Asymmetric Action Abstractions for Real-Time Planning in Extensive-Form Games

Rubens O. Moraes, Levi H. S. Lelis
Departamento de Informática
Universidade Federal de Viçosa - Campus Viçosa
Viçosa - Minas Gerais - Brazil
{rubens.moraes,levi.lelis}@ufv.br

***Abstract*—Action abstractions restrict the number of legal actions available for real-time planning in multi-unit zero-sum extensive-form games, thus allowing algorithms to focus their search on a set of promising actions. Even though unabstracted game trees can lead to optimal policies, due to real-time constraints and the tree size, they are not a practical choice. In this context, we introduce an action abstraction scheme we call asymmetric abstraction. Similarly to unabstracted spaces, asymmetrically-abstracted spaces can have theoretical advantages over regularly abstracted spaces while still allowing search algorithms to derive effective strategies in practice, even in large-scale games. Further, asymmetric abstractions allow search algorithms to "pay more attention" to some aspects of the game by unevenly dividing the algorithm's search effort amongst different aspects of the game. We also introduce four algorithms that search in asymmetrically-abstracted game trees to evaluate the effectiveness of our abstraction schemes. An extensive set of experiments in a real-time strategy game developed for research purposes shows that search algorithms using asymmetric abstractions are able to outperform all other search algorithms tested.**

- Student level: MSc.
- Date of conclusion: 08/02/2019.
- Examining board members:
  - Alexandre Santos Brandão (UFV)
  - Leandro Soriano Marcolino (Lancaster University)
  - Levi H. S. de Lelis (UFV)
  - Luiz Chaimowicz (UFMG)
- Dissertation: Clickable Link
- Derived publications:
  - Moraes and Lelis (AAAI'18) [1]
  - Moraes et al. (AIIDE'18) [2]
  - Moraes, Mariño and Lelis (AIIDE'18) [3]
  - Silva, Moraes, Lelis and Gal (TG'2018) [4]
  - Mariño, Moraes, Toledo and Lelis (AAAI'19) [5]
  - Ontañon et al. (AAAI-Magazine'18) [6]

I. Introduction

In real-time strategy (RTS) games the player controls a number 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 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 two-player deterministic games in which all units are visible to both players.

A successful family of algorithms for planning in real time in RTS games uses what we call action abstractions to reduce the number of legal actions available to the player. Action abstractions reduce the number of legal actions a player can perform by reducing the number of legal actions each unit can perform. We use the word "action" if it is clear that we are referring to a player's or to a unit's action; we write "player action" or "unit-action" otherwise. For instance, Churchill and Buro [7] introduced a method for building action abstractions through what they called scripts. A script $\bar{\sigma}$ is a function mapping a game state s and a unit u to an action m for u. A set of scripts $\mathcal{P}$ induces an action abstraction by restricting the set of legal actions of all units to actions returned by the scripts in $\mathcal{P}$. We call an action abstraction generated with Churchill and Buro's scheme a uniform abstraction.

We introduce an action abstraction scheme we call asymmetric action abstractions (or asymmetric abstractions for short). In contrast with uniform abstractions that restrict the number of actions of all units, asymmetric abstractions restrict the number of actions of only a subset of units. We show that asymmetric abstractions can retain the unabstracted spaces' theoretical advantage over uniformly abstracted ones while still allowing algorithms to derive effective strategies in practice, even in large games.

Another contribution we offer is the introduction of four general-purpose algorithms that search in asymmetrically-abstracted trees: Greedy Alpha-Beta Search (GAB), Stratified Alpha-Beta Search (SAB), and two variants of Asymmetrically Action-Abstracted NaïveMCTS, denoted as A2N and A3N. GAB and SAB are based on Alpha-Beta pruning, Portfolio Greedy Search (PGS) [7], and Stratified Strategy Selection (SSS) [8]. PGS and SSS are algorithms designed for searching in uniformly-abstracted spaces. The other two algorithms, A2N and A3N, are based on NaïveMCTS, a search algorithm that uses combinatorial multi-armed bandits (CMAB) [9] to search in unabstracted spaces. In addition to the two variants of NaïveMCTS that search in asymmetrically-abstracted trees, we also introduce a NaïveMCTS baseline that, similarly to PGS and SSS, searches in uniformly-abstracted trees; we call this baseline A1N.

We evaluate our algorithms in μRTS [9], an RTS game developed for research purposes. μRTS is a great testbed for our research because it offers an efficient forward model of the game, which is required by search-based approaches. Finally, μRTS codebase[1] contains most of the current state-of-the-art search-based methods, including the systems used in μRTS's competitions [6], thus facilitating our empirical evaluation. An extensive set of experiments show that our algorithms that search in asymmetrically-abstracted trees are able to outperform in terms of matches won their counterparts that search unabstracted and uniformly-abstracted trees.

Although we present our abstraction schemes and search algorithms in the context of RTS games, we believe our ideas and algorithms to also be applicable in other scenarios. For example, a robotic system, which controls several actuators simultaneously while trying to accomplish a task, can benefit from asymmetric abstractions. This is because some actuators might require a finer control than the others. To illustrate, the actuators controlling the arms of a robot planning a sequence of actions to open a door might need a "finer plan" than the actuators controlling the wheels of the robot, given that the robot is already in front of the door to be opened. Asymmetric action abstractions offer an approach that allows the planning system to focus on the arms of the robot rather than on its wheels. As another example, a search algorithm for uniformly-abstracted trees is the core of the artificial player of the commercial card game Prismata [10]. The idea we introduce in this paper of performing search in asymmetric trees can also be used in such card games to enhance the strength of their artificial player. For example, the artificial player could benefit from an algorithm that discovers finer plans for the "more important" cards.

This paper is organized as follows. In Section II we define RTS games as an extensive-form game and review search algorithms used for planning in unabstracted spaces. Next, in Section III, we describe the asymmetric abstractions we introduce in this paper. Section IV introduces the four novel algorithms for searching in asymmetrically-abstracted action spaces. In Section V, we evaluate the introduced algorithms with an extensive set of experiments on μRTS. Finally, in Section VI, we present the conclusions.

II. Background

RTS games can be described as zero-sum extensive-form games with simultaneous and durative actions, defined by a tuple $\nabla = (\mathcal{N}, \mathcal{S}, s_{init}, \mathcal{A}, \mathcal{B}, \mathcal{R}, \mathcal{T})$, where, $\mathcal{N} = \{i, -i\}$ is the pair 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**, i.e., states where the game has finished and no more actions can be taken. Every state $s \in \mathcal{S}$ defines a joint set of **units** $\mathcal{U}^s = \mathcal{U}^s_i \cup \mathcal{U}^s_{-i}$, for players i and $-i$. $\mathcal{A} = \mathcal{A}_i \times \mathcal{A}_{-i}$ 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-actions** $(m_1, \cdots, m_n)$, where $m_k \in a$ is the action of the k-th **ready unit** of player i. A unit is not ready if it is already performing an action (unit-actions can have different durations). We denote the set of ready units of players i and $-i$ as $\mathcal{U}^s_{i,r}$ and $\mathcal{U}^s_{-i,r}$. We denote the set of unit-actions as $\mathcal{M}$. We write $\mathcal{M}(s, u)$ to denote the set of legal actions 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 $s \in \mathcal{F}$. The **transition function** $\mathcal{T} : \mathcal{S} \times \mathcal{A}_i \times \mathcal{A}_{-i} \to \mathcal{S}$ deterministically determines the sucessor state for a state s and the set of joint actions taken at s.

The **game tree** of ∇ is a tree rooted at s_{init} in which each node represents a state 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 node representing s_k to the node representing s_j in the game tree 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 trees to be finite and denote as Ψ the **evaluation function** used by algorithms while traversing the tree. Ψ receives a state s and returns an estimate of the end-game value of s for player i. Since ∇ is zero sum, i tries to reach nodes in the tree that maximizes Ψ, while $-i$ tries to reach nodes that minimizes Ψ.

We call a **decision-point** of player j a state s in which j has at least one ready unit; s is called a **within-state** of player j otherwise. In this paper, the search algorithm controlling the units of a player is invoked at every node n of the game tree, independently if n represents a decision-point or within-state of player j. We show in Section IV-A1 how search algorithms can use the time available for search at within-states to plan ahead.

A. Search Algorithms for Unabstracted Trees

In this subsection we review two search algorithms used for planning in RTS games with unabstracted game trees: Alpha-Beta Considering Durations (ABCD) and Naïve Monte Carlo Tree Search (NaïveMCTS).

1) Alpha-Beta Considering Durations (ABCD): Minimax search with Alpha-Beta pruning [11] has been successfully applied to games such as Chess [12]. The key idea of Alpha-Beta is to compute the value of a game while pruning branches of the tree that are not reached in optimal play.

Knuth and Moore [11] showed how Alpha-Beta can be adapted to find an approximate solution for simultaneous-move games. Once the Alpha-Beta search reaches a node in the game tree in which both players act simultaneously, a policy π decides who acts first, with the other player choosing their action afterwards. The policy π transforms a simultaneous-move game into a sequential-move game, for which Alpha-Beta is suitable. The solution encountered in the transformed sequential-move game can then be applied as an approximation to the original game. Churchill et al. [13] showed that Alpha-Beta with the alternate policy defeats the same algorithm with the random policy in RTS combats.

Alpha-Beta, shown in Algorithm 1, is used in an iterative-deepening manner, with the value of d passed as input to the

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

Algorithm 1 ALPHA-BETA

Require: State s, depth d, $\alpha = -\infty$, $\beta = \infty$, evaluation function Ψ.
Ensure: An approximation of the game value of s.

```
1: if s ∈ F or d = 0 then
2:     return Ψ(s)
3: j ← B(s)
4: if j = −i then
5:     M ← ∞
6:     for each a ∈ A_{−i}(s) do
7:         M ← min(ALPHA-BETA(T(s, a), d − 1, α, β), M)
8:         if M ≤ α then
9:             return M
10:        β = min(β, M)
11: if j = i then
12:    M ← −∞
13:    for each a ∈ A_i(s) do
14:        M ← max(ALPHA-BETA(T(s, a), d − 1, α, β), M)
15:        if M ≥ β then
16:            return M
17:        α = max(α, M)
18: return M
```

first call of Alpha-Beta set to 1; d is incremented by one if the execution of Algorithm 1 finishes and there is still time available for planning. Algorithm 1 is then invoked again with the incremented value of d. Variables α and β store the best values that can be achieved by players i and $-i$, respectively.

The Ψ-value of a node is returned if the search reaches its maximum depth ($d = 0$ as d is decremented in each recursive call) or if the node is terminal (line 2). Variable j stores the player acting in the current state (line 3), which is either i or $-i$ for sequential-move games. If it is $-i$'s turn (lines 4–10), then Alpha-Beta searches for all possible transitions from state s, which is given by the actions in $\mathcal{A}_{-i}$. In the recursive call of line 7, the transition function $\mathcal{T}$ takes two arguments: the current state s and action a. This is because in Algorithm 1 we assume sequential games, thus the transition function depends on the action of only one player. The search is pruned in $-i$'s turn (lines 8 and 9) if the current lower bound for i's solution value (given by α) is at least as large as the current upper bound for $-i$'s solution (given by M). If the expression $M \leq \alpha$ in line 8 is true, then it means that player i prefers to choose an earlier action in the game tree that guarantees i a game value of α, than allow $-i$ to reach the current node of the tree, which is guaranteed to be at most as good as α. The same reasoning applies in lines 11–17, where player i is to act, instead of player $-i$.

Algorithm 1 assumes sequential-moves games. Churchill et al. [13] introduced Alpha-Beta Considering Durations (ABCD), an algorithm that accounts for simultaneous-moves.

We need to perform two modifications to transform Alpha-Beta into ABCD. First, we replace the assignment $j \leftarrow \mathcal{N}(s)$ by $j \leftarrow \pi(s)$ (line 3), where π is the policy that decides the player who is to choose their action first. For states s where $\mathcal{N}(s) = \{i\}$ or $\mathcal{N}(s) = \{-i\}$, then $\pi(s) = \mathcal{N}(s)$. If $\mathcal{N}(s) = \{i, -i\}$, then π decides which player acts first. The implementation we use employs the alternate policy: π returns i if it returned $-i$ in the previous state in which both players acted simultaneously. π randomly chooses either i or $-i$ for the first state in the tree with simultaneous actions.

The second modification deals with the "delayed effects" of an action. ABCD handles this as follows. In states with simultaneous actions, the transition function is not applied during the function's recursive calls, as shown in lines 7 and 14 of Algorithm 1. Instead, we pass as parameters the current state s and the action a the player is going to perform at s. This way the set of actions of the other player can be computed within the next function call and only then a is applied to s. The ABCD implementation we use in our experiments employ a transposition table to avoid expanding multiple paths leading to the same state [14], [15].

2) Naïve Monte Carlo Tree Search (NaïveMCTS): Ontañón [9] modeled the search problem of deriving strategies in RTS games as a combinatorial multi-armed bandits (CMAB) problem. A CMAB problem can be defined by a tuple (X, μ), where,

- $X = \{X_1, \cdots, X_n\}$, where each X_i is a variable that can assume K_i different values $\mathcal{X}_i = \{v_i^1, \cdots, v_i^{K_i}\}$, with $\mathcal{X} = \{(v_1, \cdots, v_n) \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_n\}$ being the possible combinations of value assignments for the variables in X; a value assignment $V \in \mathcal{X}$ is called a macro-arm.
- $\mu : \mathcal{X} \rightarrow \mathbb{R}$ is a reward function, that receives a macro-arm and returns a reward value for that macro-arm.

The goal in a CMAB problem is to find a macro-arm that maximizes the expected reward. This can be achieved by balancing exploration and exploitation until converging to an optimal macro-arm. In the context of RTS games, each decision-point s can be cast as a CMAB problem in which X contains one variable for each ready unit of a player in s. Thus, a macro-arm $V \in \mathcal{X}$ represents a player action and each value $v \in V$ represents a unit-action. The set $\mathcal{X}_i = \{v_i^1, \cdots, v_i^{K_i}\}$ represents the set of K_i legal actions for the i-th unit at s. Naturally, the goal is to find a macro-arm (player action) that maximizes the player's reward, which is defined by an evaluation function.

Since the number of macro-arms in $\mathcal{X}$ is often too large in RTS games, Ontañón [9] derived a sampling procedure called Naïve Sampling (NS) to help deciding which macro-arms should be evaluated during search. NS divides a CMAB problem with n variables into $n+1$ multi-armed bandit (MAB) problems:

- n local MABs, one for each variable $X_i \in X$. That is, for variable X_i representing the i-th unit, the arms of the MAB are the K_i values (unit-actions) in $\mathcal{X}_i$.
- 1 global MAB, denoted MAB_g, that treats each macro-arm V considered by NS as an arm in MAB_g. Naturally, MAB_g has no arms in the beginning of NS's procedure.

At each iteration, NS uses a policy π_0 to determine whether it adds an arm to MAB_g through the local MABs (explore) or evaluates an existing arm in MAB_g (exploit).

1) If explore is chosen, then a macro-arm V is added to MAB_g by using a policy π_l to independently choose a value for each variable in X. Here, NS assumes that the reward of a macro-arm V can be approximated by

Algorithm 2 NAÏVEMCTS

Require: State s, sampling strategies π_0, π_l and π_g, and evaluation function Ψ.
Ensure: Action a
1: root ← node(s)
2: **while** hasTime() **do**
3: leaf ← SELECTANDEXPANDNODE(root, π_0, π_l, π_g)
4: $v \leftarrow \Psi$(leaf.state)
5: PROPAGATEEVALUATION(leaf, v)
6: **return** GETMOSTVISITEDACTION(root)

Algorithm 3 SELECTANDEXPANDNODE

Require: A game tree node n_0 and sampling strategies π_0, π_l and π_g
Ensure: A node in the tree
1: $j \leftarrow \pi(n_0.state)$
2: $n \leftarrow$ NS($n_0.state$, π_0, π_l, π_g, j)
3: **if** $n \in n_0.children$ **then**
4: **return** SELECTANDEXPANDNODE($n_0.child(\alpha)$)
5: **else**
6: n_0.addChild(n)
7: **return** return n

the sum of the rewards of the individual values $v_i \in V$, denoted $\mu'(v_i)$. That is, $\mu(V) \approx \sum_{v_i \in V} \mu'(v_i)$.

2) If exploit is chosen, then a policy π_g is used to select an existing macro-arm in MAB_g.

Ontañón [9] showed that NS can be used in the context of Monte Carlo Tree Search (MCTS) by introducing an algorithm named NaïveMCTS (see Algorithm 2). NaïveMCTS differs from other MCTS algorithms in that it uses NS to decide which player actions should be evaluated in search. Instead of uniformly sampling which player action to evaluate next as in a vanilla MCTS algorithm, NaïveMCTS uses NS to select player actions composed of unit-actions that tend to yield good rewards.

NaïveMCTS expands a tree in its search procedure, which we will refer to as the MCTS tree. The MCTS tree starts only with the root node representing the current state of game (line 1). While there is time allowed for planning, NaïveMCTS iteratively selects a node to be added to the MCTS tree, through a call to SELECTANDEXPANDNODE (line 3), which is described in Algorithm 3.

Since SELECTANDEXPANDNODE uses the NS procedure described above, in addition to the root of the tree, it requires as input the policies π_0, π_l, and π_g. Similarly to ABCD, NaïveMCTS uses a policy π to sequentialize simultaneous-move states by allowing one of the players to decide their action before the other player (line 1 of Algorithm 3). The NS procedure is invoked to decide which player action will be explored. If the node n returned by NS is part of the MCTS tree (line 3), NS chooses to exploit and SELECTANDEXPANDNODE is called recursively to select a player action from n. Otherwise, NS chooses to explore, and a new macro-arm is chosen, leading to a state n that is not in the MCTS tree. In this case, n is added to the MCTS tree (line 6) and is returned to NaïveMCTS's main procedure (Algorithm 2).

Once NaïveMCTS (Algorithm 2) runs out of time, it returns the most visited player action from *root.state*, as the actions with largest estimated utility for the player are visited more often. The most visited action is returned by GETMOSTVISITEDACTION (line 6).

Algorithm 4 Portfolio Greedy Search

Require: state s, default script $\bar{\sigma}_d$, set of scripts $\mathcal{P}$, time limit e, and evaluation function Ψ.
Ensure: action a for player i's units.
1: $\bar{\sigma}_i \leftarrow$ choose a script from $\mathcal{P}$ considering that $-i$ acts according to $\bar{\sigma}_d$
2: $\bar{\sigma}_{-i} \leftarrow$ choose a script from $\mathcal{P}$ considering that i acts according to $\bar{\sigma}_i$
3: $a_i \leftarrow \{\bar{\sigma}_i(u_1), \bar{\sigma}_i(u_2), \cdots, \bar{\sigma}_i(u_n)\}$, where $u_1, u_2, \cdots, u_n \in \mathcal{U}^s_{i,r}$
4: $a_{-i} \leftarrow \{\bar{\sigma}_{-i}(u_1), \bar{\sigma}_{-i}(u_2), \cdots, \bar{\sigma}_{-i}(u_m)\}$, where $u_1, u_2, \cdots, u_m \in \mathcal{U}^s_{-i,r}$
5: **while** time elapsed is not larger than e **do**
6: **for** each $u \in \mathcal{U}^s_{i,r}$ **do**
7: **for** each $\bar{\sigma} \in \mathcal{P}$ **do**
8: $a'_i \leftarrow a_i$; $a'_i[u] \leftarrow \bar{\sigma}(s, u)$
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 e **then**
12: **return** a_i
13: **return** a_i

B. Search Algorithms for Uniformly-Abstracted Trees

In this section we present the uniform abstractions and two algorithms used for searching in uniformly-abstracted trees, Portfolio Greedy Search (PGS) and Stratified Strategy Selection (SSS).

1) Uniform Action Abstractions: We define a **uniform action abstraction** (or uniform abstraction for short) for player i as a function mapping the set of legal actions $\mathcal{A}_i$ to a subset $\mathcal{A}'_i$ of $\mathcal{A}_i$. Action abstractions can be constructed from a set of scripts $\mathcal{P}$. Let the action-abstracted legal actions of unit u at state s be the actions 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 that receives 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 actions in $\mathcal{M}(s, u, \mathcal{P})$ for all u in $\mathcal{U}^s_{i,r}$, where $\mathcal{U}^s_{i,r}$ is the set of ready units of i in s.

Algorithms using a uniform abstraction 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 unit-actions returned by the scripts in $\mathcal{P}$.

2) Portfolio Greedy Search: Churchill and Buro [7] introduced Portfolio Greedy Search (PGS), a hill-climbing search procedure for uniformly-abstracted trees. Algorithm 4 shows the pseudocode of PGS.

PGS starts by selecting the script $\bar{\sigma}_i$ from $\mathcal{P}$ that yields the largest Ψ-value when i executes an action composed of unit-actions computed with $\bar{\sigma}_i$, assuming that $-i$ executes an action composed of unit-actions computed with $\bar{\sigma}_d$ (line 1). The same process is executed to select $\bar{\sigma}_{-i}$, considering that i executes unit-actions computed by $\bar{\sigma}_i$ (line 2). Action vectors a_i and a_{-i} are initialized with the unit-actions computed from $\bar{\sigma}_i$ and $\bar{\sigma}_{-i}$. Once a_i and a_{-i} have been initialized, PGS iterates through all units u in $\mathcal{U}^s_{i,r}$ and tries to greedily improve the

Algorithm 5 Stratified Strategy Selection

Require: state s, default script $\bar{\sigma}_d$, set of scripts $\mathcal{P}$, time limit e, evaluation function Ψ, and a type system T for the set of units $\mathcal{U}_i$ in s.
Ensure: action a for player i's units, boolean c indicating if the algorithm finished a complete iteration over all types in T

1: $\bar{\sigma}_i \leftarrow$ choose a script from $\mathcal{P}$ considering that $-i$ acts according to $\bar{\sigma}_d$
2: $\bar{\sigma}_{-i} \leftarrow$ choose a script from $\mathcal{P}$ considering that i acts according to $\bar{\sigma}_i$
3: $a_i \leftarrow \{\bar{\sigma}_i(u_1), \bar{\sigma}_i(u_2), \cdots, \bar{\sigma}_i(u_n)\}$, where $u_1, u_2, \cdots, u_n \in \mathcal{U}^s_{i,r}$
4: $a_{-i} \leftarrow \{\bar{\sigma}_{-i}(u_1), \bar{\sigma}_{-i}(u_2), \cdots, \bar{\sigma}_{-i}(u_m)\}$, where $u_1, u_2, \cdots, u_m \in \mathcal{U}^s_{-i,r}$
5: $c \leftarrow$ false
6: **while** time elapsed is not larger than e **do**
7: **for** each $t \in T$ **do**
8: **for** each $\bar{\sigma} \in \mathcal{P}$ **do**
9: $a'_i \leftarrow a_i$ with the actions of all units u of type t replaced by $\bar{\sigma}(u)$
10: **if** $\Psi(\mathcal{T}(s, a'_i, a_{-i})) > \Psi(\mathcal{T}(s, a_i, a_{-i}))$ **then**
11: $a_i \leftarrow a'_i$
12: **if** time elapsed is larger than e **then**
13: return a_i and boolean c
14: $c \leftarrow$ true *// iterated over all types*
15: return a_i and boolean c

unit-action assigned to u in a_i, denoted by $a_i[u]$. Note that PGS only considers the unit-actions in the uniform abstraction, i.e., those in $\mathcal{M}(s, u, \mathcal{P})$. PGS evaluates a_i while replacing $a_i[u]$ by each of the possible unit-actions m for u. PGS keeps in a_i the action vector found during search with the largest Ψ-value. This process is repeated until PGS reaches time limit e and returns a_i (see lines 12 and 13).

Note that the action vector a_{-i} remains unchanged after its initialization (see line 4). Although in its original formulation PGS alternates between improving player i's and player $-i$'s action vectors [7], in their experiments, Churchill and Buro allow PGS to improve only player i's action vector while player $-i$'s is fixed. Moraes et al. [3] showed that, if set to improve both a_i and a_{-i}, PGS can suffer from a pathological issue they called the non-convergence problem. Due to the non-convergence problem, PGS can find worse strategies than PGS with a_{-i} fixed, even if the former is granted more computation time than the latter. In addition to identifying the pathology, Moraes et al. introduced an algorithm called Nested-Greedy Search (NGS) that alters both a_i and a_{-i} while not suffering from the pathology. We use PGS with a_{-i} fixed instead of NGS because NGS was shown to not scale well to large games [3].

3) Stratified Strategy Selection: Lelis [8] introduced Stratified Strategy Selection (SSS). Similarly to PGS, SSS performs a hill-climbing search. However, in contrast with PGS, SSS searches in the space of script assignments induced by a **type system**, which is a partition of units. SSS assigns the same script to units of the same type. For example, all units with low hit point values (type) move away from the battle (strategy of a script). A type system is defined as follows.

Definition 2 (Type System): Let $\mathcal{U}_i$ be the set of player i's units. $T = \{t_1, \ldots, t_k\}$ is a type system for $\mathcal{U}_i$ if it is a partitioning of $\mathcal{U}_i$. If $u \in \mathcal{U}_i$ and $t \in T$ with $u \in t$, we write $T(u) = t$.

Algorithm 5 shows the pseudocode of SSS. In implementation SSS also performs the seeding process described above for PGS. That is, SSS starts by selecting the script $\bar{\sigma}_i$ from $\mathcal{P}$ that yields the largest Ψ-value when i executes an action composed of unit-actions computed with $\bar{\sigma}_i$, assuming that $-i$ executes an action composed of unit-actions computed with the default script $\bar{\sigma}_d$ (line 1). The same process is executed to select $\bar{\sigma}_{-i}$, considering that i executes unit-actions computed by $\bar{\sigma}_i$ (line 2). SSS initializes actions a_i and a_{-i} with the unit-actions returned by the the scripts $\bar{\sigma}_i$ and $\bar{\sigma}_{-i}$. SSS then performs a greedy search to improve the unit-actions in a_i (lines 6–11). Namely, SSS evaluates all possible assignments of unit-actions according to the scripts in $\mathcal{P}$ to units of a given type t while the unit-actions of units with types other than t are fixed. SSS keeps in a_i the unit-actions with largest Ψ-value encountered during search (lines 10 and 11). SSS returns the player action a_i once it reaches the time limit e (lines 13 and 15). The boolean value c is used by SSS with Adaptive Type Systems (SSS+). Similarly to PGS, the action a_{-i} is fixed throughout search and SSS tries to approximate a best response to the opponent's action given by script $\bar{\sigma}_{-i}$.

Depending on the number and on the diversity of units present in the match, SSS might be unable to iterate through all types in T before reaching time limit e. Aiming at preventing SSS from not iterating at least once over all types, Lelis [8] developed a meta-reasoning system to adjust the granularity of the type system used. This adjustment occurs in between searches and is based on the estimated running time of a SSS iteration.

III. Asymmetric Action Abstractions

Uniform abstractions are restrictive in the sense that all units have their legal actions reduced to those specified by scripts. We introduce an abstraction scheme we call **asymmetric action abstractions** (or asymmetric abstractions for short) 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 actions of only a subset of the units controlled by player i; the sets of legal actions of the other units remain unchanged. We call the subset of units that do not have their set of legal actions reduced the **unrestricted units**; the complement of the unrestricted units are defined as the **restricted units**.

Definition 3: An asymmetric abstraction Ω is a function receiving as input a state s, a player i, a set of unrestricted units $\mathcal{U}'_i \subseteq \mathcal{U}^s_{i,r}$, 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 unit-actions in $\mathcal{M}(s, u, \mathcal{P})$ for all u in $\mathcal{U}^s_{i,r} \setminus \mathcal{U}'_i$ and of unit-actions $\mathcal{M}(s, u')$ 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. Asymmetric abstractions allow us to explore action abstractions in the spectrum of possibilities between the uniformly-abstracted and unabstracted game trees.

Asymmetric abstractions allow search algorithms to divide its "attention" differently among the units at a given state of the game. That is, depending on the game state, some units might

be more important than others (e.g., units with low hit points trying to survive), and asymmetric abstractions allow one to derive finer strategies to these units by accounting for a larger set of unit-actions for them. Similarly, a robotic control system might benefit from deriving finer plans to actuators that are more important at given state of the world. For example, the actuators controlling the arms of a robot trying to open a door are more likely to benefit from a finer plan than the actuators controlling the wheels of the robot. Uniform abstractions does not allow the planning system to "pay more attention" to specific parts of the system as they divide the search effort equally amongst all parts.

In addition to the ability of unevenly dividing the search effort amongst different parts of a system at a given state, the optimal strategy derived from an asymmetrically-abstracted tree is guaranteed to be no worse than the strategies derived from a uniformly-abstracted tree for a game ∇, given that both abstractions are induced 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 zero-sum extensive-form game ∇ with start state s, let $V_i^{\Phi}(s)$ be the optimal value of the game computed by considering the game tree 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 [1] 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 handcrafted or learned. By contrast, one can easily create a wide range of asymmetric abstractions by simply modifying the set of unrestricted units. Further, in contrast with asymmetric abstractions, such an enlargement scheme would not allow one to directly focus on important parts of the system.

Although Theorem 1 guarantees that one is able to derive strategies with asymmetric abstractions that are no worse than those derived with uniform abstractions, our theory is not necessarily a good predictor of what happens in practice. This is because Theorem 1 assumes that optimal strategies can be derived from the abstracted game trees and in practice we use search algorithms to only approximate optimal solutions in real time.

In the next section we introduce four novel algorithms that search in real time in asymmetrically-abstracted game trees.

IV. Searching in Asymmetrically-Abstracted Game Trees

We introduce Greedy Alpha-Beta Search (GAB) and Stratified Alpha-Beta Search (SAB), two algorithms for searching in asymmetrically-abstracted trees. 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 loops in Algorithms 4 and 5 without altering a_i. Once a local maximum is reached, PGS and SSS are unable to further improve the unit-action assignments, even if the time limit e 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 tree 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 tree for moves for all unrestricted units.

A. Greedy and Stratified Alpha-Beta Searches (GAB and SAB)

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 actions of the restricted units during the second search. In its second step, GAB uses a variant of ABCD. Although we use ABCD, one could also use other search algorithms such as UCTCD [7]. 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 $\mathcal{U}^s_{i,r} = \{u_1, u_2, u_3\}$ 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. Let $a = (W, L, R)$ be the player action returned by PGS, where W, L, R are the unit-actions 'wait' (W), 'move left' (L), and 'move right' (R). Also, let $\varsigma = \{\bar{\sigma}_1, \bar{\sigma}_2, \bar{\sigma}_1\}$ be the script vector that defined the unit-actions during PGS's search. That is, $\bar{\sigma}_1(s, u_1) = W$, $\bar{\sigma}_2(s, u_2) = L$, and $\bar{\sigma}_1(s, u_3) = R$. We use the notation $\varsigma[u_1]$ to denote the script in ς used to define the unit-action for unit u_1 in PGS's search.

GAB's second step searches in the `MFT`. The `MFT` is rooted at s, and the set of abstracted legal player actions in s is obtained by fixing $a[u_2] = L$ and considering all legal actions for 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 player actions in s is: $\{(W, L, R), (W, L, D), (U, L, R), (U, L, D)\}$.

For player i and for all descendants states s' of s in the `MFT`, if $\mathcal{M}(s', u_1) = \{W, U\}$, $\mathcal{M}(s', u_3) = \{R, D\}$, and $\varsigma[u_2] = \bar{\sigma}_2$, then the set of abstracted legal actions in s' is:

$$\{(W, \bar{\sigma}_2(s', u_2), R), (W, \bar{\sigma}_2(s', u_2), D),$$
$$(U, \bar{\sigma}_2(s', u_2), R), (U, \bar{\sigma}_2(s', u_2), D)\}.$$

That is, at states s' we consider all legal unit-actions of the unrestricted units and we fix the unit-actions of the restricted units to what is returned by the units' script in ς.

Also, for all descendants states s' of s in the `MFT`, if player $-i$'s ready units in s are $\mathcal{U}^s_{-i,r} = \{u_1, u_2, u_3, u_4\}$, the set of abstracted legal player actions for $-i$ in s' is,

$$\{(\bar{\sigma}_{-i}(s', u_1), \bar{\sigma}_{-i}(s', u_2), \bar{\sigma}_{-i}(s', u_3), \bar{\sigma}_{-i}(s', u_4)\}.$$

Here, $\bar{\sigma}_{-i} \in \mathcal{P}$ is the script computed in PGS's seeding process (see line 2 of Algorithm 4).

Definition 4 (Move-Fixed Tree): For a given state s, a subset of unrestricted units of $\mathcal{U}_i$ in s, a set of scripts $\mathcal{P}$, the script $\bar{\sigma}_{-i} \in \mathcal{P}$ defined in the seeding process of the algorithm's first step, a player action a returned by the algorithm's first step, and the script vector ς with one script for each u in $\mathcal{U}^s_{i,r}$ used by the algorithm's first step to define the unit-actions in a, 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 unit-actions $a'[u]$ 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 unit-actions $a'[u]$ fixed to $\bar{\sigma}(s', u)$ with $\bar{\sigma} = \varsigma[u]$, for all restricted units u;
3) The only abstracted legal action for player $-i$ at any state in the MFT is defined by fixing player $-i$'s unit-actions to those returned by $\bar{\sigma}_{-i}$.

By searching in the MFT, ABCD searches for actions for the unrestricted units while the actions of all other units, including the opponent's units, are fixed: player i's restricted units act according to the scripts in ς and player $-i$'s units act according to $\bar{\sigma}_{-i}$. Our two-step search approximates a best response to the strategy defined by the script $\bar{\sigma}_{-i}$. In theory, this approach could make our player exploitable. However, in practice, due to the real-time constraints, one tends to derive more effective strategies by fixing the opponent strategy, as shown in previous works [3].

Let a_1 be the player action returned by PGS in GAB's first step and a_2 be the player action returned by ABCD in GAB's second step. Also, let a' be the opponent action defined by using the script $\bar{\sigma}_{-i}$ for all opponent's units. Instead of returning a_2 directly, GAB returns the action with largest Ψ value, i.e., $\arg\max_{a \in \{a_1, a_2\}} \Psi(\mathcal{T}(s, a, a'))$. Note that one cannot compare the evaluation value of actions a_1 and a_2 as computed by ABCD and PGS. This is because ABCD performs a depth-first search and uses the Ψ function to evaluate the leaf nodes of the tree expanded by ABCD; these values are then propagated up the tree to evaluate the actions available at the root. As a result, the evaluation values of the actions at the root performed by ABCD are based on nodes deeper into the tree than the evaluation performed by PGS. Thus, once GAB has a_1 and a_2, it evaluates them again with the same Ψ function, and only then it selects the best of the two actions.

The difference between SAB and GAB is the algorithm used in their first step: while GAB uses PGS, SAB uses SSS. The second step of SAB follows exactly GAB's second step.

1) Searching in Within-States: The transition function $\mathcal{T}$ receives as input the current state, a set of joint player actions and returns the next state in the game. Since the unit-actions are durative, the state s' returned by $\mathcal{T}$ might not be a decision-point for either of the players (i.e., none of the players have a ready unit). We call the states s that are not a decision-point for player i a **within-state** for i. In RTS games, within-states are those in which the game shows an animation of the units performing their actions (e.g., a worker building a structure) and the player cannot issue an action. More generally, within-states occur when an agent is performing a predefined sequence of actions—e.g., a macro-action [16]–[18] or an option [19], [20]—and for that the agent is not ready to perform another action.

GAB and SAB are implemented to take advantage of within-states. This is achieved by performing the first step of GAB and SAB in within-states and the second step in decision-points. While the agent is performing a sequence of actions (e.g., unit building a structure) at within-states s' for player i, GAB and SAB use the game model to fast forward from s' to a decision-point s for i. Since one might encounter decision-points for player $-i$ while fast forwarding, GAB and SAB assume the opponent follows the strategy given by the script selected to initialize the opponent's action in PGS and SSS, $\bar{\sigma}_{-i}$. GAB and SAB's first step is then performed at s and its result stored in memory. Later, if s is reached in the actual game, then GAB and SAB performs only their second step, searching for the unrestricted units with ABCD; the restricted units perform the action stored in memory for s.

State s might not be reached in the actual game as the opponent might choose actions that are different from those returned by the opponent model defined in the first step of search. If instead of reaching s, the players reach a game state s'' and the set of units at s and s'' are identical for player i, i.e., $\mathcal{U}^s_i = \mathcal{U}^{s''}_i$, then GAB and SAB performs only the second step, leaving the actions of the restricted units fixed, as described above. However, if $\mathcal{U}^s_i \neq \mathcal{U}^{s''}_i$, then GAB and SAB perform both the first and second steps at s''. We show empirically in Section V-A the effectiveness of within-states search in practice.

2) Baselines for GAB and SAB: GAB$_\mathcal{P}$, SAB$_\mathcal{P}$, GAS, and SAS: In this subsection we introduce four baseline algorithms for GAB and SAB. The goal of introducing these baselines is to show empirically the advantages of (i) searching in asymmetrically-abstracted trees and (ii) of searching with a two-step scheme. Similarly to GAB and SAB, all four baselines benefit from searching in within-states.

a) GAB$_\mathcal{P}$ and SAB$_\mathcal{P}$: In contrast with GAB and SAB, GAB$_\mathcal{P}$ and SAB$_\mathcal{P}$ only account for unit-actions in $\mathcal{M}(s, u, \mathcal{P})$ for all s and u in their ABCD search. That is, GAB$_\mathcal{P}$ and SAB$_\mathcal{P}$ only consider actions a' for which the unit-actions $a'[u]$ for restricted units u are fixed (as in GAB's and SAB's MFT) and the unit-actions $a'[u']$ 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 $\mathcal{U}'$. 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.

b) GAS and SAS: The difference between GAS and PGS is that in the greedy search of the former, for a given state s, instead of limiting the number of legal actions of all units u to $\mathcal{M}(s, u, \mathcal{P})$, as PGS does, GAS considers all legal actions $\mathcal{M}(s, u)$ for unrestricted units, and the actions $\mathcal{M}(s, u, \mathcal{P})$ for restricted units. While PGS searches in a uniformly-abstracted

tree, GAS searches in an asymmetrically-abstracted tree. The difference between SAS and SAB is twofold. First, similarly to the difference between GAS and PGS, SAS also accounts for actions $\mathcal{M}(s, u)$ for all unrestricted units u. Second, in the type system T used by SAS, all unrestricted units u have their own type, i.e., $T(u) \neq T(u')$ for all unrestricted units u and all units u'. This second modification is needed to guarantee that SAS is similar to SSS in that units of the same type execute the action returned by the same script.

B. Asymmetrically Action-Abstracted NaïveMCTS (A3N)

We call Asymmetrically Action-Abstracted NaïveMCTS (A3N) the version of NaïveMCTS that accounts during search for all unit-actions of the unrestricted units and only for the actions returned by the set of scripts $\mathcal{P}$ for the restricted units.[2] The only different between NaïveMCTS and A3N is that in the latter, the NaïveSampling procedure (see call to NS in Algorithm 3) can only sample macro-arms that are in the asymmetrically-abstracted tree.

Similarly to the search algorithms discussed above, A3N can also benefit from searching within-states. This is achieved by maintaining in memory the parts of the tree expanded by A3N that are still reachable in the game. The time allowed for planning using the within-states will serve to explore more macro-arms and search deeper into the tree to obtain more accurate estimates of the end-game values of the macro-arms.

1) Baselines for A3N: A1N and A2N: Similarly to the baselines introduced for GAB and SAB, we introduce two baselines for A3N: A1N and A2N. Both are based on NaïveMCTS, with the former searching in uniformly-abstracted trees and the latter searching in asymmetrically-abstracted trees. The goal of introducing these baselines is to allow us to evaluate empirically the effectiveness of the asymmetric abstractions introduced above for A3N in comparison to uniform abstractions induced by $\mathcal{P}$ and asymmetric abstractions induced by two sets of scripts. A1N and A2N also use the A3N enhancement of reusing the search tree expanded in previous states.

a) A1N: We call A1N a version of NaïveMCTS that uses an action abstraction induced by $\mathcal{P}$. The difference between NaïveMCTS and A1N is in the unit-actions sampled by NS while adding macro-arms to MAB_g. Instead of being able to sample from all legal unit-actions, A1N's is allowed to sample only from $M(s, u, \mathcal{P})$ for all units u. As a consequence, the macro-arms added to MAB_g are restricted to the unit-actions returned by the scripts.

b) A2N: We call A2N the version of NaïveMCTS that uses an action abstraction defined by two sets of scripts: $\mathcal{P}'$ and $\mathcal{P}''$. A2N divides the set of units into two subsets: the units related to $\mathcal{P}'$ and the units related to $\mathcal{P}''$. A2N can only sample unit-actions m for the units u in the first group if m is returned by one of the scripts in $\mathcal{P}'$ for u. The unit-actions A2N can sample for the second group of units is defined analogously. Note that the two subsets of units do not need to be disjoint as some units can have actions sampled from both sets of scripts.

[2]The number '3' in A3N is the version of the algorithm; versions 1 and 2 (A1N and A2N) are described in Section IV-B1.

TABLE I
MAPS USED IN OUR EXPERIMENTS. THE NAMES ARE AS THEY APPEAR IN THE μRTS CODEBASE. WE ALSO SHOW THE SIZE OF THE MAPS AND THE MAXIMUM NUMBER OF GAME CYCLES FOR MATCHES PLAYED IN EACH MAP.

Map Name	Size	Number of Cycles
`basesWorkers8x8A`	8×8	3,000
`FourBasesWorkers8x8`	8×8	3,000
`basesWorkers16x16A`	16×16	4,000
`TwoBasesBarracks16x16`	16×16	4,000
`basesWorkers24x24A`	24×24	5,000
`basesWorkers24x24ABarrack`	24×24	5,000
`basesWorkers32x32A`	32×32	6,000
`basesWorkersBarracks32x32`	32×32	6,000
`(4)BloodBathB`	64×64	8,000
`(4)BloodBathD`	64×64	8,000

The action abstraction used by A2N is also asymmetric as the number of scripts in each set can be different, allowing A2N to derive finer plans to units in either group.

V. EMPIRICAL EVALUATION

We evaluate the algorithms proposed in this paper to searching in asymmetrically-abstracted action spaces on μRTS [9]. Our empirical evaluation is divided into four parts. First, we evaluate the effectiveness of searching in within-states (Section V-A). Then, we evaluate different strategies for selecting the set of unrestricted units (Section V-B). We then evaluate GAB, SAB, and A3N against their baselines GAS, $\text{GAB}_\mathcal{P}$, SAS, $\text{SAB}_\mathcal{P}$, A1N and A2N (Section V-C). Finally, we compare GAB, SAB, and A3N against state-of-the-art search algorithms for RTS games (Section V-D).

1) Empirical Setting for μRTS: In μRTS players need to submit an action at every decision-point. Each player is allowed 100 milliseconds for planning in each state of the game (decision-point or within-state).

Every match in our experiments is limited by several game cycles, and the match is considered a draw once the limit is reached. The maximum number of game cycles is dependent on the map. We use the limits defined by Barriga et al. [21]. Table I shows the name of the maps, their sizes, and the maximum game cycles allowed. We use 10 maps of varied sizes in our experiments, from small maps created for research, to large maps used in commercial games (`BloodBathB` and `BloodBathD` are copies of StarCraft's BloodBath map).

Each tested algorithm plays against every other algorithm ten times in each map. To ensure fairness, the players switch their starting location on the map an even number of times.

The set of scripts used by PGS, SSS, GAB, SAB, and A1N is worker rush (WR), light rush (LR), heavy rush (HR), and ranged rush (RR) [4], [22]. A3N's set of scripts is composed of LR, HR, and RR. A3N uses a different set of scripts because preliminary results showed that the algorithm tend to perform better with LR, HR, and RR. We use LR as the default script for PGS, SSS, GAB, and SAB. All these scripts train units which are immediately sent to attack the enemy. The ABCD algorithm used in GAB and SAB uses the technique called

TABLE II
WINNING RATE OF GABw, SABw, AND A3Nw AGAINST THEIR BASELINES.

Algorithms	Map Size				
	8×8	16×16	24×24	32×32	64×64
GABw × GAB	60.0	100.0	90.0	95.0	55.0
SABw × SAB	55.0	90.0	70.0	70.0	80.0
A3Nw × A3N	65.0	70.0	75.0	75.0	70.0

scripted move ordering to allow for more pruning during search [13]. In our experiments, the ABCD search of GAB and SAB first searches the actions returned by WR for maps of size 8×8 and 16×16 and the actions returned by LR for the other maps before considering other actions.

Our results will be reported in terms of winning rate. The winning rate is computed by summing the total number of victories and half of the number of draws of each algorithm evaluated and then dividing this sum by the total number of matches played; the result of the division is then multiplied by 100.

A. Evaluating Within-State Search

In this section we evaluate empirically GABw, SABw, and A3Nw against GAB, SAB, and A3N. When defining an asymmetric action abstraction one needs to define a strategy for selecting the set of unrestricted units at every state of search. In this experiment we select player i's unit that is closest to an enemy unit as the only unrestricted unit. We describe and evaluate several domain-specific strategies for selecting unrestricted units in Section V-B. Table II shows the winning rate of each within-state variant against their baseline. The results are averaged by map size.

As expected, A3Nw defeats A3N on average on all map sizes tested. The within-states versions of GAB and SAB also defeat their baselines. Both GABw and SABw obtain winning rates equal or larger than 70% against their baselines in matches played on maps of size 16×16, 24×24, and 32×32. The winning rates are lower for the smaller 8×8 map. This is because the strategy encoded in the script WR is already very strong in this map and both GAB and SAB play a strategy that is similar to WR. GABw and SABw also play a strategy similar to WR, but they are able to provide better control to the units during combat, thus improving slightly the results.

Henceforth in this paper, all experiments with GAB, SAB, A3N and their baselines are with the version of the algorithms that search in within-states. We drop the 'w' from the algorithms' names to ease notation.

1) Within-State Version of Baselines: Given the strong empirical results of the within-state version of GAB, SAB, and A3N, we have also implemented the within-state versions of all baselines, GAS, GAB$_{\mathcal{P}}$, SAS, SAB$_{\mathcal{P}}$ A1N, and A2N.

B. Evaluating Strategies and Number of Unrestricted Units

Next, we describe and evaluate nine 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 the player'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) **Farthest from Centroid (FC)**. FC selects the N units that are farthest from the centroid of all player i's units.
2) **Closest to Centroid (CC)**. CC selects the N units that are closest to the centroid of all player i's units.
3) **Closest to Enemy (CE)**. CE selects the N units that are closest to an enemy unit at every decision-point.
4) **Farthest from Enemy (FE)**. FE selects the N units that are the farthest from an enemy unit.
5) **Less life (HP-)**. HP- selects the N units with the lowest hit points.
6) **More life (HP+)**. HP+ selects the units with more hit points at a given decision-point.
7) **High Attack Value (AV+)**. Let $av(u) = \frac{dpf(u)}{hp(u)}$, where $dpf(u)$ is the amount of damage per game cycle a unit can inflict to an enemy unit and $hp(u)$ is u's current amount of hit points.
8) **Low Attack Values (AV-)**. AV- selects the units with the lowest av-values.
9) **Random (R)**. R randomly selects N units. This strategy serves as a baseline for the other strategies.

We evaluate GAB, SAB, and A3N with the nine strategies described above for values of $N \in \{1, \cdots, 10\}$. We compare each algorithm with its baseline that searches in uniformly-abstracted spaces, PGS, SSS, and A1N. Each algorithm plays against its baseline ten times in each one of the ten maps. Table III shows the average winning rate of the algorithms for different strategies and values of N. The rows show the strategies used for selecting the unrestricted set while the columns show the size of the set.

We use a cell-coloring scheme in Table III to aid us understand the results. In our color scheme, the lowest winning rate in the table (17.5 for A3N with AV- and $N = 8$) has the lightest color and the largest winning rate (82.0 for GAB with AV- and HP- with $N = 3$) has the darkest color. The remaining cell colors are chosen as a linear interpolation of the colors of the two extremes.

All three algorithms tend to perform better with smaller values of N ($N \leq 5$). This is because for larger N the space becomes too large to allow the algorithm to encounter strong strategies under real-time constraints. Table III also shows that the algorithms searching with asymmetric action abstractions can outperform their baselines that search with uniform action abstractions. The largest winning rate obtained by GAB, SAB, and A3N are 82.0 (AV- or HP- with $N = 3$), 75.0 (FE with $N = 3$), and 78.3 (FC with $N = 4$), respectively. GAB, SAB, and A3N outperform their baselines even with the random strategy with $N = 1$. Henceforth, we use GAB with HP- and $N = 3$, SAB with with FE with $N = 3$ and A3N with FC and $N = 4$. Next, we explain the results presented in Table III using domain-dependent knowledge. The reader not interested in the problem domain should skip to Section V-C.

TABLE III
WINNING RATE OF VARIANTS OF GAB AGAINST PGS IN 100 MATCHES PLAYED IN 10 MAPS, 10 MATCHES FOR EACH MAP. THE ROWS DEPICT DIFFERENT STRATEGIES (STR.) AND THE COLUMNS DIFFERENT UNRESTRICTED SET SIZES (N).

GAB vs. PGS										
Strategy	Unrestricted Set Size N									
	1	2	3	4	5	6	7	8	9	10
CC	59.0	59.5	47.0	57.0	43.0	46.0	43.0	44.0	45.5	45.5
FC	68.0	55.0	50.5	46.0	44.0	41.0	43.0	40.0	54.0	41.0
CE	67.5	68.0	62.0	60.5	59.0	53.0	50.0	43.5	49.0	56.5
FE	76.0	75.0	69.0	74.5	67.0	54.5	52.5	38.5	46.5	41.5
AV-	77.0	74.0	82.0	79.0	73.0	70.0	68.0	64.0	53.0	54.0
AV+	69.0	71.0	76.0	74.0	70.0	77.0	74.5	57.5	55.5	67.0
HP-	68.0	72.0	82.0	76.0	66.0	73.0	65.5	63.0	67.0	61.0
HP+	61.5	57.0	57.0	39.5	43.0	36.0	47.5	35.5	38.0	36.0
R	63.5	47.5	44.5	45.0	47.5	42.5	49.0	45.5	56.5	48.5
SAB vs. SSS										
Strategy	Unrestricted Set Size N									
	1	2	3	4	5	6	7	8	9	10
CC	65.0	66.5	54.0	57.0	50.0	60.0	59.0	58.0	61.0	57.0
FC	56.0	61.5	51.0	61.5	55.0	56.5	51.0	54.0	58.5	56.5
CE	72.0	60.0	56.0	60.5	54.0	56.0	55.0	51.5	55.0	48.0
FE	63.0	73.0	75.0	69.0	59.0	60.5	60.0	58.0	57.0	55.5
AV-	70.0	69.0	73.0	67.0	67.0	63.0	70.0	65.0	54.0	60.0
AV+	66.0	74.0	72.0	71.0	71.0	68.0	65.5	66.0	61.5	61.5
HP-	71.5	71.0	72.0	67.0	61.0	67.5	59.0	69.0	66.0	68.0
HP+	54.0	58.0	57.0	52.5	50.5	50.5	58.0	59.0	60.5	54.5
R	66.5	54.0	60.0	56.5	62.0	56.0	58.0	62.0	59.5	60.0
A3N vs. A1N										
Strategy	Unrestricted Set Size N									
	1	2	3	4	5	6	7	8	9	10
CC	69.6	59.2	57.1	57.5	59.2	54.2	51.3	41.3	37.5	40.0
FC	68.8	72.5	71.7	78.3	71.7	68.8	68.3	64.2	55.4	62.5
CE	72.5	69.2	75.4	71.7	77.9	75.0	72.1	63.3	57.1	57.1
FE	59.2	64.2	56.3	55.0	56.3	45.0	40.8	35.4	25.8	26.7
AV-	37.9	27.9	27.1	25.0	26.7	24.6	20.0	17.5	18.3	22.9
AV+	21.7	40.4	48.8	65.0	60.8	56.7	62.1	60.8	57.1	56.7
HP-	24.2	38.3	55.4	58.3	59.2	57.9	60.8	64.6	61.7	51.7
HP+	31.7	22.1	21.3	30.8	24.6	24.6	23.3	33.3	35.0	32.1
R	69.6	69.6	63.3	70.4	65.0	55.8	55.0	55.4	52.9	52.1

GAB performs best with AV- and HP-, two dissimilar strategies. Strategy AV- allows GAB to provide a finer control to bases and barracks, as these units minimize the AV-value (they are unable to cause damage and have a large number of hit points). Strategy HP- allows GAB to provide a finer control to weaker units such as workers or combat units that have suffered damage. SAB also obtains good results with both AV- and HP-. These results are in contrast with A3N's, as A3N can be worse than A1N if using either AV- and HP-.

The discrepancy in results with the AV- strategy happens because both GAB and SAB use scripts to sort the actions explored in the ABCD search. For example, in maps of size 24×24, ABCD evaluates the action provided by the LR script before any other action. The LR strategy builds a barracks as soon as possible so that light units can be trained, and a barracks can only be built if the player "saves" resources. Due to the ABCD move ordering, both GAB and SAB are able to evaluate the sequence of actions to successfully build a barracks while using the AV- strategy. By contrast, A3N does not employ a move ordering approach and the actions needed to produce a barracks might not even be evaluated if one considers all legal actions of bases and barracks (as it happens with the AV- strategy). Moreover, A3N uses a playout function to evaluate actions that is shorter than the one used by GAB and SAB. As a result, even if A3N evaluates the action of building a barracks, the algorithm is shortsighted and thus unable to perceive the value of building such a structure. A3N also obtains poor results with strategy HP+ for exactly same reasons just described.

The discrepancy of results of GAB/SAB and A3N with strategy HP- can be explained by similar arguments. The HP- strategy allows the algorithms to mostly control workers, which are the units with the lowest hit point values. Workers are usually either battling the opponent or collecting resources. Due to its lack of move ordering, if providing a finer control to workers collecting resources, A3N often mistakenly sends such units to attack the opponent, thus interrupting their task. The move ordering used by GAB and SAB's ABCD search allows the algorithms to not harm the player's strategy for resource gathering.

Strategies that are strong for GAB are also strong for SAB. However, in contrast with GAB, SAB outperforms its baseline with almost any strategy and value of N. This happens likely because SAB has the weakest of the baselines. The SSS search is more limited than PGS because it is constrained to a type system. Thus, SAB's ABCD search can more easily improve upon the actions encountered by the algorithm's first step.

A3N performs best with the CE and FC strategies. Both strategies allow A3N to provide a finer control to units in direct combat with the enemy. A3N performs better with these strategies than GAB and SAB likely because A3N does not assume a model of the opponent and is thus more robust in combat scenarios. By contrast, GAB and SAB's ABCD search assume a model of the opponent and the algorithms might perform poorly if the opponent follows a strategy that is different than the one assumed during search.

C. Comparison with Baselines

In this section, we evaluate GAB, SAB and A3N against their baselines PGS, GAS, GAB$_{\mathcal{P}}$, SSS, SAS, SAB$_{\mathcal{P}}$, A1N and A2N. Table IV shows the results for GAB and SAB, while Table V shows the results for A3N. The numbers in the tables indicate the winning rate of the row player against the column player.

All three algorithms, GAB, SAB, and A3N, outperform their baselines. The results of GAB against GAB$_{\mathcal{P}}$, SAB against SAB$_{\mathcal{P}}$, and A3N against A1N demonstrate that algorithms that search with asymmetric action abstractions can substantially outperform their counterparts that search with uniform abstractions. The results of GAB against GAS and SAB against SAS demonstrate that the two-step search scheme of GAB and SAB can be effective as both GAS and SAS also search in asymmetrically-abstracted action spaces—the algorithms differ only in their search scheme. Finally, the superiority of

TABLE IV
WINNING RATE OF THE ROW PLAYER AGAINST THE COLUMN PLAYER. COMPARISON OF GAB AND SAB WITH THEIR BASELINES.

	PGS	GAS	GAB$_{\mathcal{P}}$	GAB	Avg.
PGS	-	78.5	84.0	19.0	60.5
GAS	21.5	-	52.0	8.0	27.2
GAB$_{\mathcal{P}}$	16.0	48.0	-	20.0	28.0
GAB	81.0	92.0	80.0	-	84.3
	SSS	SAS	SAB$_{\mathcal{P}}$	SAB	Avg.
SSS	-	85.0	73.0	26.0	61.3
SAS	15.0	-	27.5	8.0	16.8
SABp	27.0	72.5	-	26.0	41.8
SAB	74.0	92.0	74.0	-	80.0

TABLE V
WINNING RATE OF THE ROW PLAYER AGAINST THE COLUMN PLAYER. COMPARISON OF A3N WITH ITS BASELINES.

	A1N	A2N	A3N	Avg.
A1N	-	24.0	5.5	14.8
A2N	76.0	-	27.0	51.5
A3N	94.5	73.0	-	83.8

A3N against A2N demonstrates the effectiveness of generating asymmetric action abstractions by having a set of unrestricted units for which all unit-actions are considered during search. A2N's asymmetry relies on the strategies encoded in scripts as all units are restricted to a set of scripts. By contrast, A3N's asymmetry allows the search procedure to discover strategies different than those encoded in scripts by considering all unit-actions for a small set of units.

D. Comparison with State-of-the-Art Algorithms

In this section we evaluate GAB, SAB, A3N against the current state-of-the-art search-based methods for RTS games. Namely, we test the following algorithms: Portfolio Greedy Search (PGS) [7], Stratified Strategy Selection (SSS) [8], Adversarial Hierarchical Task Network (AHT) [23], an algorithm that uses Monte Carlo tree search and HTN planning; NaïveMCTS [9] (henceforth referred as NS); the MCTS version of Puppet Search (PS) [21], Strategy Tactics (STT) [24], Strategy Creation via Voting (SCV) [4], and four hard-coded scripts focused in rush, called Light rush (LR), Ranged rush (RR), Heavy rush (HR), and Worker Rush (WR) [22].

Table VI shows the winning rate of the row player against the column player. Overall, A3N wins more matches than any approach tested, suggesting that if a large diversity of maps and opponents are considered, then A3N is the current state-of-the-art in μRTS. GAB is a close second place with an average winning rate of 75.3. In terms of direct confronts A3N obtains a winning lower than 50.0 only against STT; GAB obtains a winning lower than 50.0 only against A3N. SAB and STT obtain similar average winning rate, 65.4 and 67.1, respectively. Both A3N and STT are able to defeat the weaker opponents by a large margin (e.g., both win almost all matches against the script HR), but A3N is able to better exploit stronger opponents such as PGS, SSS, SAB, and GAB. A3N achieves a larger overall winning rate than STT because of the results against these opponents.

The size of the search space of μRTS matches is mainly defined by the structure and the size of the map in which the matches take place. Matches played in smaller maps tend to be quicker, with fewer units being controlled by the players at any moment of the match. Matches played in larger maps tend to take longer and the players control a larger number of units, thus increasing size of the search space. The distinction between small and large maps is important because algorithms searching in unabstracted spaces tend to perform better in smaller maps than algorithms that search in uniformly-abstracted spaces. This is because the search space is small enough for the algorithms to encounter strong strategies while accounting for all legal actions. However, the strategy of searching in unabstracted spaces does not scale to large maps, where algorithms that search in uniformly-abstracted spaces tend to perform better due to their search being focused on the set of promising actions returned by scripts. We hypothesize that asymmetric action abstractions allow search algorithms to derive strong strategies in both small and large maps.

VI. CONCLUSIONS

In this paper we introduced asymmetric action abstractions for multi-unit zero-sum extensive-form games. We also introduced A2N, A3N, GAB, and SAB, four search algorithms for searching in asymmetrically-abstracted spaces. Similarly to uniformly-abstracted spaces, asymmetric abstractions also use domain-knowledge in the form of scripts. However, in contrast with uniform abstractions, which restrict all units to the unit-actions returned by the scripts, asymmetric action abstractions restrict only a subset of the units—the restricted units. Algorithms searching with asymmetric action abstractions account for all legal unit-actions of the remaining units—the unrestricted units. As a result, the strategy derived by search algorithms are focused on the unrestricted units, as the algorithms are able to derive finer plans for such units. Asymmetric action abstractions can be seen as an attention scheme, where the search "pays more attention" to a subset of units.

We evaluated our algorithms with an extensive set of experiments on μRTS. Our results suggest that A3N is the current state-of-the-art algorithm in this domain if one considers a large diversity of maps and opponents, similarly to the setting used in the μRTS annual competition [6]. If one considers large maps such as those used in commercial games, then GAB presented the strongest results. These algorithms were used to develop a system that won the **IEEE MicroRTS Competition in 2018** in combination with techniques we developed for inducing uniform action abstractions [5] .[3]

Although we performed our experiments on μRTS, the ideas of this paper are general and could be applied to

[3]https://sites.google.com/site/micrortsaicompetition/competition-results/2018-cig-results

TABLE VI
COMPARISON OF GAB, SAB, AND A3N WITH CURRENT STATE-OF-THE-ART SEARCH-BASED METHODS.

	HR	RAR	AHT	NS	WR	A1N	SSS	PGS	PS	SCV	LR	STT	SAB	GAB	A3N	Avg.
HR	-	85.0	13.5	57.5	15.0	17.5	30.0	23.0	16.0	18.0	12.5	5.0	9.0	8.0	3.0	22.4
RAR	15.0	-	57.0	78.0	60.0	24.0	16.0	11.0	3.0	20.0	0.0	16.0	11.0	12.0	1.0	23.1
AHT	86.5	43.0	-	18.5	10.0	13.0	27.0	24.5	39.5	28.5	39.5	9.0	19.5	19.5	2.0	27.1
NS	42.5	22.0	81.5	-	41.0	35.0	22.0	20.0	28.0	26.5	20.0	20.0	27.0	23.0	6.5	29.6
WR	85.0	40.0	90.0	59.0	-	29.0	49.0	43.0	40.0	50.0	35.0	38.5	27.5	31.5	26.5	46.0
A1N	82.5	76.0	87.0	65.0	71.0	-	36.5	34.0	49.0	46.0	34.0	28.0	24.0	26.0	11.5	47.9
SSS	70.0	84.0	73.0	78.0	51.0	63.5	-	54.0	42.0	37.0	28.0	27.0	32.0	17.0	16.0	48.0
PGS	77.0	89.0	75.5	80.0	57.0	66.0	46.0	-	45.5	42.5	46.0	27.5	32.0	13.0	18.5	51.1
PS	84.0	97.0	60.5	72.0	60.0	51.0	58.0	54.5	-	42.5	47.0	30.0	29.0	26.0	28.5	52.9
SCV	82.0	80.0	71.5	73.5	50.0	54.0	63.0	57.5	57.5	-	61.0	40.5	44.0	34.5	23.5	56.6
LR	87.5	100.0	60.5	80.0	65.0	66.0	72.0	54.0	53.0	39.0	-	53.0	35.5	17.0	41.0	58.8
STT	95.0	84.0	91.0	80.0	61.5	72.0	73.0	72.5	70.0	59.5	47.0	-	53.0	29.0	52.5	67.1
SAB	91.0	89.0	80.5	73.0	72.5	76.0	68.0	68.0	71.0	56.0	64.5	47.0	-	36.5	23.0	65.4
GAB	92.0	88.0	80.5	77.0	68.5	74.0	83.0	87.0	74.0	65.5	83.0	71.0	63.5	-	47.5	75.3
A3N	97.0	99.0	98.0	93.5	73.5	88.5	84.0	81.5	71.5	76.5	59.0	47.5	77.0	52.5	-	78.5

other games. For example, in collectible card games such as Hearthstone [25] and Magic: The Gathering [26] the player has to decide on the action of several cards. Algorithms could use asymmetric action abstractions to focus their search on a subset of the cards. The ideas introduced in this paper might also be applied in problems other than games. For example, a robotic system that controls several actuators while trying to accomplish a task can benefit from asymmetric action abstractions. This is because some actuators might require a finer control than the others.

REFERENCES

[1] R. O. Moraes and L. H. S. Lelis, "Asymmetric action abstractions for multi-unit control in adversarial real-time scenarios," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI, 2018, pp. 876–883.

[2] R. O. Moraes, J. R. H. Mariño, L. H. S. Lelis, and M. A. Nascimento, "Action abstractions for combinatorial multi-armed bandit tree search," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI, 2018, pp. 74–80.

[3] R. O. Moraes, J. R. H. Mariño, and L. H. S. Lelis, "Nested-greedy search for adversarial real-time games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2018, pp. 67–73.

[4] C. R. Silva, R. O. Moraes, L. H. S. Lelis, and Y. Gal, "Strategy generation for multi-unit real-time games via voting," *IEEE Transactions on Games*, 2018.

[5] J. R. Marino, R. O. Moraes, C. Toledo, and L. H. Lelis, "Evolving action abstractions for real-time planning in extensive-form games," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 2330–2337.

[6] S. Ontañón, N. A. Barriga, C. R. Silva, R. O. Moraes, and L. H. Lelis, "The first microrts artificial intelligence competition." *AI Magazine*, vol. 39, no. 1, 2018.

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

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

[9] S. Ontañón, "Combinatorial multi-armed bandits for real-time strategy games," *Journal of Artificial Intelligence Research*, vol. 58, pp. 665–702, 2017.

[10] D. Churchill and M. Buro, "Hierarchical portfolio search: Prismata's robust AI architecture for games with large search spaces," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2015, pp. 16–22.

[11] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[12] M. Campbell, A. Hoane, and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57 – 83, 2002.

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

[14] L. Atkin and D. Slate, "Computer chess compendium," D. Levy, Ed. Berlin, Heidelberg: Springer-Verlag, 1988, ch. Chess 4.5-The Northwestern University Chess Program, pp. 80–103. [Online]. Available: http://dl.acm.org/citation.cfm?id=61701.67010

[15] A. L. Zobrist, "A new hashing method with application for game playing," 1990.

[16] R. E. Korf, "Macro-operators: A weak method for learning," *Artificial Intelligence*, vol. 26, no. 1, pp. 35–77, 1985.

[17] J. E. Laird, P. S. Rosenbloom, and A. Newell, "Chunking in soar: The anatomy of a general learning mechanism," *Machine Learning*, vol. 1, no. 1, pp. 11–46, 1986.

[18] G. A. Iba, "A heuristic approach to the discovery of macro-operators," *Machine Learning*, vol. 3, no. 4, pp. 285–317, 1989.

[19] R. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, pp. 181–211, 1999.

[20] M. C. Machado, M. G. Bellemare, and M. Bowling, "A laplacian framework for option discovery in reinforcement learning," in *Proceedings of the International Conference on Machine Learning*, 2017, pp. 2295–2304.

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

[22] M. Stanescu, N. A. Barriga, A. Hess, and M. Buro, "Evaluating real-time strategy game states using convolutional neural networks," in *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*. IEEE, 2016, pp. 1–7.

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

[24] N. A. Barriga, M. Stanescu, and M. Buro, "Combining strategic learning and tactical search in real-time strategy games," *The AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2017.

[25] A. Dockhorn and S. Mostaghim, "Introducing the hearthstone-ai competition," 2019.

[26] C. D. Ward and P. I. Cowling, "Monte carlo search applied to card selection in magic: The gathering," in *Proceedings of the International Conference on Computational Intelligence and Games*. IEEE Press, 2009, pp. 9–16.