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

Rubens O. Moraes, Levi H. S. Lelis

Departamento de Informática, Universidade Federal de Viçosa, Brazil

{rubens.moraes, levi.lelis}@ufv.br

Abstract

Action abstractions restrict the number of legal actions available during search in multi-unit real-time adversarial games, thus allowing algorithms to focus their search on a set of promising actions. 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 search algorithms that use an action abstraction scheme we call asymmetric abstraction. Asymmetric abstractions retain the un-abstracted spaces' theoretical advantage over regularly abstracted spaces while still allowing the search 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 our search algorithms are able to substantially outperform state-of-the-art approaches.

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 focus on the combat scenarios that arise in RTS games. A simplified version of RTS combats in which the units cannot move was shown to be PSPACE-hard in general (Furtak and Buro 2010).

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 game. 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 combats (Churchill and Buro 2013). This happens because the state space of RTS combats 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 $\mathcal{P}$.

In this paper we introduce search algorithms that use what we call 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).

The algorithms we introduce for searching in asymmetrically abstracted spaces are based on Portfolio Greedy Search (PGS) (Churchill and Buro 2013) and Stratified Strategy Selection (SSS) (Lelis 2017), two state-of-the-art approaches. 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 asymmetrically abstracted spaces. We use PGS and SSS in this paper as 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. Search algorithms need an efficient forward model of the game to plan. By contrast, learning approaches do not necessarily require such a model. Examples of learning approaches to unit control include the work by Usunier et al. (2016) and Liu et al. (2016). Likely due to the use of an efficient forward model, search algorithms tend to scale more easily to large-scale combat scenarios than learning-based methods. While the former can effectively handle battles with more than 100 units, the latter are usually tested on battles with no more than 50 units.

Preliminaries

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. We assume matches with deterministic actions in which all units are visible to both players. 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 $s \in \mathcal{S}$ defines a grid map containing a joint set of **units** $\mathcal{U} = \mathcal{U}_i \cup \mathcal{U}_{-i}$, for players i and $-i$. Every unit $u \in \mathcal{U}$ has properties such as u's x and y coordinates on the map, attack range ($r(u)$), attack damage ($d(u)$), hit points ($hp(u)$), and weapon cool-down time, i.e., the time the unit has to wait before repeating an attack action ($cd(u)$). $s_{init} \in \mathcal{D}$ is the start state and defines the initial position of the units $\mathcal{U}$ on the map.

- $\mathcal{A} = \mathcal{A}_i \times \mathcal{A}_{-i}$ 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 busy 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.

- 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. The effect of moves U, L, R, D is to change the unit x and y coordinates on the map; the effect of an attack move is the reduction of the target unit's hp value by the d-value of the unit performing the attack. We write $\mathcal{M}(s, u)$ to denote the set of legal moves of unit u at s.

- $\mathcal{R}_i : \mathcal{F} \to \mathbb{R}$ is a **utility function** with $\mathcal{R}_i(s) = -\mathcal{R}_{-i}(s)$, for any $s \in \mathcal{F}$. We use the LTD2 formula introduced by Kovarsky and Buro (2005) as utility function. LTD2 evaluates a state s with $\mathcal{U} = \mathcal{U}_i \cup \mathcal{U}_{-i}$ as follows.

$$\sum_{u \in \mathcal{U}_i} \sqrt{hp(u)} \cdot dpf(u) - \sum_{u \in \mathcal{U}_{-i}} \sqrt{hp(u)} \cdot dpf(u).$$

Here, dpf is the amount of damage u can cause per frame of the game and is defined as $dpf(u) = d(u)/(cd(u)+1)$ (we use $cd(u) + 1$ to ensure a valid operation if $cd = 0$).

- The **transition function** $\mathcal{T} : \mathcal{S} \times \mathcal{A}_i \times \mathcal{A}_{-i} \to \mathcal{S}$ determines the sucessor state for a state s and the set of joint actions taken at s.

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 to decide on the player's next action.

The **game tree** of a match is a tree rooted at s_{init} whose nodes represent states in $\mathcal{S}$ and every edge represents a joint action in $\mathcal{A}$. 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 all matches to be finite, i.e., that the tree is bounded. 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 for player i.

A **player strategy** is a function $\sigma_i : \mathcal{S} \times \mathcal{A}_i \to [0, 1]$ for player i, which maps a state s and an action a to a probability value, indicating the chance of taking action a at s. A strategy profile $\sigma = (\sigma_i, \sigma_{-i})$ defines the strategy of both players. The optimal **value of the game** rooted s for player i is denoted as $V_i(s)$ and can be computed by finding a Nash Equilibrium profile. Due to the problem's size and real-time constraints, it is impractical to find optimal profiles for most RTS combats. State-of-the-art approaches use abstractions to reduce the game tree size and then derive player strategies from the abstracted trees.

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

---

Algorithm 1 Portfolio Greedy Search

Require: state s, available units $\mathcal{U}_i^r = \{u_1^i, \cdots, u_{n_i}^i\}$ and
  $\mathcal{U}_{-i}^r = \{u_1^{-i}, \cdots, u_{n_{-i}}^{-i}\}$ in s, unit strategies $\mathcal{P}$, time
  limit t, and evaluation function Ψ.
Ensure: action 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 $|\mathcal{U}_i^r|$ **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

---

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 search in a game tree for which player i's legal actions are limited to $\mathcal{A}_i'(s)$ for all s. This way, algorithms focus their search on 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 $\mathcal{P}$.

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

Searching in Uniformly Abstracted Spaces

Churchill and Buro (2013) introduced PGS, a method for searching in uniformly abstracted spaces. Algorithm 1 presents PGS, which 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 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. $\mathcal{U}_{-i}^r$), assuming $-i$ (resp. i) executes an action selected by the NOKAV script. 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 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 actions (Churchill and Buro 2013), Churchill and Buro suggested 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. SSS uses a meta-reasoning method to select the type system to be used. 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 for which player i's legal actions are limited to $\mathcal{A}_i''(s)$ for all s. If the set of unrestricted units is equal to 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 *Let Φ be a uniform abstraction and Ω be an asymmetric abstraction, both defined with the same set of scripts $\mathcal{P}$. For a finite 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 (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 enlarging the set $\mathcal{P}$ used to induce Φ. The problem of enlarging $\mathcal{P}$ is that new scripts might not be readily available as they need to be either hand-crafted or learned. By contrast, one can easily create a wide range of asymmetric abstractions by modifying the set of unrestricted units. Also, depending on the combat scenario, some units might be more important than others 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.

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 has 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 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 call this PGS variant Greedy Asymmetric Search (GAS).

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

The MFT is rooted at s, and the set of abstracted legal actions 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 abstracted legal actions in s is: $\{(W, L, R), (W, L, D), (U, L, R), (U, L, D)\}$. 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 abstracted legal actions in s' is:

$$\{(W, \bar{\sigma}_1(s', u_2), R), (W, \bar{\sigma}_1(s', u_2), D),$$
$$(U, \bar{\sigma}_1(s', u_2), R), (U, \bar{\sigma}_1(s', u_2), D)\}.$$

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 $\mathcal{P}$, a default script $\bar{\sigma} \in \mathcal{P}$, and an action a returned by the search algorithm's first step, a Move-Fixed Tree (MFT) is a tree rooted at s with the following properties.*

1. *The set of abstracted legal actions for player i at the root s 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 abstracted legal actions for player i at states s' descendants of s is limited to actions a' that have moves $a'[u]$ fixed to $\bar{\sigma}(s', u)$, for all restricted units u;*

3. *The only abstracted legal action for player $-i$ at any state in the MFT is defined by fixing the move returned by $\bar{\sigma}$ to all units in $\mathcal{U}_{-i}$.*

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). By fixing the opponent's moves to NOKAV we are computing a best response to NOKAV, and in theory, this could make our player exploitable. However,

likely due to the real-time constraints, in practice one tends to derive more effective strategies by fixing the opponent to NOKAV, as mentioned in previous works (Lelis 2017). The development of action abstraction schemes different than using NOKAV for the opponent is an open research question.

Let s_1 and s_2 be the states returned by the transition function $\mathcal{T}$ after applying action a_1 returned by GAB's first step (PGS) and action a_2 returned by GAB's second step (ABCD), respectively, from the state s representing the game's current decision point. GAB returns a_1 if $\Psi(s_1) > \Psi(s_2)$, and a_2 otherwise.

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 $\text{GAB}_\mathcal{P}$ and $\text{SAB}_\mathcal{P}$ that search in uniformly abstracted spaces to compare asymmetric with uniform abstractions.

The difference between GAB and SAB and their variants $\text{GAB}_\mathcal{P}$ and $\text{SAB}_\mathcal{P}$ 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, $\text{GAB}_\mathcal{P}$ and $\text{SAB}_\mathcal{P}$ 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 $\mathcal{M}(s, u', \mathcal{P})$.

$\text{GAB}_\mathcal{P}$ and $\text{SAB}_\mathcal{P}$ 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 $\text{GAB}_\mathcal{P}$ to GAB and $\text{SAB}_\mathcal{P}$ to SAB, which abstraction scheme allows one to derive stronger strategies.

Strategies for Selecting Unrestricted Units

In this section we describe three strategies for selecting the unrestricted units. A selection strategy receives a state s and a set size N and returns a subset of size N of player i's units. The selection of unrestricted units is dynamic as the strategies can choose different unrestricted units at different states. Ties are broken randomly in our strategies.

1. **More attack value (AV+)**. Let $av(u) = \frac{dpf(u)}{hp(u)}$. AV+ selects the N units with the largest av-values, which allows search algorithms to provide finer control to units with low hp-value and/or large dpf-value. This strategy

is expected to perform well as it might be able to preserve longer in the match the units which are about the be eliminated from the match and have good attack power.

2. **Less attack value (AV-)**. AV- selects the N units with the smallest av-values. We expect this strategy to be outperformed by AV+, as explained above.

3. **Random (R)**. R randomly selects N units at s_{init} to be the unrestricted units. R replaces an unrestricted unit that has its hp-value reduced to zero by randomly selecting a restricted unit. This is a domain-independent strategy that in principle could be applied to other multi-unit domains.

Empirical Methodology

We use SparCraft[1] as our testbed, which is a simulation environment of Blizzard's StarCraft. In SparCraft the unit properties such as hit points are exactly the same as StarCraft. However, SparCraft does not implement fog of war, collisions, and unit acceleration (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 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) Dg; (iii) Zl and Dg; (iv) Zl, Dg, and Lg; and (v) 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 the 20 combat configurations used in the experiments. 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 away from the enemy. 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. We use $\mathcal{P} = \{\text{NOKAV}, \text{Kiter}\}$ and a time limit of 40 milliseconds for planning in all experiments.

We use the Ψ function described by Churchill et al. (2012). Instead of evaluating state s directly with LTD2,

---

[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.

our Ψ simulates the game forward from s for 100 state transition steps until reaching a state s'; we then use the LTD2-value of s' as the Ψ-value of s. The game is simulated from s according to the NOKAV script for both players.

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 $\text{GAB}_{\mathcal{P}}$ and $\text{SAB}_{\mathcal{P}}$ 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 $\text{GAB}_{\mathcal{P}}$ and $\text{SAB}_{\mathcal{P}}$ 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 $\text{GAB}_{\mathcal{P}}$ against PGS and $\text{SAB}_{\mathcal{P}}$ against SSS; the detailed results are also omitted for space. The highest winning rate obtained by $\text{GAB}_{\mathcal{P}}$ against PGS was 0.74 while using the AV+ strategy to control 9 units in its ABCD search. The highest winning rate obtained by $\text{SAB}_{\mathcal{P}}$ 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

#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.

ABCD search than $\text{GAB}_{\mathcal{P}}$ and $\text{SAB}_{\mathcal{P}}$ (9 units). This is because GAB and SAB's ABCD search does not restrict the moves of the units, while $\text{GAB}_{\mathcal{P}}$ and $\text{SAB}_{\mathcal{P}}$'s ABCD search does. $\text{GAB}_{\mathcal{P}}$ and $\text{SAB}_{\mathcal{P}}$ 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 approaches with a detailed empirical study.

Asymmetric versus Uniform Abstractions

We test GAB, $\text{GAB}_{\mathcal{P}}$ and PGS (G-Experiment); and SAB, $\text{SAB}_{\mathcal{P}}$, SSS (S-Experiment). GAB, $\text{GAB}_{\mathcal{P}}$, SAB, and $\text{SAB}_{\mathcal{P}}$ use the best performing unrestricted set size N and selection strategies as described above.

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. 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 $\text{GAB}_{\mathcal{P}}$ 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 $\text{GAB}_{\mathcal{P}}$ 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 sub-

stantially 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 $SAB_\mathcal{P}$.

SAB's winning rate over SSS is often larger than GAB's over PGS. For example, in combat scenarios with Zealots only (Zl), GAB's largest winning rate over PGS is 0.81 (with 32 units), which is smaller than the smallest winning of SAB over SSS (0.90 with 50 units). This is likely because SAB's first step (SSS) tends to finish much more quickly than GAB's (PGS). SSS searches for actions for types of units, while PGS searches for actions for units directly, and the number of types tend to be much smaller than the number of units (Lelis 2017). As a result, SAB performs its second step more often than GAB, which allows SAB to derive finer strategies to its unrestricted units in more decision points than GAB. In addition to executing the second step more often, SAB usually allows more computation time for its second step. SAB allowed 32.6 milliseconds of computation time on average for its second step, while GAB allowed 21.8 milliseconds on average for its second step.

Comparison of GAS with GAB and PGS   We also ran experiments comparing GAS with GAB and PGS in combat scenarios containing (i) Zl, (ii) Dg, and (iii) Zl and Dg; we used the same number of units shown in Table 2 for these scenarios. For each combat scenario we ran 1,000 matches. GAS won 55% of the matches against PGS and only 14% against GAB. These results highlight the significance of combining novel search algorithms with asymmetric abstractions. GAS is able to only marginally outperform PGS. By contrast, the two-step scheme used with GAB substantially outperforms both PGS and GAS.

Conclusions and Future Work

We introduced GAB and SAB, two search algorithms that use an abstraction scheme we call asymmetric action abstraction. For not being too restrictive while filtering actions and for assigning finer strategies to a particular subset of units, GAB and SAB are able to substantially outperform the state-of-the-art search-based algorithms for RTS combats. 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 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 *Let Φ be a uniform abstraction and Ω be an asymmetric abstraction, both defined with the same set of scripts $\mathcal{P}$. Also, let $\mathcal{A}'_i(s)$ be the set of actions available at state s according to Φ and $\mathcal{A}''_i(s)$ the set of actions available at s according to Ω. $\mathcal{A}'_i(s) \subseteq \mathcal{A}''_i(s)$ for any s.*

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 $\mathcal{A}'_i(s) \subseteq \mathcal{A}''_i(s)$. $\square$

Let Σ'_i and Σ''_i be the set of player i's strategies whose supports contain only actions in $\mathcal{A}'_i$ and $\mathcal{A}''_i$, respectively. Also, let Σ_{-i} be the set of all player $-i$'s strategies. Lemma 1 allows us to write the following corollary.

Corollary 1 *For abstractions Φ and Ω defined from the same set of scripts $\mathcal{P}$ we have that $\Sigma'_i \subseteq \Sigma''_i$.*

Theorem 1 *Let Φ be a uniform abstraction and Ω be an asymmetric abstraction, both defined with the same set of scripts $\mathcal{P}$. For a finite 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 prove the theorem by induction on the level of the game tree. The base case is given by leaf nodes s_l. Since $V_i^\Omega(s_l) = V_i^\Phi(s_l) = \mathcal{R}_i(s_l)$, the theorem holds. The inductive hypothesis is that $V_i^\Omega(s') \geq V_i^\Phi(s')$ for any state s' at level $j+1$ of the tree. For any state s at level j we have that,

$$V_i^\Omega(s) = \max_{\sigma_i \in \Sigma''_i} \min_{\sigma_{-i} \in \Sigma_{-i}} \sum_{a_i \in \mathcal{A}(s)} \sum_{a_{-i} \in \mathcal{A}_{-i}(s)} \sigma_i(s, a_i) \cdot$$

$$\sigma_{-i}(s, a_{-i}) \cdot V_i^\Omega(\mathcal{T}(s, a_i, a_{-i}))$$

$$\geq \max_{\sigma_i \in \Sigma'_i} \min_{\sigma_{-i} \in \Sigma_{-i}} \sum_{a_i \in \mathcal{A}(s)} \sum_{a_{-i} \in \mathcal{A}_{-i}(s)} \sigma_i(s, a_i) \cdot$$

$$\sigma_{-i}(s, a_{-i}) \cdot V_i^\Phi(\mathcal{T}(s, a_i, a_{-i})) = V_i^\Phi(s).$$

The first equality is the definition of the value of a zero-sum simultaneous move game. The inequality is because $\Sigma'_i \subseteq \Sigma''_i$ (Corollary 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 at level $j+1$ of the tree (inductive hypothesis). The inequality also holds if the transition $\mathcal{T}(s, a_i, a_{-i})$ returns a terminal state z at level $j+1$ as $V_i^\Omega(z) = V_i^\Phi(z) = \mathcal{R}_i(z)$. The last equality is analogous to the first one. $\square$

References

Balla, R.-K., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *Proceedings of the 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*.

Furtak, T., and Buro, M. 2010. On the complexity of two-player attrition games played on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 113–119.

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.

Kovarsky, A., and Buro, M. 2005. Heuristic search applied to abstract combat games. In *Advances in Artificial Intelligence: Conference of the Canadian Society for Computational Studies of Intelligence*, 66–78. Springer.

Lelis, L. H. S. 2017. Stratified strategy selection for unit control in real-time strategy games. In *International Joint Conference on Artificial Intelligence*, 3735–3741.

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. 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.

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.