by the first step, a Move-Fixed Tree (MFT) is a tree rooted at s with the following properties.*

1. *The set of legal actions $\mathcal{A}_i(s)$, where s is the root of the MFT, is limited to actions a' that have moves $a'[u]$ fixed to $a[u]$, for all restricted units u;*
2. *The set of legal actions $\mathcal{A}_i(s')$, for states s' descendants of s in the MFT, is limited to actions a' that have moves $a'[u]$ fixed to $\bar{\sigma}(s', u)$, for all restricted units u, where $\bar{\sigma} \in \mathcal{P}$ is the MFT's default script.*
3. *The set of legal actions for player $-i$ in state s'', $\mathcal{A}_{-i}(s'')$, is fixed to the move returned by the MFT's default script to all units in $\mathcal{U}_{-i}$ and all s'' in the MFT.*

By searching in the MFT, ABCD searches for moves for the unrestricted units while the moves of all other units, including the opponent's units, are fixed. We fix the opponent's moves to the NOKAV (our default script) as was done in previous work (Churchill and Buro 2013; Wang et al. 2016; Lelis 2017). The output of GAB is the action vector returned by ABCD's search in the MFT. GAB returns the action vector a returned by PGS if its ABCD search is unable to find an action vector with better Ψ value.

Stratified Alpha-Beta Search (SAB) The difference between SAB and GAB is the search algorithm used in their first step: while GAB uses PGS, SAB uses SSS. The second step of SAB follows exactly the second step of GAB.

GAB and SAB for Uniform Abstractions For any state s, the value of $\Psi(\mathcal{T}(s, a_i, a_{-i}))$ for the action a_i returned by PGS is a lower bound for the Ψ-value of the action returned by GAB. Similarly, SAB has the same guarantee over SSS. This is because the second step of GAB and SAB are performed only after a local maximum is reached. If the second step is unable to find an action with larger Ψ than the first step, both GAB and SAB return the action encountered in the first step. We introduce variants of GAB and SAB called GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$ that search in uniformly abstracted spaces to compare asymmetric with uniform abstractions.

The difference between GAB and SAB and their variants GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$ is that the latter only account for unit moves in $\mathcal{M}(s, u, \mathcal{P})$ for all s and u in their ABCD search. That is, in their second step search, GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$ only consider actions a' for which the moves $a'[u]$ for restricted units u are fixed (as in GAB's and SAB's MFT) and the moves $a'[u']$ for unrestricted units u' that are in $\mathcal{M}(s, u', \mathcal{P})$.

GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$ focus their search on a subset of units $\mathcal{U}'$ by searching deeper into the game tree with ABCD for $\mathcal{U}'$. In addition to searching deeper with ABCD, GAB and SAB focus their search on a subset of units $\mathcal{U}'$ by accounting for all legal moves of units in $\mathcal{U}'$ during search. If granted enough computation time, optimal algorithms using Ω derive stronger strategies than optimal algorithms using Φ (Theorem 1). In practice, due to the real-time constraints, algorithms are unable to compute optimal strategies for most of the decision points. We analyze empirically, by comparing GAB$_{\mathcal{P}}$ to GAB and SAB$_{\mathcal{P}}$ to SAB, which abstraction scheme allows one to derive stronger strategies.

Strategies for Selecting Unrestricted Units

Asymmetric abstractions depends on (i) the unrestricted set size, and on (ii) the units composing the unrestricted set. We introduce three strategies for selecting units to compose the unrestricted set at a given decision point. Let N be the unrestricted set size. Also, we assume ties to be broken randomly.

1. **More attack value (AV+)**. AV+ selects the N units with the largest av-values at every decision point. By selecting the units with the largest av-values, AV+ provides a finer control to units with low hp-value and large dpf-value. We expect this strategy to perform well because it might be able to preserve longer in the match the units that have low hp and large dpf-values.
2. **Less attack value (AV-)**. AV- selects the units with smaller av-values. We expect this strategy to be outperformed by AV+, as explained above.
3. **Random (R)**. R randomly selects N units in the beginning of the match to be the unrestricted units. R replaces an unrestricted unit that has its hp-value reduced to zero by randomly selecting a restricted unit. We highlight that this is a domain-independent strategy and could be used in other domains in which one controls multiple units in an adversarial scenario such as soccer for robots.

Empirical Methodology

We use SparCraft[1] as our testbed. SparCraft is a simulation environment of Blizzard's StarCraft. In SparCraft the unit properties such as hit points, cool-down time, and damage are exactly the same as the original game. However, SparCraft does not implement fog of war, collisions, and unit acceleration (all units move at constant speed) (Churchill and Buro 2013). We use SparCraft because it offers an efficient forward model of the game, which is required by search-based methods. All experiments are run on 2.66 GHz CPUs.

Combat Configurations

We experiment with units with different hp, d, and r-values. We use $\uparrow$ to denote large and $\downarrow$ to denote small hp and d-values. Also, we call u a melee unit if u's attack range equals zero ($r = 0$), and we call u a ranged unit if u is able to attack from far ($r > 0$). Namely, we use the following unit kinds: Zealots (Zl, $\uparrow hp$, $\uparrow d$, melee), Dragoons (Dg, $\uparrow hp$, $\uparrow d$, ranged), Zerglings (Lg, $\downarrow hp$, $\downarrow d$, melee), Marines (Mr, $\downarrow hp$, $\downarrow d$, ranged). We consider the combat scenarios where each player controls units of the following kinds: (i) Zl; (ii) Zl and Dg; (iii) Zl, Dg, and Lg; and (iv) Zl, Dg, Lg, and Mr. We experiment with matches with as few as 6 units and as many as 56 units on each side. The largest number of units controlled by a player in a typical StarCraft combat is around 50 (Churchill and Buro 2013). The first two columns of Table 2 show all combat configurations used. The number of units is distributed equally amongst all kinds of units. For example, the scenario Zl+Dg+Lg+Mr with a total number of 56 units has 14 units of each kind.

The units are placed in a walled arena with no obstacles of size 1280 × 780 pixels; the largest unit (Dragoon) is approximately 40 × 50 pixels large. The walls ensure finite matches by preventing units from indefinitely moving

[1] github.com/davechurchill/ualbertabot/tree/master/SparCraft

GAB vs. PGS						
Strategy	Unrestricted Set Size N					Avg.
	2	4	6	8	10	
AV+	0,88	0,92	0,89	0,87	0,86	0.88
AV-	0,69	0,76	0,78	0,82	0,82	0.77
R	0,78	0,86	0,87	0,88	0,88	0.85

SAB vs. SSS						
Strategy	Unrestricted Set Size N					Avg.
	2	4	6	8	10	
AV+	0,89	0,92	0,90	0,88	0,90	0.87
AV-	0,69	0,76	0,78	0,70	0,82	0.75
R	0,75	0,80	0,83	0,84	0,85	0.81

Table 1: Winning rate of GAB against PGS and of SAB against SSS for different strategies and set sizes.

#Units		GAB$_{\mathcal{P}}$ PGS	GAB PGS	GAB GAB$_{\mathcal{P}}$	SAB$_{\mathcal{P}}$ SSS	SAB SSS	SAB SAB$_{\mathcal{P}}$
Zl	**(8)**	0.73	0.72	0.52	0.65	0.95	0.93
	(16)	0.78	0.79	0.57	0.70	0.96	0.94
	(32)	0.77	0.81	0.54	0.72	0.93	0.81
	(50)	0.80	0.78	0.50	0.69	0.90	0.76
Dg	**(8)**	0.69	0.94	0.88	0.60	0.91	0.88
	(16)	0.71	0.85	0.84	0.62	0.93	0.88
	(32)	0.68	0.81	0.82	0.65	0.88	0.81
	(50)	0.64	0.78	0.78	0.67	0.87	0.79
Zl+Dg	**(8)**	0.64	0.76	0.68	0.59	0.93	0.90
	(16)	0.66	0.82	0.78	0.66	0.93	0.86
	(32)	0.66	0.79	0.79	0.64	0.91	0.81
	(50)	0.65	0.74	0.71	0.63	0.90	0.77
Zl+Dg Lg	**(6)**	0.58	0.94	0.91	0.59	0.94	0.94
	(18)	0.66	0.93	0.90	0.67	0.94	0.89
	(42)	0.66	0.89	0.89	0.65	0.92	0.83
	(54)	0.64	0.86	0.89	0.63	0.89	0.79
Zl+Dg Lg+Mr	**(8)**	0.60	0.92	0.88	0.58	0.95	0.94
	(16)	0.64	0.94	0.91	0.59	0.95	0.91
	(40)	0.65	0.92	0.90	0.61	0.91	0.82
	(56)	0.66	0.92	0.90	0.60	0.85	0.75

Table 2: Top player's winning rate against bottom player.

away from the enemy. For each combat scenario we generate 1,000 start states as explained by Lelis (2017). Player i's units are placed at a random coordinate to the right of the center of the arena (with distance varying from 0 and 128 pixels). Player $-i$'s units are placed at a symmetric position to the left of the center. Then, we add 220 pixels to the x-coordinate of player i's units, and subtract 220 pixels from the x-coordinate player $-i$'s units, thus increasing the distance between enemy units by 440 pixels.

Testing Selection Strategies and Values of N

First, we test different strategies for selecting unrestricted units as well as different values of N. We test GAB against PGS and SAB against SSS (the algorithms used in the first step of GAB and SAB) with AV+, AV-, and R, with N varying from 1 to 10. Table 1 shows the average winning rates of GAB and SAB in 100 matches for each of the 20 combat configurations. Since the winning rate does not vary much with N, we show the winning rate of only even values of N. The "Avg." column shows the average across all N (1 to 10).

Both GAB and SAB outperform their base algorithms for all selection strategies and N values tested, even with the domain-independent R. The strategy that performs best is AV+, which obtains a winning rate of 0.92 with N of 4 for both GAB and SAB. The winning rate can vary considerably depending on the selection strategy for a fixed N. For example, for N of 2, PGS and SAB with AV+ obtain a winning rate of 0.88 and 0.89, respectively, while they obtain a winning rate of only 0.69 with AV-. These results demonstrate the importance of carefully selecting the set of units for which the algorithm will focus its search on.

Although GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$ do not search in asymmetrically abstracted spaces, their performance also depends on the set of units controlled in the algorithms' ABCD search. Thus, we tested GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$ with AV+, AV-, and R for selecting the units to be controlled in the algorithms' ABCD search. We also tested different number of units controlled in such searches: we tested set sizes from 1 to 10. Similar to the GAB and SAB experiments, we tested GAB$_{\mathcal{P}}$ against PGS and SAB$_{\mathcal{P}}$ against SSS; the detailed results are also omitted for space. The highest winning rate obtained by GAB$_{\mathcal{P}}$ against PGS was 0.74 while using the AV+ strategy to control 9 units in its ABCD search. The highest winning rate obtained by SAB$_{\mathcal{P}}$ against SSS+ was 0.78 while using the R strategy to control 9 units in its ABCD search.

GAB and SAB tend to perform best while controlling a smaller set of units (4 units in our experiment) in their ABCD search than GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$ (9 units). This is because GAB and SAB's ABCD search does not restrict the moves of the units, while GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$'s ABCD search does. GAB$_{\mathcal{P}}$ and SAB$_{\mathcal{P}}$ are able to effectively search deeper for a larger set of units than GAB and SAB. On the other hand, GAB and SAB are able to encounter finer strategies to the unrestricted units. Next, we directly compare these abstraction approaches with a detailed empirical study.

Asymmetric versus Uniform Abstractions

We test GAB, GAB$_{\mathcal{P}}$ and PGS (G-Experiment); and SAB, SAB$_{\mathcal{P}}$, SSS (S-Experiment). GAB, GAB$_{\mathcal{P}}$, SAB, and SAB$_{\mathcal{P}}$ use the best performing set sizes and strategies for defining the units controlled in their ABCD searches as described above. We also experimented with ABCD and UCTCD with un-abstracted spaces, but we omit their results since they performed much worse than the other algorithms. We use $\mathcal{P} = \{\text{NOKAV, Kiter}\}$, with a time limit of 40 milliseconds for planning. PGS, GAB, and GAB$_{\mathcal{P}}$ use the Ψ function described by Churchill and Buro (2013), and SSS+, SAB, and SAB$_{\mathcal{P}}$ use the one described by Lelis (2017).

The winning rates in 1,000 matches of the algorithms in the G-Experiment are shown on the lefthand side of Table 2. The first two columns of the table specify the kind and the total number of units controlled by each player in the match, respectively. If the player controls more than one kind of unit, the number of units is divided equally by the number of kinds of units controlled. The remaining columns show the winning rate of the top algorithm, shown in the first row of the table, against the bottom algorithm. For example, in matches with 16 Zealots and 16 Dragoons (total of 32 units) GAB defeats PGS in 79% of the matches. The winning rates of the algorithms in the S-Experiment are shown on the righthand side of the table.

We observe in the third and fourth columns of the table that both GAB$_\mathcal{P}$ and GAB outperform PGS in all configurations tested. However, these results do not allow us to verify the effectiveness of asymmetric abstractions if analyzed individually. This is because both GAB$_\mathcal{P}$ and PGS search in uniformly abstracted spaces, and GAB's advantage over PGS could be due to the use of a different search strategy, and not due to the use of a different abstraction scheme. By comparing the numbers across the two columns we observe that GAB, which uses asymmetric abstractions, obtains substantially larger winning rates over PGS than GAB$_\mathcal{P}$, which uses uniform abstractions. For example, in matches with 8 Zealots and 8 Dragoons (16 units in total), GAB$_\mathcal{P}$'s winning rate is 0.66 against PGS, while GAB's is 0.82.

The column GAB vs GAB$_\mathcal{P}$ of the table allows a direct comparison between uniform and asymmetric abstractions. GAB substantially outperforms GAB$_\mathcal{P}$ in almost all configurations, and its winning rate is never below 0.50. These results highlight the importance of focusing the search effort on a subset of units through an asymmetric abstraction.

The results for the S-Experiment are similar to those of the G-Experiment: SAB has a higher winning rate over SSS than SAB$_\mathcal{P}$ and SAB substantially outperforms SAB$_\mathcal{P}$. These results suggest that asymmetric abstractions allow one to derive stronger strategies than uniform abstractions.

Conclusions and Future Work

We introduced a novel abstraction scheme we call asymmetric abstraction. For not being too restrictive while filtering actions and for allowing search algorithms to assign finer strategies to a particular subset of units, algorithms using asymmetric abstractions are able to outperform those using uniform abstractions. We introduced GAB and SAB, algorithms that use asymmetric abstractions, and GAB$_\mathcal{P}$ and SAB$_\mathcal{P}$, algorithms that use uniform abstractions. Our empirical results showed that GAB$_\mathcal{P}$ and SAB$_\mathcal{P}$ already represent an improvement over the state-of-the-art search-based algorithms for RTS combats. GAB and SAB represent yet a much larger improvement. As future work we intend to apply GAB and SAB to complete RTS games and to compare them to other search-based approaches designed to play complete games such as NaiveMCTS (Ontañón 2013) and PuppetSearch (Barriga, Stanescu, and Buro 2017). We are also interested in developing algorithms that learn how to select the unrestricted set of units in the scenarios that appear in complete RTS games.

Appendix: Proofs

The proof of Theorem 1 hinges on the fact that one has access to more actions with Ω than with Φ. This idea is formalized in Lemma 1.

Lemma 1 *$\mathcal{A}'_i(s) \subseteq \mathcal{A}''_i(s)$ for any state s if both Φ and Ω are defined with the same set of scripts $\mathcal{P}$.*

Proof. By definition, the actions in $\mathcal{A}'_i(s)$ are generated by the Cartesian product of $\mathcal{M}(s,u,\mathcal{P})$ for all u in $\mathcal{U}_i$ in s. The actions in $\mathcal{A}'_i(s)$ are generated by the Cartesian product of $\mathcal{M}(s,u,\mathcal{P})$ for all u in $\mathcal{U}_i \setminus \mathcal{U}'_i$ and of $\mathcal{M}(s,u)$ for all u in $\mathcal{U}'_i$. Since, also by definition, $\mathcal{M}(s,u,\mathcal{P}) \subseteq \mathcal{M}(s,u)$, we have that $\mathcal{A}'_i(s) \subseteq \mathcal{A}''_i(s)$. □

Lemma 2 *Assume abstractions Φ and Ω defined with the same set of scripts $\mathcal{P}$ and a match with start state s in which a single joint action leads the match to a terminal state. Let $V_i^\Phi(s)$ be the optimal value of the game computed by considering the space induced by Φ; define $V_i^\Omega(s)$ analogously. We have that $V_i^\Omega(s) \geq V_i^\Phi(s)$.*

Proof.

$$\begin{aligned} V_i^\Omega(s) &= \max_{\sigma_i \in \Sigma_i} \min_{\sigma_{-i} \in \Sigma_{-i}} \sum_{a_i \in \mathcal{A}''(s)} \sum_{a_{-i} \in \mathcal{A}_{-i}(s)} \sigma_i(s,a_i) \cdot \\ &\qquad \sigma_{-i}(s,a_{-i}) \cdot \mathcal{R}_i(\mathcal{T}(s,a_i,a_{-i})) \\ &\geq \max_{\sigma_i \in \Sigma_i} \min_{\sigma_{-i} \in \Sigma_{-i}} \sum_{a_i \in \mathcal{A}'(s)} \sum_{a_{-i} \in \mathcal{A}_{-i}(s)} \sigma_i(s,a_i) \cdot \\ &\qquad \sigma_{-i}(s,a_{-i}) \cdot \mathcal{R}_i(\mathcal{T}(s,a_i,a_{-i})) = V_i^\Phi(s)\,. \end{aligned}$$

The first equality is the definition of the value of a zero-sum game. The inequality is because the max operation over a set E cannot be smaller than the max over a subset of E, and $\mathcal{A}'_i \subseteq \mathcal{A}''_i$. The last equality is analogous to the first. □

Theorem 1 *Assume abstractions Φ and Ω defined with the same set of scripts $\mathcal{P}$ and a match with start state s. Let $V_i^\Phi(s)$ be the optimal value of the game computed by considering the space induced by Φ; define $V_i^\Omega(s)$ analogously. We have that $V_i^\Omega(s) \geq V_i^\Phi(s)$.*

Proof. We use Lemma 2 inductively on the level of the game tree, which we assume to be finite. The base case is given by Lemma 2, the inductive hypothesis is that $V_i^\Omega(s') \geq V_i^\Phi(s')$ for any state at level $j+1$ of the tree. For any state s at level j we have that,

$$\begin{aligned} V_i^\Omega(s) &= \max_{\sigma_i \in \Sigma_i} \min_{\sigma_{-i} \in \Sigma_{-i}} \sum_{a_i \in \mathcal{A}''(s)} \sum_{a_{-i} \in \mathcal{A}_{-i}(s)} \sigma_i(s,a_i) \cdot \\ &\qquad \sigma_{-i}(s,a_{-i}) \cdot V_i^\Omega(\mathcal{T}(s,a_i,a_{-i})) \\ &\geq \max_{\sigma_i \in \Sigma_i} \min_{\sigma_{-i} \in \Sigma_{-i}} \sum_{a_i \in \mathcal{A}'(s)} \sum_{a_{-i} \in \mathcal{A}_{-i}(s)} \sigma_i(s,a_i) \cdot \\ &\qquad \sigma_{-i}(s,a_{-i}) \cdot V_i^\Phi(\mathcal{T}(s,a_i,a_{-i})) = V_i^\Phi(s)\,. \end{aligned}$$

The first equality is because $V_i^\Omega(s)$ can be computed by using the values of $V_i^\Omega(s')$, where s' are the states at level $j+1$ in the tree. The inequality is because $\mathcal{A}'_i(s) \subseteq \mathcal{A}''_i(s)$ (Lemma 1) and $V_i^\Omega(\mathcal{T}(s,a_i,a_{-i})) \geq V_i^\Phi(\mathcal{T}(s,a_i,a_{-i}))$, as $\mathcal{T}(s,a_i,a_{-i})$ returns a state in level $j+1$ (inductive hypothesis). The last equality is analogous to the first one. □

Although in the proof of Theorem 1 we assume an optimal opponent searching in an un-abstracted space, the result holds for any fixed opponent. For example, $V_i^\Omega(s') \geq V_i^\Phi(s')$ holds if the opponent is using action abstractions.

References

Balla, R.-K., and Fern, A. 2009. Uct for tactical assault planning in real-time strategy games. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 40–45.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2017. Game tree search based on non-deterministic action scripts in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte Carlo planning in RTS games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*.

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Proceedings of the Conference on Computational Intelligence in Games*, 1–8. IEEE.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Justesen, N.; Tillman, B.; Togelius, J.; and Risi, S. 2014. Script- and cluster-based UCT for StarCraft. In *IEEE Conference on Computational Intelligence and Games*, 1–8.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the European Conference on Machine Learning*, 282–293. Springer-Verlag.

Lelis, L. H. S. 2017. Stratified strategy selection for unit control in real-time strategy games. In *International Joint Conference on Artificial Intelligence*, –.

Liu, S.; Louis, S. J.; and Ballinger, C. A. 2016. Evolving effective microbehaviors in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games* 8(4):351–362.

Ontañón, S., and Buro, M. 2015. Adversarial hierarchical-task network planning for complex real-time games. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1652–1658.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games* 5(4):293–311.

Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 58–64.

Ross, S. M. 1971. Goofspiel ? the game of pure strategy. *Journal of Applied Probability* 8(3):621?625.

Sailer, F.; Buro, M.; and Lanctot, M. 2007. Adversarial planning through strategy simulation. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 80–87.

Usunier, N.; Synnaeve, G.; Lin, Z.; and Chintala, S. 2016. Episodic exploration for deep deterministic policies: An application to StarCraft micromanagement tasks. *CoRR* abs/1609.02993.

Wang, C.; Chen, P.; Li, Y.; Holmgård, C.; and Togelius, J. 2016. Portfolio online evolution in StarCraft. In *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, 114–120.